INFORMATION RETRIEVAL AND PERSONALIZATION

LARGE LANGUAGE MODELS EVALUATION IN PREDICTING CHECKSTYLE CONFIGURATION FROM JAVA CODE INPUTS

STUDENTS: Alessio Mattiace, Claudio Saponaro

INDEX

INTRODUCTION

This project presents a comprehensive methodology for evaluating the capability of Large Language Models (LLMs) to generate and validate Checkstyle configuration files for Java code style enforcement.

Our study, which constitutes a component of the broader research detailed in "LLM-Driven Checkstyle Configuration: Empowering Developers to Simplify Code Style Alignment", examines the efficacy of artificial intelligence in learning style conventions.

This study focuses on the systematic examination of Checkstyle.xml configurations generated by LLMs, assessing their conformity to established reference standards found in existing repositories.

Through comparative analysis, the research aims to evaluate the model's ability to identify, interpret, and apply predefined coding conventions and best practices. Specifically, the investigation explores the degree of alignment between LLM-generated configurations and the stylistic patterns inherent in the input Java code, using existing Checkstyle.xml true files as benchmark references.

Our research experiment aims to address a specific question:

“Is an LLM, able to generate a Checkstyle configuration file by inferring style rules based on the observation of a Java language code?”

To address this research question, we have implemented a four-phase methodological framework:

- First, we established a comprehensive **dataset** comprising repositories that utilize Checkstyle.xml for Java code style verification.
- Subsequently, we extracted both the **Checkstyle.xml** configurations and associated **Java source files** from each repository.
- The third phase involved leveraging **LLM APIs** to generate predicted Checkstyle configurations based on the analyzed codebases stylistic characteristics.
- Finally, we developed and implemented a rigorous **evaluation methodology** to assess the accuracy and effectiveness of the generated configurations.

The findings provide valuable insights into the potential and limitations of LLMs in maintaining consistent coding standards across software projects.

SECTION 1 – State-Of-Art Analysis

The task of evaluating the proficiency of LLMs in generating Checkstyle configurations requires a deep understanding of code stylization, authorship attribution, and static analysis tools. This section explores existing research on these topics, drawing insights from relevant literature to contextualize the challenge of assessing whether an LLM captures document patterns or acts randomly.

In our project we have followed the principle explained in [1], which presents a system that applies machine learning to deduce coding style from a code corpus and subsequently transforms arbitrary code into the learned style.
The system extends beyond simple formatting to include naming conventions, ordering, and alternative programming constructs, demonstrating that a learned style can be effectively applied to new code instances.

This concept is crucial for our study as it suggests that if a Checkstyle configuration is consistently generated, an LLM might be capturing style patterns rather than acting randomly; this important step is well documented in the loss construction and evaluation of our project.

The paper also suggested that the more code provided to the ML unlock higher capabilities of understanding the pattern within the code style in java codes in the training set, however this is related to the fact that the author trained a ML model from scratch using this java codes.
The author experimented a vast class of classifiers including: Decision Trees (J48-Weka), SVM, Naïve Bayes, Maximum Entropy Models and Instance-Based Learning.
However we have not followed this approach, instead we have used a pre-trained architecture such as a LLM to experiment if using this architecture works fine as well although not specifically trained on this task.

Static Code Analysis Tools (SCATs) identify potential issues in source code without execution: an automatized approach presented in [2], rely on past code review comments to automatically configure Checkstyle, demonstrating that natural language processing (NLP) and machine learning techniques can effectively map review comments to Checkstyle rules. Auto-SCAT achieves a precision of 75.1% and a recall of 74.6% suggesting that automated systems can meaningfully predict Checkstyle configurations.

Another approach is explained in [3], where is shown an AI-driven approach to detecting and fixing code style inconsistencies without manual configuration. Using unsupervised machine learning, the system analyzes a project's codebase, learns its stylistic conventions, and generates interpretable rules to ensure consistency. Based on decision forests, it identifies dominant style patterns and suggests automated corrections, reducing code review effort. Unlike traditional linters, STYLE-ANALYZER adapts dynamically to each repository's preferences. Tested on large open-source projects, it enhances code quality by providing clear, context-aware style recommendations.

In [4], the ICAA (Intelligent Code Analysis Agent) technology is illustrated; it allows for static code analysis to detect errors or potential inefficiencies by combining LLMs like GPT-3 or GPT-4 with traditional code analysis techniques. Although it is not a stylistic configuration analysis, it can be useful for understanding how to leverage LLMs to comprehend patterns in a program’s code.

SECTION 2 - General Structure Of A Checkstyle: Modules And Properties

The Checkstyle configuration file is an XML document that defines a set of rules to ensure that a project's Java code follows a consistent style. This tool is particularly useful in collaborative development environments, where maintaining a common standard facilitates code comprehension, improves maintainability, and reduces the risk of errors caused by inconsistent formatting.

Each Checkstyle configuration file is based on a modular architecture, where each module is responsible for a specific check. Some modules operate on the entire source code, while others analyze the syntactic structure of each Java file. Checkstyle customization is achieved through properties assigned to the modules, allowing developers to adjust the severity level of reports or define precise stylistic rules.

Hierarchical Structure

At the core of the Checkstyle configuration is the root module, called Checker. This module is essential and serves as the starting point for all style checks within a project. Its role is to coordinate the checks, applying general rules that may affect the entire Java file.

Within the Checker, there are two categories of modules. Global modules handle checks that are independent of the syntactic structure of the code, such as whitespace verification, mandatory copyright headers, or empty line management. Another fundamental module is TreeWalker, which is responsible for a more in-depth analysis of the code’s syntax. This module traverses the Abstract Syntax Tree (AST) of the Java file, applying specific rules to each code element, such as methods, classes, comments, and code blocks.

An important aspect to consider is that modules must follow a precise hierarchy. Global modules cannot be children of TreeWalker, while all checks that depend on the syntactic structure of the code must be defined within the TreeWalker. This distinction is crucial to ensure that Checkstyle functions correctly and analyzes the code in a structured and effective manner.

Module Customization

One of Checkstyle's strengths is its high level of customization, which allows checks to be adapted to the specific needs of a project. Each module can be configured through various properties that determine its behaviour.

For example, the severity property allows setting the importance level of a violation, classifying it as an error, warning, or informational message. The format property, on the other hand, enables the definition of precise rules for the naming conventions of classes, methods, and variables, ensuring, for instance, that variable names follow the camelCase format. There are also properties that regulate indentation, the presence of Javadoc comments, the use of curly braces, and many other stylistic conventions.

In addition to configuring modules, it is possible to exclude parts of the code from analysis. Checkstyle provides two main exclusion mechanisms:

- BeforeExecutionExclusionFileFilter, which allows ignoring entire files before the check is performed.
- Suppressions File, which enables more targeted exclusions by specifying files, classes, or methods that should not be subject to certain checks.

This flexibility is particularly useful when working with automatically generated code or third-party libraries that do not need to adhere to the same stylistic constraints as the rest of the project.

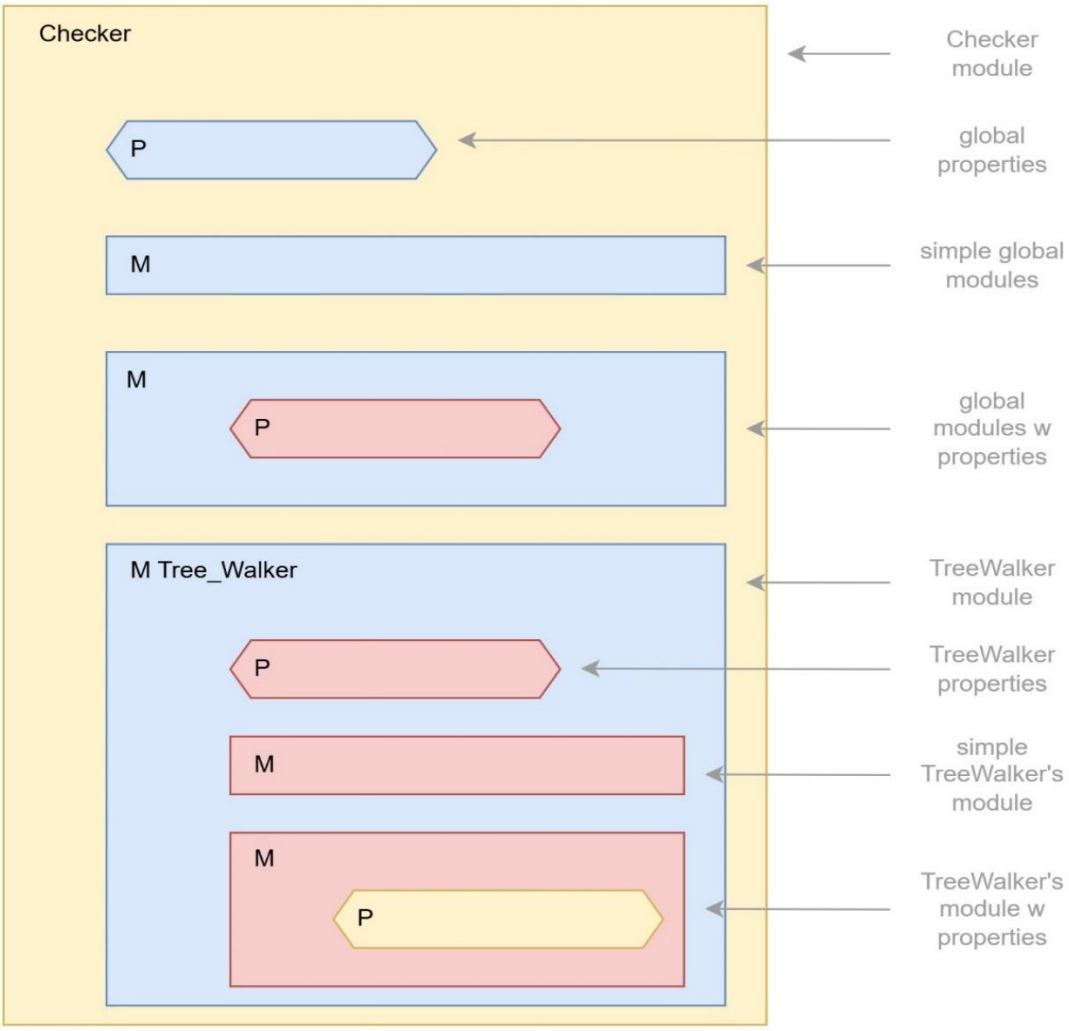

Figure 2.1 - General structure of a Checkstyle: modules and properties

SECTION 3 – Creation And Analysis Of The Starting Dataset

The first phase of the work was dedicated to the creation of a dataset used for the analysis and evaluation of the ability of using a LLM to correctly generate a Checkstyle.xml configuration file.
Specifically, the dataset must contain links to GitHub repositories that already adopt Checkstyle for style checking in Java files. The analysis of these repositories allows for the extraction of preliminary information and useful statistics to understand trends in the real-world usage of Checkstyle and to build a solid foundation for evaluating the configurations generated by the LLM.

3.1 – Initial Solution, Challenges, And Final Approach

Initial approach

The initial idea for dataset creation involved using REST APIs to send requests to GitHub and automatically retrieve all files named *Checkstyle.xml*, identifying their respective repositories.
The collected information for each repository included:

- Repository URL
- Checkstyle.xml file URL
- Number of stars
- Number of forks
- Number of pull requests
- Number of commits
- Number of open issues

Analyzing these metadata would allow for a statistical study on the adoption distribution of Checkstyle in open-source projects, aiming to select a representative set of repositories for our experiment.

Challenge N.1: Exceeding GitHub API Request Limits

One of the first difficulties encountered was exceeding the request limit imposed by the GitHub API, which triggers a *429 Too Many Requests* error when the maximum number of calls per time unit is surpassed. A possible workaround was to introduce a waiting period between requests, but this approach made the process excessively slow.

Solution adopted:
We decided to use the PyGithub library, which allows authentication via a personal access token (PAT). This enabled a significantly higher number of requests compared to the standard REST API, greatly accelerating the data collection process.

Challenge N.2: Inability to Trace Parent Repository of Checkstyle.xml files
Another significant issue was that using PyGithub, it was not immediately possible to trace the parent repository of a Checkstyle.xml file found via query.

Solution adopted:
We modified the search process: instead of directly searching for Checkstyle.xml files, we iterated through each repository containing Java code to verify whether it included a Checkstyle configuration file. This approach provided more structured results, though it increased query execution time.

Challenge N.3: GitHub Query Result Limitation
Another limitation imposed by GitHub is that a single query can return a maximum of 1000 results. This posed a problem since our goal was to collect as many repositories as possible.

Solution adopted:
Run, iteratively, several queries through the API: each of them searches on different last modification periods.
In this way, the result of an iteration, contains all repositories last modified in a certain time interval; the interval was set to 30 days and the start date to 01/01/2014.

Challenge N.4: Low-Quality or Irrelevant Repositories in the Dataset
Many of the identified repositories were of low quality or had little relevance to the analysis, either due to low usage or incomplete Checkstyle configurations.

Solution adopted:
We introduced a relevance filter, including only repositories with at least ten stars. This threshold helped exclude minor, insignificant repositories, allowing us to focus on projects actively adopted by a community of developers.

Final approach:

The Dataset_creator function enables the systematic collection of GitHub repositories that utilize Checkstyle for style checking in Java files. The dataset creation process involves a targeted analysis of repositories to extract structured information useful for evaluating Checkstyle configurations. The final dataset comprises a selection of relevant repositories characterized by a consistent adoption of Checkstyle, ensuring a representative sample for studying stylistic configurations. For each selected repository, key metadata such as the number of stars, forks, commits, and pull requests are gathered to identify actively maintained projects of community interest. The adopted approach ensures dataset quality through filtering criteria based on popularity and the completeness of Checkstyle configurations, facilitating an in-depth analysis of usage trends and an effective evaluation of configurations generated by the LLM model.

3.2 – Analysis Of The Obtained Dataset

Distribution of Repository Metrics:
After applying the strategies described above, it was possible to construct a dataset containing 809 GitHub repositories with a Checkstyle.xml file: of these, only 636 were found to be valid (they compile without style errors), so the dataset was reduced to ensure consistent data analysis.
The analysis of the collected repository metrics revealed the following distribution:

Statistic	# Stars	# Issues	# Pulls	# Commits	# Forks
Mean	434.78	26.16	5.62	1,161.42	120.95
Std. Dev.	2,721.12	101.02	28.56	5,020.92	834.48
Min	11	0	0	1	0
25%	22	2	0	74	8
50%	48	7.5	1	246.5	20
75%	157	21	5	790.5	52.25
Max	60,361	2,099	661	83,579	19,527

Figure 3.1 – Dataset analysis.

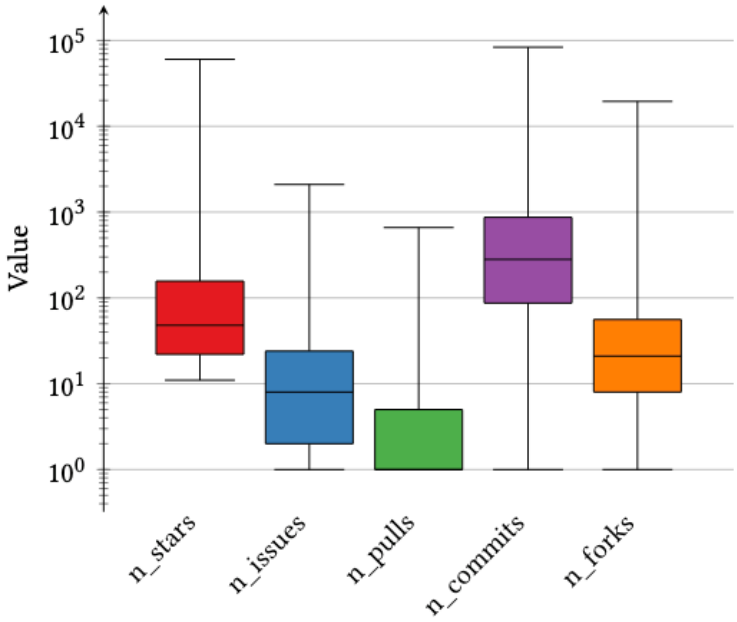

Figure 3.2 – Boxplot analysis.

The analysis reveals that:

- Most of the selected repositories have a star count ranging between 22 and 157, with some outliers exceeding 60,000 stars.
- The number of commits varies significantly, indicating the presence of both highly mature projects and relatively new repositories.
- The distribution of forks and pull requests suggests that many of the selected projects are actively developed and maintained.
- The most relevant repositories in terms of popularity tend to be fewer in number.

To facilitate the next phase of analysis, the repositories have been sorted based on the number of stars.

	repo_url	n_stars	n_issues	n_pulls	n_commits	n_forks	checkstyle_url
1	https://github.com/TheAlgo…	60361	5	4	2671	19527	https://github.com/TheAl
2	https://github.com/redisso…	23466	368	42	10038	5385	https://github.com/redis
3	https://github.com/Tencent…	17200	534	14	853	3339	https://github.com/Tence
4	https://github.com/Konloch…	14746	85	4	1448	1153	https://github.com/Konlo
5	https://github.com/orhanob…	13841	81	11	146	2142	https://github.com/orhar
6	https://github.com/zhihu/M…	12528	466	57	223	2065	https://github.com/zhihu
7	https://github.com/Tencent…	11830	44	2	372	1610	https://github.com/Tence
8	https://github.com/lingoch…	11053	181	10	834	2205	https://github.com/lingo
9	https://github.com/wildfir…	7966	12	9	2357	1804	https://github.com/wildf
10	https://github.com/apache/…	5659	503	163	11818	1031	https://github.com/apach

Figure 3.3 - Final dataset ordered by star number (first 10 rows).

SECTION 4 - Pipeline For Generating And Evaluating The Generated Checkstyle

In this section, all the various phases of the framework for evaluating LLMs in generating a Checkstyle.xml configuration file are illustrated.

Below is a graphical representation of the final architecture (In BLUE the different stages of the pipeline, in GREEN the functions used in a specific step, in RED the output obtained after each step):

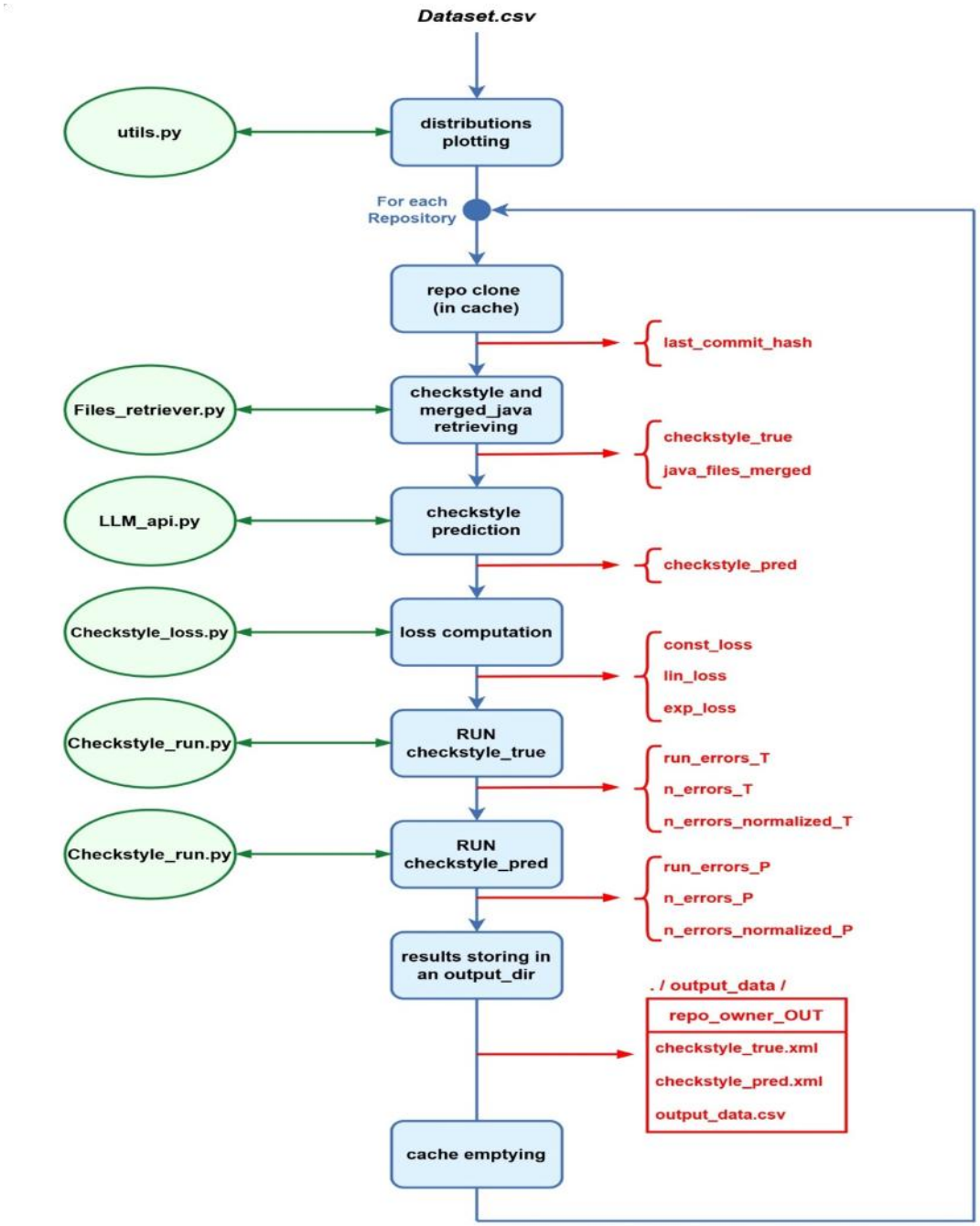

Figure 4.1 - Final architecture of the generated Checkstyle evaluation framework.

4.1 – Java Files And Checkstyle Ground Truth Retrieving

The primary goal of this phase is to efficiently extract Java source files from software repositories while simultaneously retrieving their corresponding Checkstyle configuration files. This process is crucial for analyzing and inferring the coding style rules that govern a given project. The extracted Java files serve as input for the LLM, enabling them to generate an optimized Checkstyle configuration file that aligns with the project's stylistic conventions.

Initial Approach

The initial implementation involved iterating through all files in a GitHub repository using the get_repo function from the **PyGithub** library, with a focus on the main branch. The workflow was structured as follows:

1. Retrieve the **Checkstyle.xml** file and all **Java files** from the repository.
2. **Merge all retrieved Java files** into a single concatenated file.
3. Store these extracted files in a dedicated **files_storing** folder for further processing.

Challenge N.1: Handling Suppressed Files

In certain repositories, a *suppressions.xml* file is present, which defines exceptions for specific files or code elements, preventing them from being checked against certain style rules. This introduced a critical issue: some files listed in the suppression rules might have been erroneously included in the dataset.

Solution Adopted:

- Retrieve the *suppressions.xml* file from the repository, if present.
- Parse its structure using Python's *xml.etree.ElementTree* library to extract file paths subject to suppression.
- Filter out these files from the dataset before they are passed to the LLM.

This step ensures that only relevant Java files are included in the style-checking process, maintaining the integrity of the extracted dataset.

Challenge N.2: Exclusions Defined in Checkstyle.xml

Another complication arose when certain repositories included a *BeforeExecutionExclusionFileFilter* module within the Checkstyle.xml configuration. This module defines regex patterns that specify files to be excluded from style enforcement, meaning some Java files were not meant to be analyzed at all.

Solution Adopted:

- Parse the Checkstyle.xml file to identify the *BeforeExecutionExclusionFileFilter* module.
- Extract the regex patterns defined within the module.
- Match these patterns against the list of retrieved Java files and exclude any that fit the criteria.

This filtering mechanism ensures that the LLM only receives Java files that are actually subject to Checkstyle rules, preventing misinterpretations caused by excluded files.

Challenge N.3: API Constraints and Redundant Repository Access

The initial approach relied on retrieving Java files individually via API requests using PyGithub. However, this method introduced two major inefficiencies:

1. **Exponential Backoff Delays:** The PyGithub library enforces an exponential backoff strategy for failed API requests, leading to delays and inefficiencies, particularly when handling multiple repositories or exceeding GitHub's API rate limits.
2. **Redundant files retrieving:** A later phase of the process required cloning the entire repository to execute the Checkstyle on each file; so, accessing files via the APIs, resulted in unnecessary computational overhead.

Solution Adopted:

To mitigate these inefficiencies, a refined approach was developed, integrating a caching mechanism and optimized repository cloning:

1. **Repository Cloning with Minimal Depth:** Instead of retrieving individual files, the entire repository is cloned into a cache directory at the start of the pipeline. To reduce storage and processing overhead, only the latest commit is retained using *depth=1*. This ensures that only the most recent version of the codebase is analyzed while eliminating redundant retrieval operations.
2. **Commit Hash Preservation for Reproducibility:** To maintain consistency across different executions of the pipeline, the hash of the latest commit is stored. This allows future evaluations to be conducted on the exact same code version, ensuring reproducibility in style enforcement assessments.

This optimized approach significantly improves efficiency by reducing the number of API requests, minimizing unnecessary computations, and ensuring consistent results in Checkstyle evaluations.

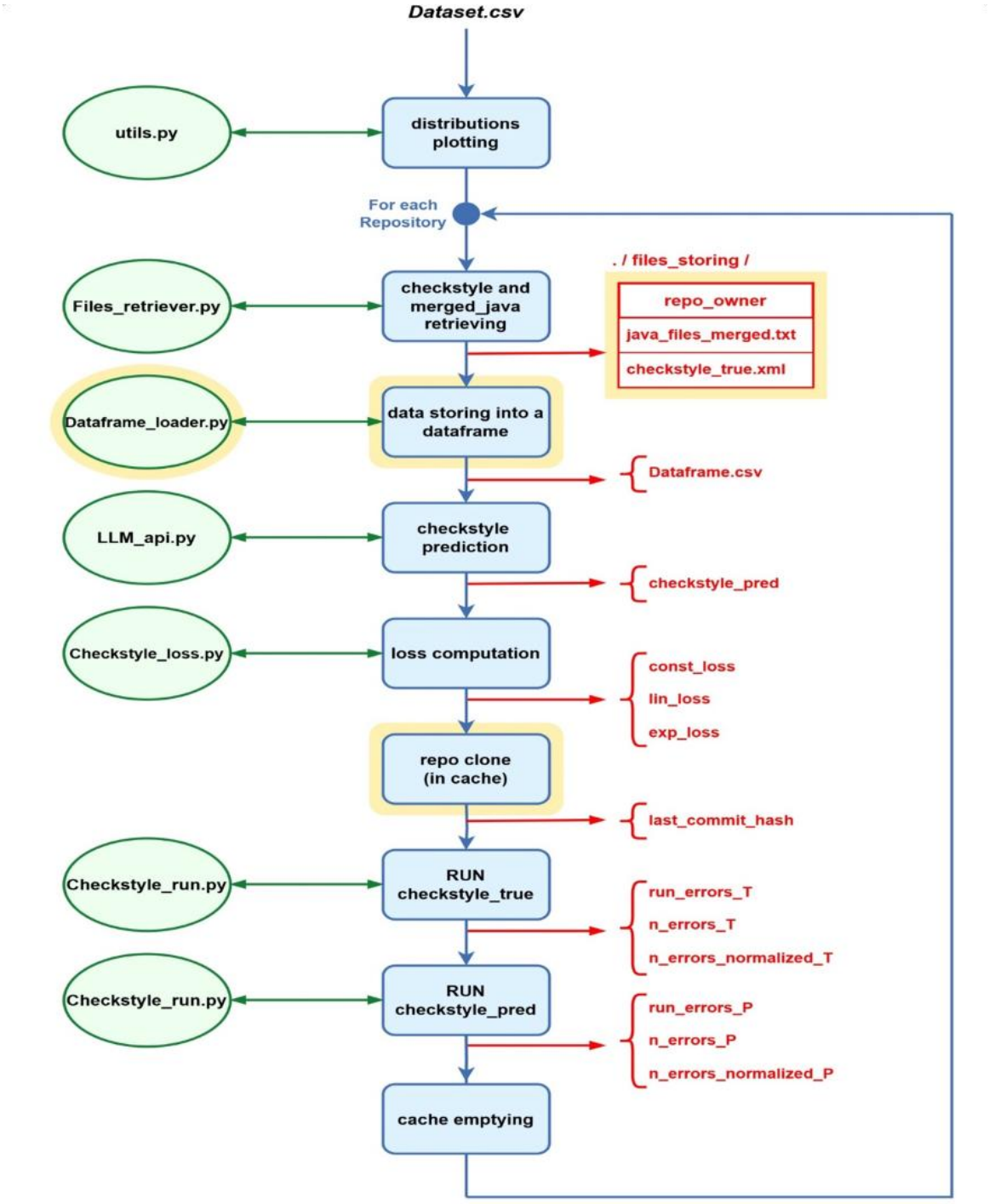

Figure 4.2 – Initial architecture of the generated Checkstyle evaluation framework (In yellow all the components modified in the next steps for code optimization).

Final Approach

The implemented function is designed to extract Java source files and their associated Checkstyle configurations from software repositories, optimizing for efficiency, consistency, and reproducibility in style analysis. The system clones the entire repository with minimal depth to capture the most recent code version, thereby avoiding redundant API accesses. After acquiring the files, filters are applied to exclude those subject to suppression rules, whether defined in a suppressions.xml file or specified within the Checkstyle configuration. To ensure reproducibility of analyses, the hash of the latest commit is preserved, allowing future evaluations on the same code version. This process ensures that the LLM receives only Java files that are effectively subject to the project's style rules, enhancing accuracy in generating a checkstyle.xml file consistent with the conventions adopted in the analyzed code.

4.2 - Checkstyle Generation Trough LLM

The primary goal of this module is to leverage API calls to an LLM to generate a Checkstyle configuration file that accurately reflects the coding style of a given Java codebase.

The initial approach adopted for interacting with the LLM involved using **LiteLLM**: an abstraction library that simplifies interactions with multiple LLM providers while maintaining a uniform API request structure.

This approach allowed flexibility in testing different models without requiring substantial modifications to the underlying implementation.

The LLM model selected for this task was **Gemini 1206 experimental**, a powerful model developed by Google.

The API was configured to process a prompt that included a list of Java files extracted from the same repository, ensuring that all input files adhered to the same stylistic conventions. The LLM was then expected to infer the most appropriate Checkstyle configuration based on these files.

Challenge N.1: Selecting the Optimal LLM Model

One of the initial challenges encountered was selecting the most appropriate model for Checkstyle inference. Google offers multiple versions of Gemini, including **Gemini 1.5**, **Gemini 1206**, and **Gemini 2.0 Flash**.

Solution Adopted:
To maximize the accuracy of Checkstyle generation, Gemini 1206 (experimental version) was chosen primarily for its extensive context window capable of processing up to 2 million tokens per prompt. This was crucial, as the system needed to analyze multiple Java files simultaneously to identify stylistic patterns.

Challenge N.2: Inconsistent Checkstyle Structure in Predictions

A recurring issue was the inconsistency in the generated Checkstyle.xml configurations. The LLM often produced configurations that lacked a structured hierarchy or misclassified certain style rules.

Solution adopted:
To improve the accuracy and consistency of the generated Checkstyle files, the prompt was refined to include:

- A detailed explanation of each module and its function.
- Module inference heuristics**,** explaining which modules can be predicted based on the Java code provided.

This information is stored in a file named *modules_explanation.txt* and added to the prompt. By incorporating these elements, the LLM was guided to produce a more accurate and structured XML file.

Challenge N.3: Lack of Reproducibility in Predictions

LLMs exhibit inherent non-determinism due to their probabilistic nature. This resulted in slight variations in Checkstyle configurations across multiple runs, making it difficult to validate and reproduce outputs.

Solution adopted:
To enhance reproducibility, the **temperature**, used in the structure of the request, parameter of the LLM API was set to 0. This configuration ensures that the model consistently predicts the most probable token at each step, removing variability in generated responses.

Challenge N.4: Handling Large Java Codebases

Some Java projects contained a vast number of files, leading to a concatenated input exceeding the model's maximum token limit. This caused outright API failures.

Solution adopted:
Limiting the concatenated Java code to 100,000 tokens.

Rather than employing arbitrary truncation, the system ensured that each included Java class contributed meaningfully to the inference process without exceeding the limit.

Challenge N.5: API Rate Limits and Error 429

Frequent API requests led to rate-limiting errors, causing interruptions in Checkstyle generation.

Solution adopted:
To mitigate this, a **pool of API keys** was implemented. When an API key reached its request limit, the system automatically switched to the next available key, ensuring uninterrupted processing.

Final approach

The function implements an automated system for generating a Checkstyle.xml configuration file from a given Java codebase. Utilizing API calls to an LLM, the system analyzes Java files within a project to infer stylistic rules consistent with the existing code. To ensure a structured and accurate output, the process involves constructing an optimized prompt that includes a detailed description of Checkstyle modules, an analysis of patterns present in the code, and output constraints to maintain consistency in the generated XML file. Additionally, to enhance the system's reliability and reproducibility, strategies are employed to control response variability and manage the volume of code analyzed.

4.3 – Loss Function For Generated Checkstyle Evaluation

The goal of this section is to define a robust loss function that can systematically evaluate the accuracy of a Checkstyle.xml file generated by an LLM, comparing it to the actual Checkstyle.xml configuration used in a predefined project. The loss function must penalize both the omission of relevant modules and the inclusion of irrelevant ones, while also accounting for the inherent difficulty in inferring certain modules purely from Java source code.

Initial Approach: Simple Loss Function
As an initial attempt, we designed a SimpleLoss function that:

- Traverses the syntax tree of the Checkstyle.xml files.
- Computes the percentage of modules correctly predicted by the LLM (modules present in both Checkstyle_pred and Checkstyle_true).
- Uses this percentage as an approximation of the LLM's accuracy in module selection.

While straightforward, this approach proved insufficient in capturing the true correctness of the generated Checkstyle configurations.

Challenge N.1: Overestimation of Accuracy

A model that aims to maximize the percentage of correct modules may be incentivized to predict as many modules as possible, assuming that the majority of them are present in Checkstyle_true. This would result in an excessive number of false positives (FP), leading to an overestimation of accuracy which doesn't reflect the real accuracy of the model

Solution adopted:
A more sophisticated function must be introduced, one that separately accounts for:

- **True Positives (TP):** Correctly predicted modules.
- **False Positives (FP):** Incorrectly predicted modules.
- **False Negatives (FN):** Missing modules that should have been predicted.

Three sets are built, one for each case: each module can be stored in one of them. Each of these components should contribute to the final loss, ensuring a balanced evaluation.

Challenge N.2: Difficulty in inferring certain Checkstyle modules

Some Checkstyle rules can be easily inferred from the Java source code (e.g., Indentation, WhitespaceAround), while others require project-specific knowledge and cannot be derived from the source code alone (e.g., JavadocMethod, HeaderCheck).

Solution Adopted:

We introduce adaptive penalization based on module predictability.
Modules that are inherently difficult or impossible to infer should have a lower penalty when misclassified, so we introduce “Smooth” modules.
Then, modules are divided into five penalty levels:

- 0 (No penalty).
- LL (Very Low)
- L (Low)
- M (Medium)
- H (High)
- HH (Very High)

Each module is assigned a penalty level, ensuring that errors in highly predictable modules are more severely penalized than errors in those that are difficult to infer.

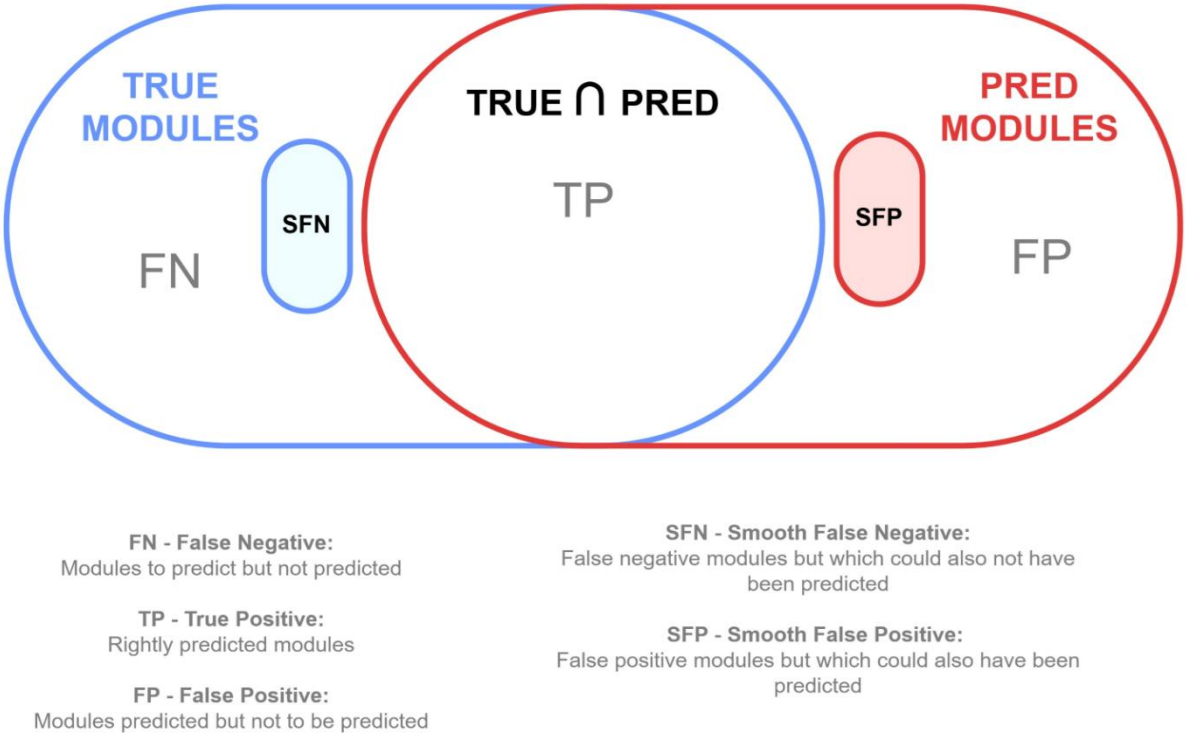

Figure 4.2 – Sets schema and explanation

Challenge N.3: Modules classification for customized penalty assignment

To assign a customized penalty for each module, it is necessary to classify them rigorously based on well-defined characteristics.

Solution adopted:
A hierarchical classification tree is introduced to correctly classify each module based on its characteristics, allowing different penalties for each category (corresponding to a leaf):

- **First division (Depth 1):** Separates modules that are difficult/impossible to infer (Smooth) from those that are inferable:
  - D (Difficult-to-infer modules).
  - N (Not-inferable modules).
  - Other (inferable modules).

- **Second division (Depth 2):** For inferable modules, categorization based on the type of main related properties:

- Simple – no properties (e.g., FinalClass).
- Boolean – most properties are boolean (e.g., UnisedImports).
- Regex – most properties are regular expressions (e.g., MethodName).
- Integer – most properties have integer values (e.g., Indentation).
- Others – properties difficult or useless to predict (e.g., AvoidStaticImport).

A leaves_for_modules.csv dataset stores the classification of each module (created manually with the official documentation as source), enabling fine-tuned penalty assignment.
Each leaf node ultimately has three different penalties based on the set in which the module falls (FN, TP, FP).

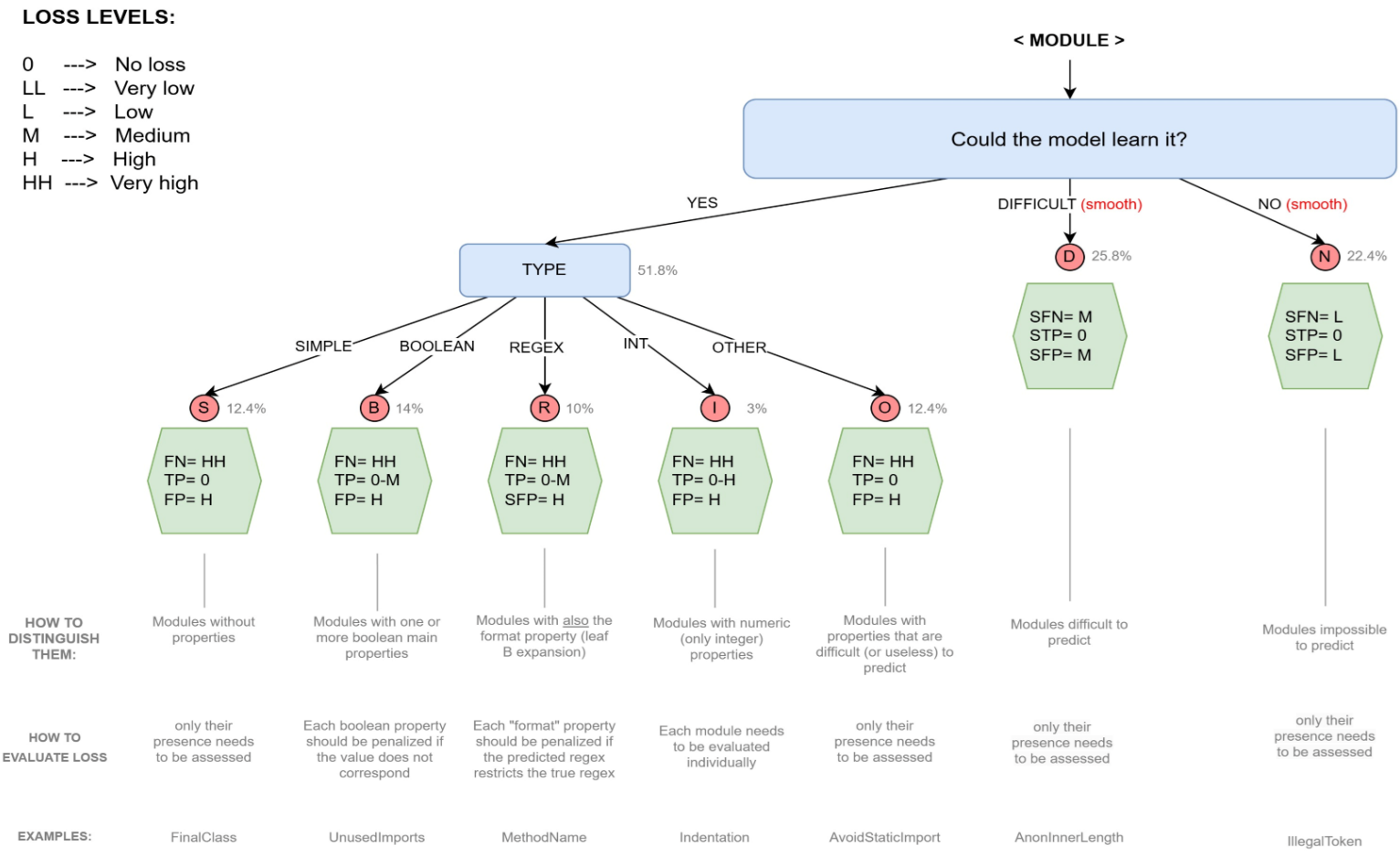

Figure 4.3 – Initial structure of the module's classification tree.

Challenge N.4: Defining specific penalty values
Once penalty levels are defined, it must be decided specific values to attribute to each one.

Solution adopted:
Three different approaches to computing loss have been tested:

- Constant scaling: Assigns the same penalty to all non-zero loss levels:
  [0 = 0, LL = 1, L = 1, M = 1, H = 1, HH = 1]
- Linear scaling: Penalizes larger deviations more severely.
  [0 = 0, LL = 1, L = 2, M = 3, H = 4, HH = 5]
- Exponential scaling: Amplifies penalties for high-severity misclassifications.
  [0 = 0, LL = 1, L = 4, M = 9, H = 16, HH = 25]

Challenge N.5: Inefficiency in module classification
Penalizing each individual module based on the type of its properties can be complex, costly, and often not sufficiently impactful in making the loss function consistent.

Solution adopted:
Leaves S, B, R, I, and O (which represent different inferable module types) are merged into a single category, so the classification tree is pruned.
The leaves_for_modules.csv dataset is simplified, ensuring that the model remains computationally efficient.

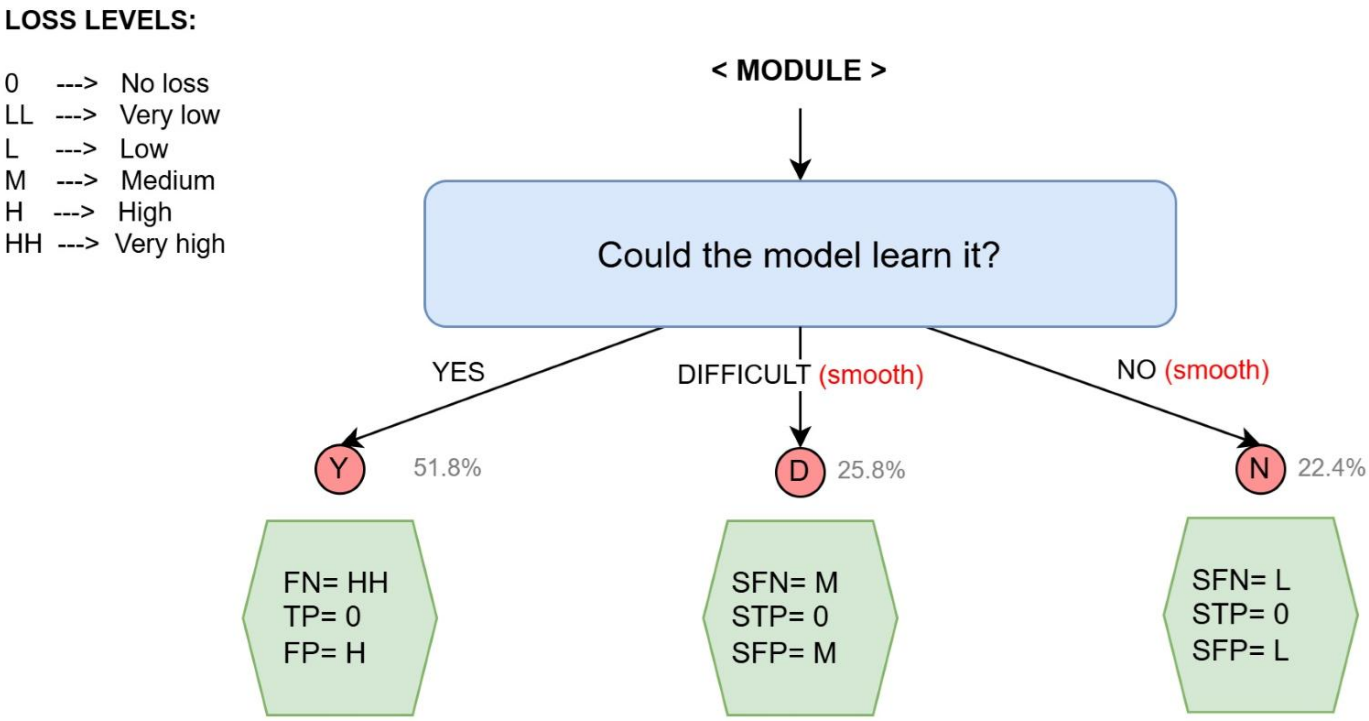

Figure 4.4 – Initial structure of the module's classification tree.

Challenge N.6: Unnormalized Loss Leading to Size-Dependent Biases
The magnitude of the calculated loss is not normalized to the size of the Checkstyle in terms of modules. A *Checkstyle_pred* with fewer modules will always have a lower loss compared to a more restrictive one.

Solution adopted:
We normalize the loss using:

$$\frac{Loss}{len(TP) + len(FP)}$$

This ensures that the loss is proportional to the size of Checkstyle_true, improving comparability.

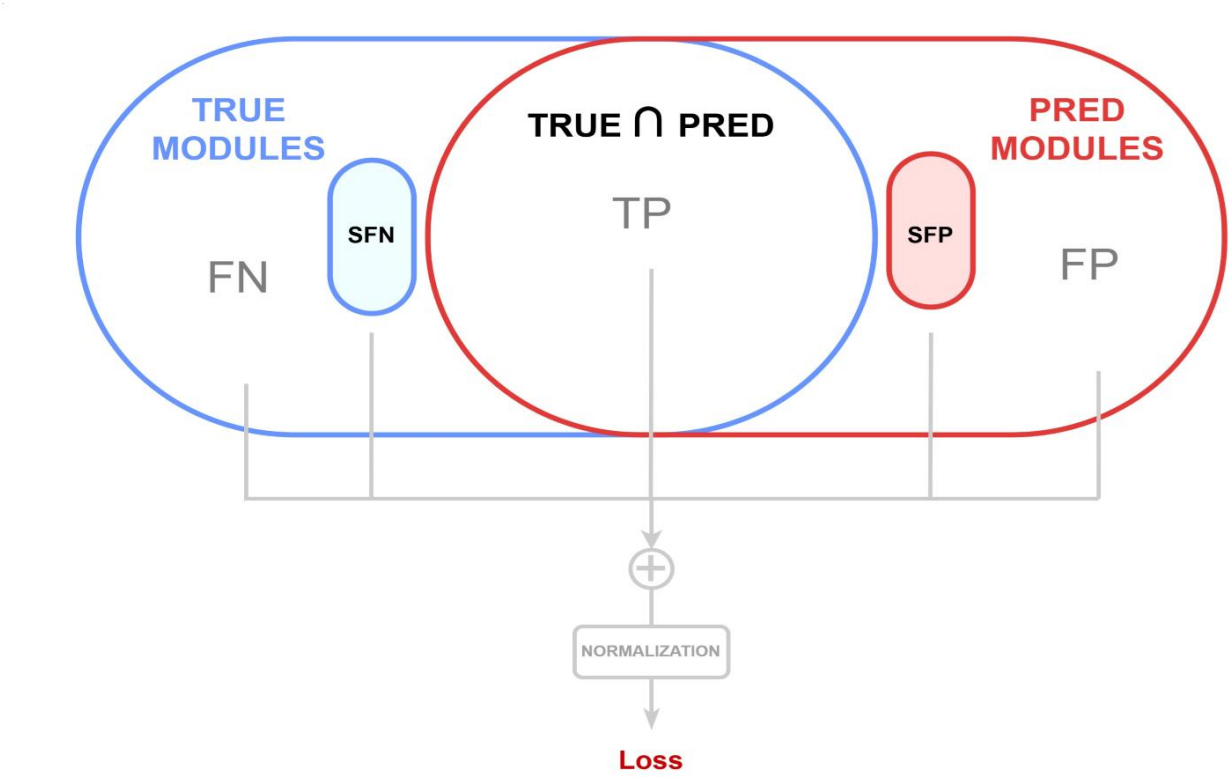

Figure 4.5 – Sets schema with final sum and normalization.

Challenge N.8: Bias due to imbalance between FP and FN
If the number of False Positives (FP) and False Negatives (FN) is highly unbalanced, the loss function may unfairly favor larger or smaller Checkstyle configurations.

Solution adopted:
A refined normalization formula is introduced, dividing the loss by the number of all modules involved in the analysis:

$$\frac{Loss}{len(TP) + len(FP) + len(FN)}$$

which ensures that loss values are size-independent and better reflect LLM performance across different Checkstyle configurations.
An in-depth analysis follows regarding the reasons for this choice.

Normalization type in-depth analysis:
Studying the two different types of normalization taking into account two Checkstyle_true of different sizes and their corresponding Checkstyle_pred:

- **len(FP) = len(FN):**

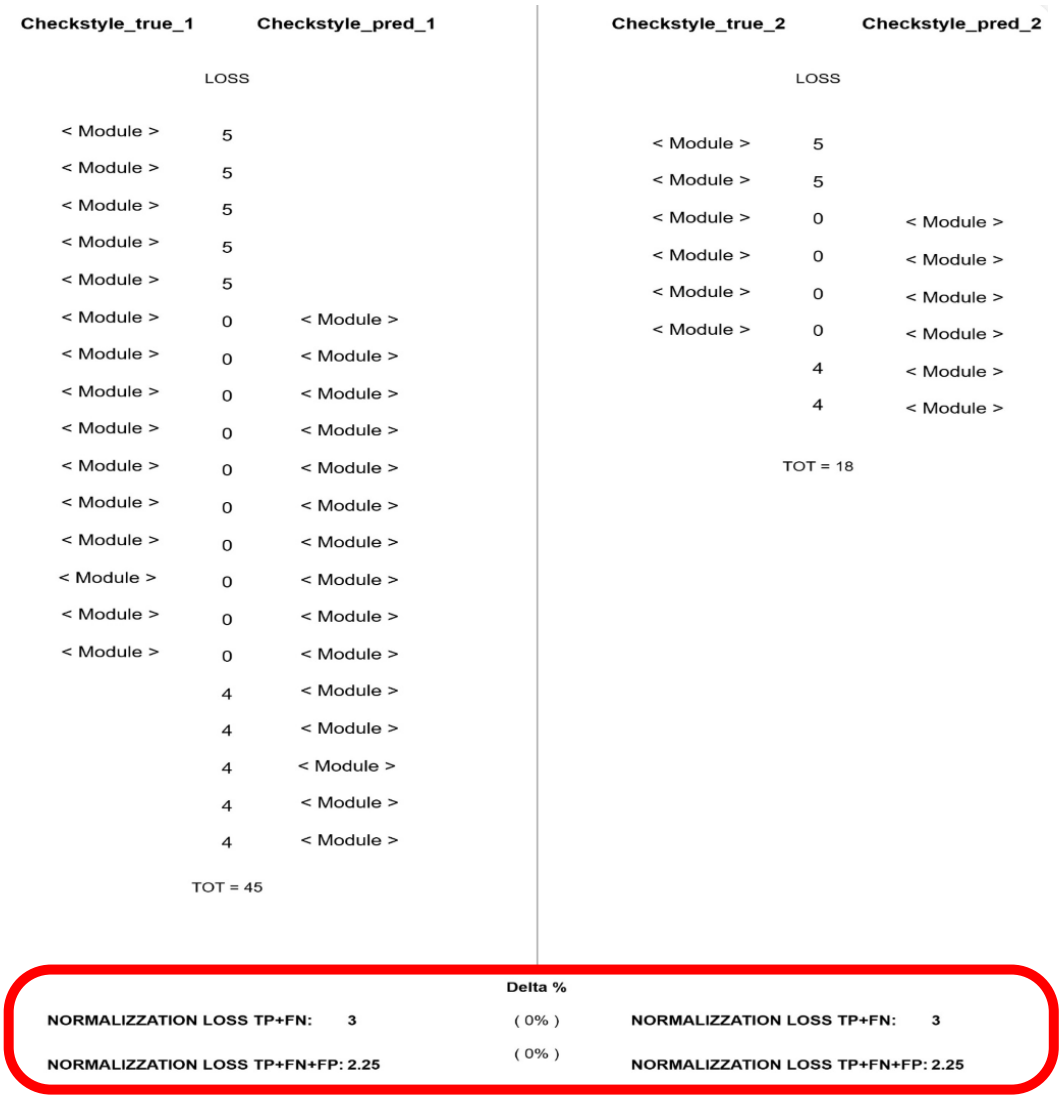

Figure 4.6 - Normalization analysis (1).

If FP and FN are equal in number, the normalizations (FN+TP) and (FN+TP+FP) are equal because the percentage difference (Delta %) between the two differently sized Checkstyles is 0.

- **len(FP) > len(FN):**

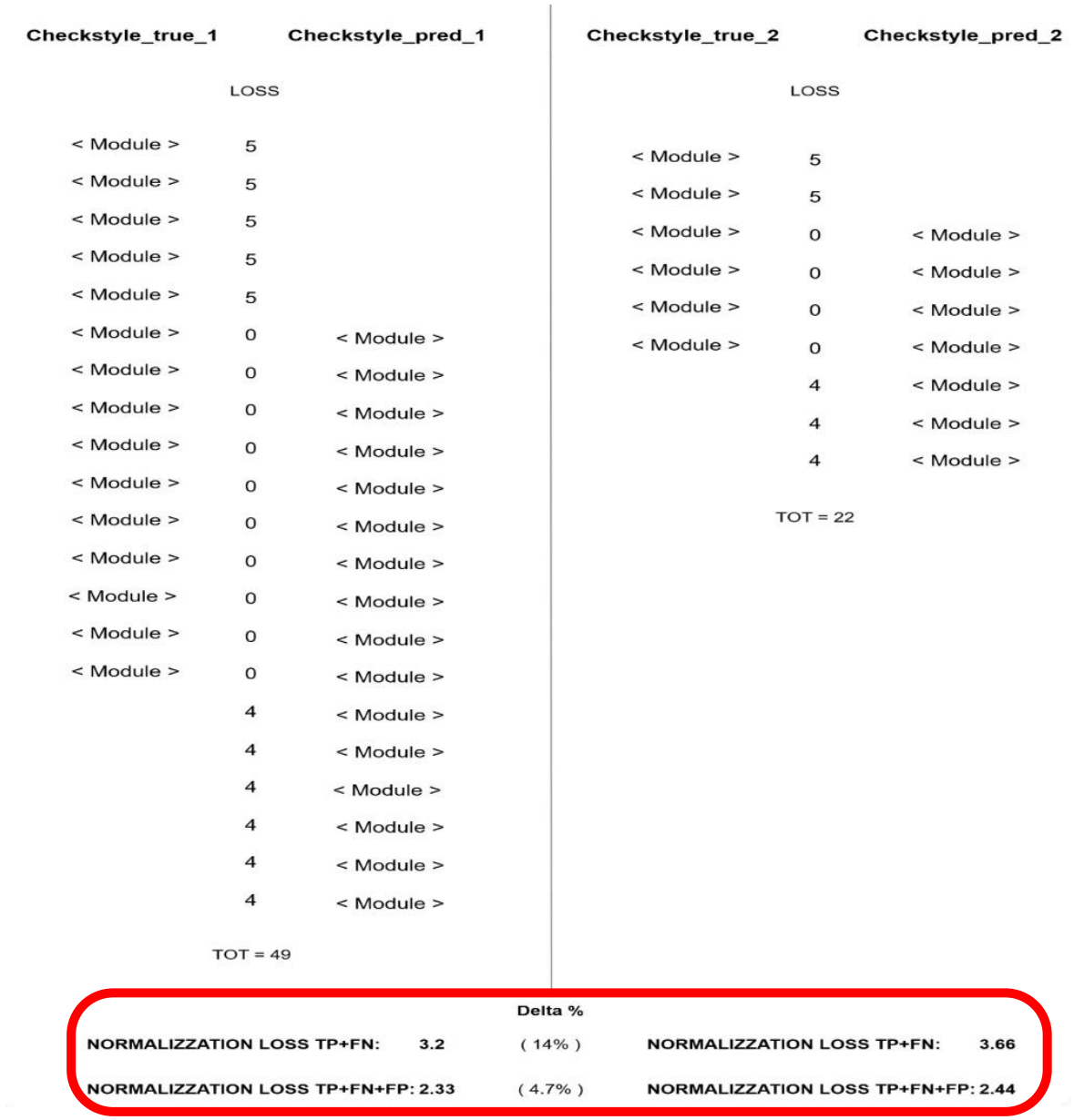

Figure 4.7 – Normalization analysis (2).

If FPs are more than FNs, normalization (TP+FN) greatly penalizes the 'smaller' Checkstyles. Losses are more comparable, however, using (FN+TP+FP): lower Delta.

- **len(FP) < len(FN):**

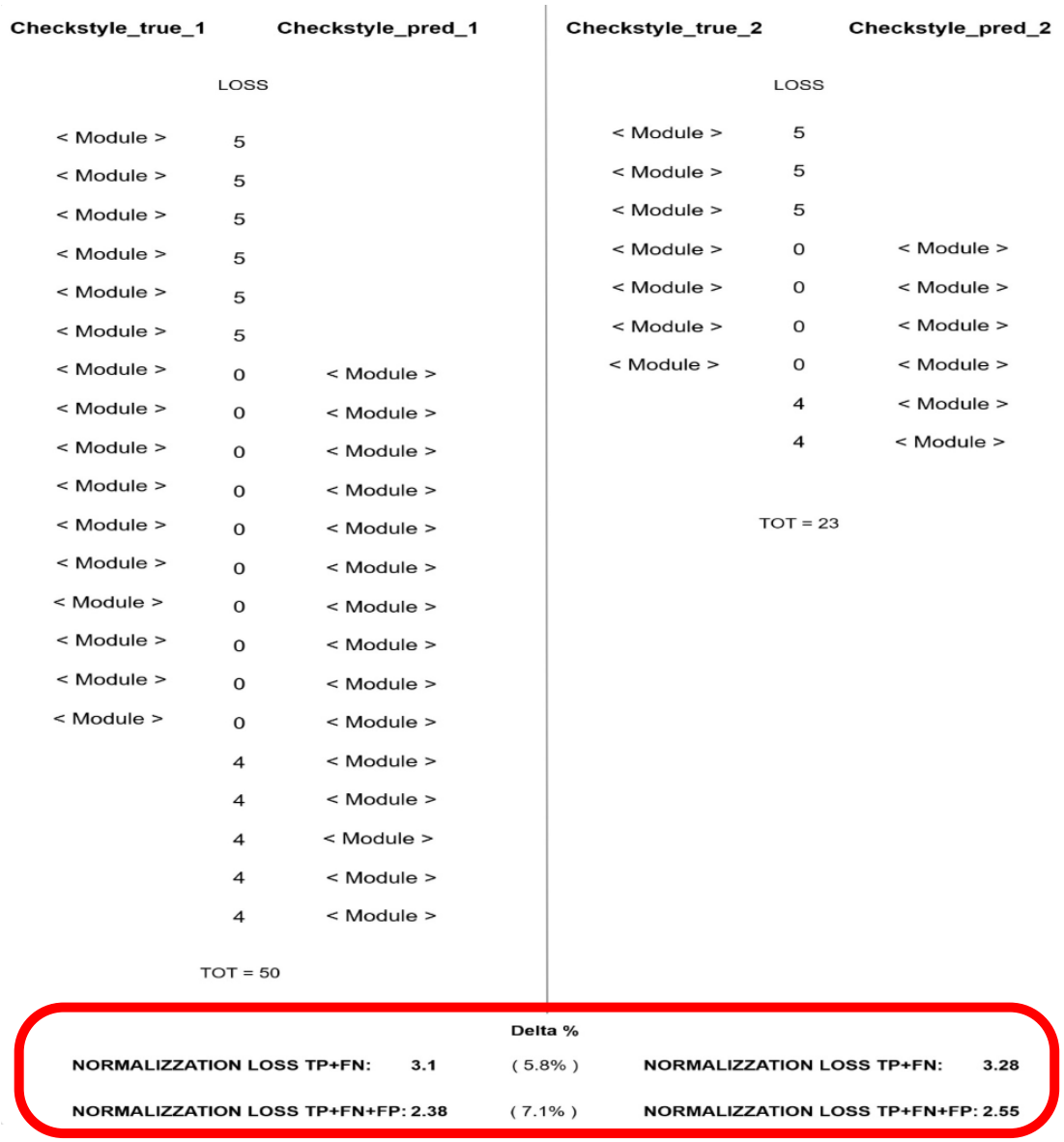

Figure 4.8 - Normalization analysis (3).

If FPs are less than FNs, the two types of normalization have a very similar Delta. As a result, the best normalization to use is (FN+TP+FP).

Final approach

The evaluation function for the Checkstyle.xml configuration generated by an LLM employs an advanced loss model to accurately assess the quality of the prediction.

This method considers not only the simple percentage of correctly identified modules but also introduces a more sophisticated evaluation based on True Positives (TP), False Positives (FP), and False Negatives (FN). Each module in the Checkstyle.xml is classified according to its inferability from the source code and penalized adaptively across five severity levels, ensuring a fairer assessment.

To enhance the robustness of the function, the loss is normalized by the size of the reference Checkstyle, preventing distortions due to the configuration's length.

4.4 – Generated Checkstyle Execution

The primary objective of this study is to assess whether a LLM can generate a Checkstyle configuration file that is both syntactically and structurally correct. If the generated Checkstyle file meets these criteria, it is then employed to identify all style violations in Java files that were originally written to conform to the Checkstyle_true configuration.

A crucial aspect of this analysis is to determine the extent to which the generated Checkstyle configuration introduces additional constraints beyond those specified in Checkstyle_true. Any additional style violations detected using the predicted Checkstyle are considered false positives (FP), reflecting stylistic constraints inferred by the LLM rather than the actual project requirements.

Initial Solution

To facilitate the evaluation process, the execution of the style check was automated using terminal commands. This involved running the style check for both the Checkstyle_true configuration (provided in the repository) and the Checkstyle_pred configuration (generated by the LLM). Automating this process allowed for a direct comparison of the errors detected by each configuration, thereby enabling an objective assessment of the accuracy and reliability of the LLM-generated Checkstyle file.

Challenge N.1: Automating Style Check Execution Based on Repository Structure

Different Java projects utilize different build systems (e.g., Maven or Gradle). Automating the style check required adapting the execution process to the repository's structure.

Solution Adopted:

The solution was to execute the Checkstyle validation differently based on the presence of a build system configuration file:

- Maven projects (identified by the presence of a `pom.xml` file) used *Maven Check* for style validation.

- Gradle projects (identified by build.gradle) used Gradle Checkstyle to perform the style check.

Challenge N.2: Version inconsistency

The Gradle version used locally for style checking was often different from the version specified in the project. This mismatch frequently resulted in execution failures, preventing the Checkstyle validation from being completed.

Solution Adopted:

To ensure compatibility across all repositories, an alternative approach was adopted:

Instead of relying on Maven or Gradle, a universal Checkstyle execution method was implemented using Python's subprocess library: a specific Checkstyle JAR file (*Checkstyle-<version>-all.jar*) was executed directly, bypassing the need for a specific build system.

Challenge N.3: Evaluating Checkstyle Execution Output

Differentiating between a successful execution with no style errors and an execution failure due to syntactic or structural issues in the generated Checkstyle file.

Solution Adopted:

The standard output and error logs were examined to determine execution success:

If the standard output was null, this indicated that Checkstyle had failed due to an invalid configuration. If Checkstyle executed successfully, two cases were considered:

- No style errors were detected, implying that the LLM-generated Checkstyle was equivalent to Checkstyle_true.
- Style errors were detected, requiring an analysis of their type and frequency.

Challenge N.4: Poorly Structured *Checkstyle Files*

Many generated Checkstyle files failed to perform style checking correctly due to structural inconsistencies. The primary issue was improper organization of modules, particularly the misplacement of *TreeWalker* child modules.

Solution Adopted:

The prompt used for generating the Checkstyle file is improved to explicitly distinguish between global modules and *TreeWalker* modules.

The *module_explanation.txt* file has been split into two parts:

- *global_module_explanation.txt*
- *TW_modules_explanation.txt*

This adjustment significantly improved the correctness of the generated Checkstyle files.

Prompt Structure

Below you will be provided with a textual corpus containing the content of various merged Java files, all belonging to the same project (for each file, its name and absolute path in the repository are specified). Based on this, generate a possible checkstyle.xml file that might have been used as a reference to ensure the Java code passes Checkstyle checks. You must try to infer all possible modules and properties based on the writing style of the provided Java code and the official Checkstyle documentation (you must adhere to its typical structure).
Here is the textual corpus to be interpreted as a merging of the Java files:
[MERGED JAVA FILES]
A checkstyle configuration file must include a "Checker" module, which contains modules responsible for global checks, and the "TreeWalker" module, which performs specific checks by walking the syntax tree of the Java file. Below are all the global modules that can be children of Checker, but not of TreeWalker:
[GLOBAL MODULES DESCRIPTION]
Below is a list of ALL the modules that can be used in the TreeWalker module of checkstyle.xml file, for each of these modules, it is necessary to understand whether its effects are actually observable in the Java code to infer whether it is useful to use it. Use only those strictly necessary and, based on their description, think if use those that you think might have been used for writing the provided Java code:
[TREE WALKER MODULE DESCRIPTION]
When generating the response, it's crucial to avoid inserting global modules among the children of TreeWalker. I only want the content of the checkstyle.xml file without any explanation or justification for your choices.

Figure 4.9 – Final prompt structure.

Challenge N.5: Error count not normalized

The number of style errors detected increased as the number of Java classes in a repository grew. This made the absolute number of errors an unreliable indicator of Checkstyle quality.

Solution Adopted:

The number of detected errors was normalized against the total number of Java classes in the repository.

Instead of using the raw count of style violations, a ratio of errors per Java class was calculated; this normalization allowed for a more meaningful comparison of the strictness of Checkstyle_pred relative to Checkstyle_true.

Challenge N.6: Ensuring generalization

The evaluation process needed to test the generalization ability of Checkstyle_pred by running it on files that the LLM had not encountered during its training phase.

Solution Adopted:

A data-splitting strategy was employed:

- 70% of the Java classes were included in the input prompt to the LLM.
- The remaining 30% were used for validation by running *Checkstyle_pred* on them.

This approach ensured that the generated Checkstyle not only fits the visible examples but could also be generalized to non-visible Java files.

Final Approach

The Checkstyle_run function evaluates whether an advanced language model can create a correct Checkstyle configuration file for Java projects. It automates code style checks by comparing the model-generated configuration with the project's original one. To handle different building systems like Maven and Gradle, the function identifies the project type and adjusts the verification process accordingly. Additionally, it directly executes a Checkstyle JAR file to avoid compatibility issues between versions. The function analyzes the output to distinguish between successful and failed executions, normalizing the number of errors relative to the total Java classes in the project. Finally, it employs a data-splitting strategy, using 70% of the classes as input for the model and the remaining 30% to test the generated configuration on new files, ensuring its general applicability.

SECTION 5 – Results Analysis

In the **output_data** folder, all the results of our study have been uploaded: for each analyzed repository (containing a valid and well-structured Checkstyle to be used as ground truth), the corresponding Checkstyle_pred and a dataframe containing the following information have been uploaded:

- **Last_commit_hash:** Hash code of the repository's last commit to ensure the reproducibility of the experiment.
- **Constant_loss**
- **Linear_loss**
- **Exponential_loss**
- **Run_errors_Pred:** Style errors obtained using Checkstyle_pred on 30% of the repository's Java code that the LLM had not "seen".
- **Num_run_errors_Pred:** Number of style errors obtained using Checkstyle_pred.
- **Normalized_num_run_errors_Pred:** Average number of style errors per Java file (running Checkstyle_pred).

This data is subsequently analyzed to provide an objective response to the initial research question.

The two evaluation functions, Checkstyle_run and Checkstyle_loss, were able to effectively assess the capability of LLMs in this field, but the magnitude of the loss computed in Checkstyle_loss does not have an absolute meaning; it is simply a numerical value that serves as a relative indicator: a lower value indicates a better Checkstyle_pred, but its value itself has no concrete interpretation outside this context. It can be used to compare two generated Checkstyles or during specific LLM tuning for this task.

Below, we analyze the results of the Checkstyle_run function in three phases:

First evaluation step

The first evaluation step assessed whether the LLM could generate syntactically valid Checkstyle configuration files. The generated configurations were tested using

Checkstyle itself to determine if they complied with the expected XML structure and rule definitions.

Outcome	# Repositories	Percentage
Successful	536	**84.28%**
Failed	100	15.72%
Total	636	100%

Figure 5.1 - Results of generating Checkstyle configuration files on the 636 repositories in our initial dataset.

The results indicate that in 84.28% of cases, the LLM successfully produced a syntactically correct Checkstyle configuration. The remaining failures were primarily due to minor structural errors in the XML output, which could be mitigated through additional post-processing or iterative refinements.

Second evaluation step

The second evaluation aimed to measure how well the LLM captured and replicated the existing style rules of each project. The generated configurations were applied to Java files reserved for testing (30% of the repository's source code) to determine whether the LLM's rules aligned with the project's established style.

Style violation	# Projects	Percentage
No one detected	283	**52.9%**
At least one detected	253	47.1%
Total	**536**	**100%**

Figure 5.2 - Evaluation of the ability to generate Checkstyle configurations aligned with project style.

In 52.9% of cases, the generated configuration perfectly matched the project's existing style, meaning no additional violations were reported. However, in 47.1% of cases, the generated configuration was stricter than the original, flagging additional style violations. This suggests that the LLM inferred more rigid coding conventions than those explicitly defined in the project's Checkstyle file.

Third evaluation step

In the cases where additional violations were detected, an analysis was conducted to determine which Checkstyle modules were most frequently introduced by the LLM. The most common added rules are listed below:

Module	Description	# Occurrences	Percentage
WhitespaceAround	Ensures proper spacing around operators and punctuation.	183	72.3%
Indentation	Enforces consistent indentation levels.	165	65.2%
FinalParameters	Requires method parameters to be `final` to prevent modification.	136	53.8%
WhitespaceAfter	Ensures a space follows specific tokens (e.g., commas, semicolons).	119	47.0%
VisibilityModifier	Requires explicit visibility modifiers for class members.	116	45.8%
NewlineAtEndOfFile	Ensures a file ends with a single newline character.	111	43.9%
NeedBraces	Enforces the use of braces in control structures.	109	43.1%
ConstantName	Requires constants to follow uppercase snake_case naming.	102	40.3%

Figure 5.3 - Most frequent style violations detected in the generated Checkstyle configurations (only rules appearing in more than 40% of projects are reported).

In our analysis of the Checkstyle modules that the model most frequently infers incorrectly, we observed that most of these modules are not particularly problematic to infer. Specifically, many of the modules that cause false positives—such as WhitespaceAround, Indentation, WhitespaceAfter, and NewlineAtEndOfFile—are primarily related to whitespace management. These stylistic conventions are commonly adhered to by developers, making them patterns that the model naturally learns. However, they may not always be strictly enforced across an entire project, leading the model to impose stricter constraints than those originally defined.

Furthermore, the presence of modules like FinalParameters and VisibilityModifier suggests that the model tends to infer more rigid constraints, even when the original configuration does not explicitly enforce them. This behavior does not significantly impact the usability of the generated Checkstyle, as its primary objective is to promote internal consistency rather than simply replicate pre-existing configurations.

Overall, while the model does introduce some stricter style constraints that result in false positives, these errors are mostly associated with minor and commonly accepted stylistic conventions. Addressing these issues could be achieved through few-shot learning, fine-tuning techniques, or integrating automatic formatting tools that help refine the inferred rules.

CONCLUSIONS

The evaluation of Large Language Models for generating Checkstyle configuration files has demonstrated promising results. Our analysis confirms that LLMs can often produce Checkstyle configurations that are well-structured and coherent with the coding styles observed in Java projects. More importantly, in cases where errors do occur, they are usually minor and can be efficiently resolved through straightforward post-processing steps.

A key strength of LLM-generated Checkstyle configurations lies in their ability to infer implicit coding patterns from the provided Java source code. The configurations produced are generally aligned with project conventions, reducing the need for extensive manual adjustments. Even when misconfigurations or omissions are present, they rarely lead to major issues, as minor refinements can quickly rectify them.

These findings suggest that integrating LLMs into software development workflows for Checkstyle configuration generation is a viable approach: the ability to automate this process significantly reduces the manual effort required.

Future Developments

Future work will focus on refining the evaluation framework and enhancing the capabilities of LLMs in generating more precise Checkstyle configurations. The following directions outline potential areas of improvement:

- **Granular Loss Calculation**: Future development should implement a more detailed loss function that penalizes different types of module properties separately. For instance, boolean properties should be treated differently from integer-based properties to provide a more accurate evaluation of generated configurations.
- **Refining Module Division**: The classification of modules into leaves should be optimized based on how easily an LLM can infer them. By improving the categorization of modules (as stored in *leaves_for_modules.csv*), we can enhance the precision of Checkstyle generation.
- **Evaluation of Alternative LLMs**: Additional models should be tested and compared in terms of performance, including aspects such as loss, latency, and

context length. This comparison will provide insights into selecting the best LLM for this specific task.

- **Advanced Prompt Engineering**: Enhancing the LLM's input prompt using advanced techniques like "pay or die," "chain-of-thought," and other structured prompting methodologies could lead to better inference accuracy and more structured output.
- **Exploring Different Loss Configurations**: Alternative numerical strategies should be considered for evaluating loss at the five levels, such as logarithmic or factorial scoring methods. These refinements could lead to a more nuanced assessment of the quality of generated configurations.

By addressing these aspects, we can improve the overall effectiveness of LLMs in generating Checkstyle configurations and contribute to the broader field of automated code style enforcement.

BIBLIOGRAPHY

[1] S. P. Reiss, «Automatic code stylizing», in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, Atlanta Georgia USA: ACM, nov. 2007, pp. 74–83. doi: 10.1145/1321631.1321645.

[2] F. Zampetti, S. Mudbhari, V. Arnaoudova, M. Di Penta, S. Panichella, e G. Antoniol, «Using code reviews to automatically configure static analysis tools», *Empir Software Eng*, vol. 27, fasc. 1, p. 28, gen. 2022, doi: 10.1007/s10664-021-10076-4.

[3] V. Markovtsev, W. Long, H. Mougard, K. Slavnov, e E. Bulychev, «STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms», 2019, *arXiv*. doi: 10.48550/ARXIV.1904.00935.

[4] G. Fan, X. Xie, X. Zheng, Y. Liang, e P. Di, «Static Code Analysis in the AI Era: An In-depth Exploration of the Concept, Function, and Potential of Intelligent Code Analysis Agents», 2023, *arXiv*. doi: 10.48550/ARXIV.2310.08837.