GreenCodeAnalyzer: Detecting Energy Code Smells in Data Science with Static Analysis

CS4575 Sustainable Software Engineering Project 2

Group 1

Marina Escribano Esteban
Kevin Hoxha
Inaesh Joshi
Todor Mladenović

Data Science and Artificial Intelligence Technology
Delft University of Technology
The Netherlands
4th April 2025

1 Introduction

With the increasing reliance on Python for data science applications, ranging from machine learning and artificial intelligence to large-scale data processing, energy efficiency in code execution has become a critical concern. The International Energy Agency (IEA) estimates that data centers, heavily utilized by AI and data science, consumed 460 terawatt-hours globally in 2022, a figure projected to rise sharply by 2030 [1]. As data-driven workflows grow in complexity and scale, inefficient code can lead to excessive energy consumption, impacting both computational costs and environmental sustainability [2].

Despite this growing concern, the energy consumption of Python programs remains largely overlooked, particularly in the context of data-intensive computations. Developers tend to prioritize performance and functionality, often at the expense of energy efficiency [3]. Data science workflows typically involve repeated computations, large datasets, and complex transformations, all of which increase the likelihood of inefficient code patterns, such as unnecessary dataframe joins, redundant model training, or excessive looping. These inefficiencies not only raise computational costs but also contribute to the environmental footprint of data science practices, which are increasingly under pressure to adopt sustainable methodologies [4].

Currently, most tools for measuring energy consumption in Python code rely on dynamic analysis, requiring program execution. Tools such as pyJoules [5], CodeCarbon [6], and PyPen [7] offer real-time profiling capabilities but are not suitable for early-stage development when execution is either expensive or infeasible. In contrast, static analysis tools like Pylint[1] and MyPy[2] check for code quality and type safety, but they lack the ability to detect energy-related issues. This reveals a clear gap: there is currently no static analysis tool focused on identifying energy-related code smells in Python, particularly within data science contexts.

To address this issue, we propose GreenCodeAnalyzer, a static analysis tool designed to detect energy-intensive data science coding patterns in Python and provide actionable recommendations to developers. Implemented as a Visual Studio Code plug-in, GreenCodeAnalyzer integrates energy-aware analysis directly into the development workflow, without requiring code execution, allowing developers to receive early feedback on energy inefficiencies and write more sustainable code from the outset. The tool leverages Python's Abstract Syntax Trees (ASTs) to statically traverse and analyze source code, which enables efficient rule matching and code smell detection tailored to common data science practices. It currently supports 20 static analysis rules derived from literature and practical observations, targeting common inefficiencies. This proactive approach not only helps avoid costly rework later in the development cycle but also aligns software engineering practices with broader environmental goals [8, 9].

The tool currently supports rules for four of the most common data science libraries in Python: Pandas, PyTorch, TensorFlow, and SciKit-Learn. We tested the tool on 20 code files, where it identified a total of 100 instances of energy-related code smells, of which 78 were relevant and 22 were false positives, resulting in a precision of 78%. By delivering practical, domain-specific feedback, our tool empowers data scientists to adopt a more energy-conscious approach to writing code, ultimately contributing to more sustainable data-driven innovation.

This report details our process of implementing and evaluating GreenCodeAnalyzer. Section 2 provides the theoretical background and static analysis principles, and Section 3 reviews related work. Section 4 presents our proposed approach, followed by Section 5, which describes its implementation as a Visual Studio Code plug-in. Section 6 outlines our experimental setup, with results and discussion covered in Sections 7 and 8. Section 9 discusses potential threats to validity, Section 10 explores future directions, Section 11 describes our dissemination efforts, and Section 12 summarizes our key findings. Through this study, we show how GreenCodeAnalyzer contributes to sustainable software engineering and addresses a key gap in making data science more energy-efficient.

2 Background

This section introduces foundational concepts motivating the static analysis tool proposed in Section 4. It emphasizes static code analysis as a proactive approach to identify energy code smells.

2.1 Energy-Aware Software Engineering and Data Science

Concerns over the rising energy consumption and environmental impact of technology have driven increased interest in Green ICT (Information and Communication Technology) [2, 10]. Green ICT encompasses the energy efficiency aspects throughout the entire lifecycle and context of ICT systems. Within this broad field, energy-aware software development

[1] Pylint's website: `https://pypi.org/project/pylint/`
[2] MyPy's website: `https://mypy-lang.org/`

is a critical component.

Energy-aware software engineering specifically focuses on employing tools and methods that prioritize energy consumption as a primary software design goal [11]. Currently, few programmers have a clear understanding of the energy usage of their software or which parts of their programs consume the most power. As a result, energy considerations are often overlooked during initial development stages and only addressed after deployment. If energy targets are not met post-deployment, it can result in lengthy and costly redevelopment cycles.

One critical aspect within energy-aware software engineering is energy-aware data science, which specifically addresses the energy impacts of data-intensive tasks, such as machine learning model training, big data processing, and complex analytical workflows. Due to their computational intensity, data science and AI applications have considerable energy demands. As their use continues to expand, the overall energy consumption associated with these technologies is expected to significantly increase. Currently, data science and AI workloads account for approximately 10% of data center electricity demand, and this share is projected to rise to 20% by 2030 [12].

While energy consumption fundamentally occurs through physical processes within hardware, software significantly influences how efficiently hardware is utilized. Inefficient software can lead to energy waste, which cannot be addressed solely through hardware advancements. Thus, enhancing software energy efficiency is crucial for reducing overall energy usage.

A key requirement for achieving this is energy transparency - the capability for software developers to clearly understand or visualize their program's energy consumption without explicitly executing and measuring it. According to Eder et al. [11], two primary foundations underpin energy transparency: energy modeling and static energy analysis. Energy modeling involves constructing abstract models that associate hardware-level energy consumption with specific software elements. Static energy analysis uses these energy models to estimate the energy consumption of software without actual execution.

The approach outlined in Section 4 describes a simplified form of energy analysis aimed at integrating energy considerations early in the development cycle, primarily through identifying "energy code smells" (see Section 2.2).

2.2 Energy Code Smells

Morisio et al. [13] define energy code smells as patterns in source code that can increase software's impact on power consumption. These are suboptimal implementation choices made at the source code level, leading to inefficient utilization of hardware resources and consequently higher energy or power consumption. Given the multiple abstraction levels and organizational structures within software, smells can manifest at the code, design, or architectural level. Typical examples include needlessly draining CPU cycles, unnecessarily keeping hardware active, or preventing hardware from entering energy-saving states.

This concept extends the traditional notion of code smells which refers to software design patterns that negatively impact maintainability and code quality [14]. Through refactoring and optimization, code smells can be addressed. Energy code smells adapt this concept specifically towards energy efficiency and sustainability in software development. Just as eliminating traditional code smells enhances software maintainability and performance, refactoring energy code smells can improve software energy efficiency.

Identifying and optimizing even minor inefficiencies in source code can collectively result in significant energy savings and positive sustainability impacts [9]. However, developers may not commonly recognize these energy code smells, as they are not widely documented or well-known. Consequently, increasing awareness of energy code smells holds important implications for sustainable software engineering.

Energy code smells can exist in diverse software contexts, and some research has aimed at identifying and mitigating common energy inefficiencies. For instance, in the context of general software development, Gurung et al. identified energy code smells related to inefficient loop constructs in Java [15], such as unnecessary iterations and flawed loop conditions. Morisio et al. [13] specifically explored and validated energy code smells within embedded systems, defining smells such as dead local stores, which describe situations where a local variable is assigned a value that is never subsequently read or utilized. Despite the rising popularity and high computational demands of data science - particularly given its current significance - little research has focused on energy code smells within the data science domain.

2.3 Static Analysis

Static analysis involves examining source code without executing it. By parsing and inspecting code for specific patterns or rules, static analysis tools can identify potential issues early in software development. Automated Static Analysis Tools (ASATs) scan the source or binary code of a software system, searching for predefined problems [16]. By addressing the warnings reported by ASATs, development teams can resolve issues early in the software lifecycle.

Since their introduction in the 1970s, static analysis tools have become widely adopted by companies due to the growing complexity of software [17]. These tools evaluate a wide range of program properties, from simple coding style guidelines to advanced software bugs and complex security vulnerabilities.

ASATs typically use techniques such as data-flow analysis and control-flow analysis to detect defects in source code [16]. However, ASAT techniques often struggle to scale effectively. To manage complexity, abstractions are applied, but these abstractions - along with the inherent undecidability of accurately checking certain program properties - result in high false positive and false negative rates. Additionally, warnings from ASATs can sometimes be difficult to understand, making it important to carefully select which issues to highlight and how to communicate them clearly to developers, as evaluating warnings can be time-consuming.

Despite these challenges, research indicates that 71% of developers do consider warnings from ASATs [18], particularly when contextually relevant, thus confirming the potential value of ASATs in enhancing code quality.

3 Related Work

A variety of tools have been developed to assess and improve the energy efficiency of software. These tools can be broadly categorized into dynamic analysis tools, which measure energy consumption during runtime, and static analysis tools, which identify energy inefficiencies by analyzing code without execution. Below, we explore these categories separately.

3.1 Dynamic Energy Analysis Tools

Dynamic analysis tools monitor and profile energy consumption during software execution, offering insights into real-time energy usage.

CodeCarbon [6] is a lightweight Python package that estimates the carbon dioxide emissions generated by cloud or personal computing resources during code execution. It calculates emissions based on power consumption and location-dependent carbon intensity, helping developers reduce emissions by optimizing code or hosting their cloud infrastructure in regions that use renewable energy sources.

PyJoules [5] is a toolkit focused on measuring the energy footprint of a host machine while executing Python code. It tracks consumption from hardware components like Intel CPU socket packages, RAM, or GPUs using interfaces such as Intel's Running Average Power Limit (RAPL).

Similarly, **EnergiBridge** [19] is a cross-platform measurement utility that leverages hardware interfaces like RAPL to measure consumption during runtime.

PyPen [7] is a cross-platform profiler that collects detailed execution data through instrumentation (i.e. inserting monitoring instructions). It identifies energy hotspots, which are sections of the software where energy consumption is highest.

ALEA (Adaptive Lightweight Energy Analyzer) [20] provides detailed energy profiling through probabilistic analysis, offering fine-grained information at the basic block level with minimal performance overhead.

MANAi (Method ANalysis for AI) [21] is an IntelliJ IDEA plug-in that profiles and visualizes the energy consumption of Java unit test methods, delivering real-time visual feedback.

The dependence of dynamic tools like these on code execution can make them less practical for early-stage development or large-scale applications where running the code is resource-intensive. Our approach, GreenCodeAnalyzer, focuses on early-stage development through static analysis.

3.2 Static Analysis for Energy Efficiency

Static analysis tools examine source code to detect energy code smells without requiring execution. This approach allows developers to identify potential issues early in the development process.

Green Code Initiative [22] is a collaborative effort aimed at reducing the environmental impact of software by promoting energy efficiency. It develops static analysis tools for multiple programming languages (e.g. Java, C#) to identify code structures that lead to excessive energy and resource use.

GreenCode [23] is a VSCode plug-in that provides real-time insights into code, highlighting optimization opportunities in SQL queries and other sections with color-coded severity indicators.

Ec0lint [24] is an open-source tool designed for frontend developers to minimize the carbon footprint of websites, claiming significant reductions in CO2 emissions through static analysis.

GreenForLoops [15] is a Java Maven custom SonarQube plug-in that targets energy code smells in Java loops, such as unnecessary iterations or inefficient looping conditions.

EnergyAnalyzer [25] focuses on embedded software, constructing a control-flow graph to analyze execution paths and estimate worst-case energy consumption, informed by hardware event counters (e.g., cache misses, executed instructions).

SAAD (Static Application Analysis Detector) [26] targets Android applications, detecting resource leaks and layout defects by analyzing component call graphs and resource management. It integrates with Android's Lint tool to filter out energy-related defects.

In the context of data science, **GreenPyData** [27] is an open-source SonarQube plug-in for PyTorch, identifying six code smells, such as unnecessary bias in convolution layers before batch normalization.

However, static analysis for energy code smells in data science remains largely unexplored. While GreenPyData is an exception, it is limited to the PyTorch library, and most existing tools focus on general software development or other fields. Additionally, most of these tools target languages like Java or C#, despite Python being the most popular programming language [28]. Our approach fills this gap by focusing specifically on Python and data science, covering libraries such as Scikit-learn, NumPy, Pandas, and TensorFlow, in addition to PyTorch.

4 Solution Proposal

We approach the identified gap in the literature by building a tool with a user-centered focus. This considers both the developer perspective for anyone wanting to expland our project any further, and the end-user wanting to employ our tool to obtain more energy-aware Python code. With this in mind, we build a tool that seamlessly integrates into a VSCode plug-in for ease of use. GreenCodeAnalyzer can be defined as a static analysis tool, implemented in Python, for reducing unnecessary energy consumption in Python scripts within the domain of DS and AI.

The tool identifies code smells in Python scripts, and offers a 'green optimization' for each type of smell. The code smells and their optimizations are determined through a literature study on common patterns and library calls that have equivalent yet less energy consuming alternatives. The rules are carefully chosen, each tailored to one or more data-science-specific libraries in Python. Specifically, we define 20 rules as we believe this number strikes the balance between generalizability throughout AI-specific code and the quality each rule is managed to be crafted to.

When using our plug-in and being offered optimizations, end-users can choose to apply these to effectively reduce the energy consumption of their scripts at run time.

5 Implementation

This section outlines implementation details to detecting energy-related code smells. First, we describe how Abstract Syntax Trees (ASTs) enable structured static code analysis. Then, we detail how we identify energy code smells and discuss an ASAT in the formal of a Visual Studio Code (VSCode) plug-in.

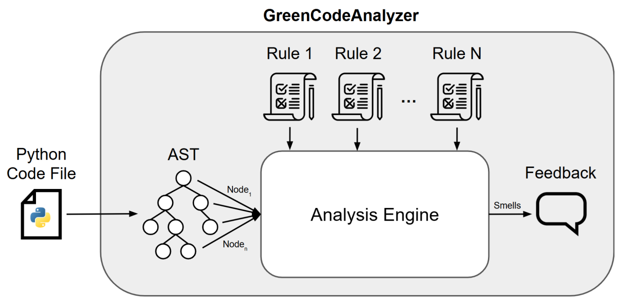


Figure 1: Conceptual view of GreenCodeAnalyzer. The system takes source code as input, parses it into an Abstract Syntax Tree (AST), and applies multiple modular rules per node to detect energy code smells. The identified smells are then provided as visual feedback.

5.1 Static Code Analysis Using Abstract Syntax Trees

We implement a static analysis tool designed to detect energy code smells in of Python software in the context of data science. A conceptual view of this tool is illustrated in Figure 1. As depicted, the static analysis process begins by taking a Python code file as input. The input code is parsed into an Abstract Syntax Tree (AST) using Python's built-in *ast* library[3]. An AST is a structured, hierarchical representation of code, where each programming construct - such as functions, loops, and conditional statements - is represented as a distinct node within the tree. An example of an AST is depicted in Figure 2.

After constructing the AST, we traverse each node systematically. During this traversal, the analysis engine applies a predefined set of energy-related rules to each node to identify potential energy code smells. The detailed methodology defining these rules and the specific criteria for identifying energy code smells are discussed in Section 5.2.

Once all relevant nodes have been analyzed, the analysis engine aggregates the identified code smells, annotating the corresponding line numbers. These

[3]AST documentation: `https://docs.python.org/3/library/ast.html`

results are then passed to the plug-in frontend, which visually presents the findings to users, facilitating immediate feedback and actionable insights. Further details on the plug-in's functionalities and user feedback mechanisms are provided in Section 5.3.

The internal implementation details of the analysis engine and code in general are intentionally omitted here for brevity. Developers interested in contributing or reviewing the specifics of the analysis engine can find the complete implementation on the GitHub repository [4].

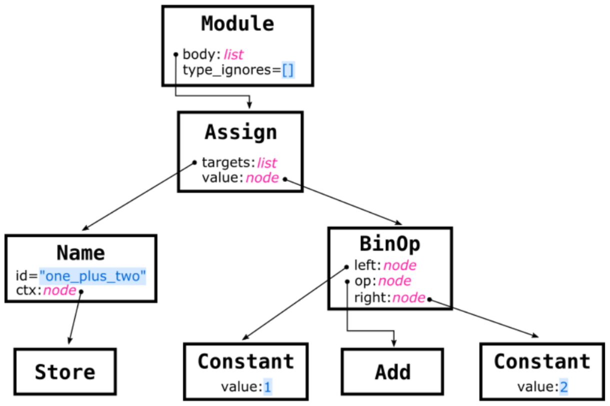


Figure 2: An example of an Abstract Syntax Tree (AST) [29], which represents the syntactic structure of code as a hierarchical tree. Each node corresponds to a language construct, with branches capturing relationships between operations, variables, and values.

5.2 Code Smell Identification

Energy code smells are identified using modular rules, as illustrated in Figure 1. This design approach maintains the open-source nature of the project and allows developers to easily extend the ruleset with additional energy code smells that have not yet been considered.

In our approach, we identify energy-related inefficiencies (code smells) specifically within Python code used for data science applications. Identification is based on previous literature, existing repositories that explore related concepts, and logical analysis. The developed static analysis tool systematically detects and highlights these inefficiencies. The current ruleset specifically focuses on five widely used data science libraries: PyTorch, NumPy, Pandas, TensorFlow, and Sci-Kit Learn. However, the rules are designed with extensibility in mind, allowing developers to easily expand the coverage to additional libraries.

[4]GreenCodeAnalyzer GitHub: https://github.com/ianjoshi/green-code-analyzer

In total, we define 20 energy code smells, each with its own corresponding detection rule. An overview of these identified smells, along with the description for each, is provided in Table 1.

The rules operate on individual nodes within the Abstract Syntax Tree (AST). For example, a rule might specifically target a For Loop, a variable assignment, or an entire module. Certain rules require the systematic collection of metadata, such as variable chains or counters, during analysis. This metadata is stored internally within each rule since only one instance of each rule exists during the execution of the GreenCodeAnalyzer plug-in.

The exact methodology for detecting smells varies by rule. By isolating implementation details within each rule, modularity is significantly enhanced. Some rules utilize regular expression patterns, whereas others might employ variable tracking through assignment chains.

5.3 Green Code Analyzer: A Visual Studio Code Plug-In

To enable seamless interaction with the project's logic and provide real-time feedback, we develop a Visual Studio Code extension: **GreenCodeAnalyzer**. This extension allows developers to detect and address energy inefficiencies in Python code directly within their IDE, promoting energy-aware development "left" (earlier) in the coding process. The architecture of GreenCodeAnalyzer lies at both ends of the project pipeline: initially taking in the user's Python file and later highlighting inefficiencies detected using visual cues directly within the code editor, offering users immediate feedback.

The workflow begins when a user runs the extension to analyze the Python file they are actively working on. The extension isolates this file and runs the analyzer tool on it, which executes its logic by first extracting the Abstract Syntax Tree (AST) and comparing it against a predefined set of rules (Table 1). When the analysis is complete, a structured output with details of all detected inefficiencies is returned to the plug-in.

The plug-in then parses this output, extracting key details such as the lines of code that contain inefficiencies, the specific rule each line violated, and the recommended optimization. It then visually represents these findings within the editor by marking problematic lines with color-coded gutter icons, providing a clear indication of energy inefficiencies in the code. When the user hovers anywhere over an affected line, a hover tooltip appears, providing more detailed information on the smell. If multiple rules were matched to

Energy Code Smells	**Description**
Batch Matrix Multiplication	When performing multiple matrix multiplications on a batch of matrices, use optimized batch operations rather than separate operations in loops.
Blocking Data Loader [27]	Prevent using data loading strategies that stall GPU execution (e.g., single-process or sequential data loading).
Broadcasting [30, 31]	Normally, when you want to perform operations like addition and multiplication, you need to ensure that the operands' shapes match. Tiling can be used to match shapes but stores intermediate results.
Calculating Gradients [32, 33]	When performing inference (i.e., forward pass without training or backpropagation), some frameworks by default track operations for autograd, while others track only if specified.
Chain Indexing [30]	Chain indexing refers to using `df["one"]["two"]`, which can be interpreted as two separate calls, causing performance overhead.
Conditional Operations [30]	When performing a conditional operator on an array or dataframe inside loops, consider vectorized methods (e.g., `np.where()`).
Data Parallelization [27]	Refrain from wrapping models in `torch.nn.DataParallel` when `torch.nn.parallel.DistributedDataParallel (DDP)` is superior, even on a single node.
Element-Wise Operations [30]	Doing element-wise computations in Python loops is inefficient; use vectorized or library-based methods.
Excessive GPU Transfers	Frequently moving data between CPU and GPU (e.g., `.cpu()` then `.cuda()`) leads to overhead and stalls.
Excessive Training [34]	Continuing to train a model after validation metrics plateau wastes time and resources. Use early stopping or convergence criteria.
Filter Operations [30]	When performing a filter operator on an array, tensor, or dataframe inside loops. Use Boolean indexing or masking for efficiency.
Ignoring Inplace Ops [35]	Failing to use in-place variants of PyTorch operations (e.g., `add_`, `mul_`) leads to extra memory allocations.
Inefficient Caching of Common Arrays [36]	Recreating the same arrays or tensors repeatedly in a loop (e.g., `np.arange(0,n)`) rather than caching them.
Inefficient Data Loader Transfer [27]	Using standard pageable CPU memory for large GPU data transfers can stall the GPU; pinned memory (`pin_memory=True`) is often faster.
Inefficient Data Frame Join [37]	Performing repeated joins on large DataFrames without indices or merge strategies can be extremely slow.
Inefficient Iterrows	Using `iterrows` in Pandas to manipulate data row-by-row is far slower than vectorized methods.
Large Batch Size Memory Swapping [38]	Setting a batch size that exceeds GPU memory forces excessive swaps or CPU fallback, slowing training.
Recomputing GroupBy Results [37]	Calling `.groupby()` multiple times on the same data for similar aggregates wastes CPU time.
Reduction Operations [30]	Performing sums, means, etc. in Python loops is inefficient; libraries offer optimized vectorized reduction ops.
Redundant Model Re-Fitting [39]	Continuously calling `.fit()` on the same dataset multiple times without changes in hyperparameters or data leads to repeated overhead.

Table 1: The 20 implemented energy code smells and their respective descriptions. Where applicable, references are provided to support the formulation of the smell. For further insight into the rationale behind smells, associated libraries, and recommended optimizations, refer to the GreenCodeAnalyzer website: `https://mescribano23.github.io/GreenCodeAnalyzer/`.

the same line, the tooltip displays them consecutively in the tooltip. For every identified smell, the tooltip presents three key details:

1. The code smell rule detected
2. A description of the inefficiency
3. A suggested optimization to improve energy efficiency

A visualization of how the tool is seen in an editor window can be found in Figure 3. Additionally, the user has the option to clear the editor margins, removing all highlighted lines.

By providing a plug-in which integrates with the analyzer tool and provides visual results, we allow developers to easily spot and address energy inefficiencies directly in the development environment.

6 Experimental Setup

We design the experimental setup to assess the utility of GreenCodeAnalyzer in real-world settings. It includes a preliminary check to ensure that the defined rules can successfully detect their respective code smells, followed by an experiment to evaluate the plug-in's practical effectiveness.

A successful preliminary check is a prerequisite before proceeding to the main utility experiment. This step involves defining Python functions with deliberately crafted violations of each rule. To pass the preliminary check, every crafted violation has to be correctly identified with its corresponding code smell. This is manually inspected and verified.

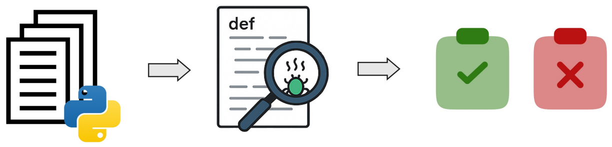


Figure 4: Diagram of the experiment flow. The plug-in takes a Python code file to analyze. This file is then matched against energy rules to identify any energy code smells. When the analysis is complete, a manual report of the true positives and false positives is done.

A general diagram of the procedure behind the utility experiment can be seen in Figure 4. A more detailed breakdown of the steps illustrated are:

1. Pick a Python file from a diverse set of repositories in the domain of artificial intelligence and data science (hereafter referred to as 'test-files'). The repositories include both personal projects and externally authored public repositories.
2. Run the plug-in on each test-file.
3. For each detected code smell, record the number of True Positives and false positives, providing descriptions of why the false positives were classified as such.
4. Repeat steps 2-3 for each test-file in the collection until at least 20 different files with at least one code smell are inspected.

The number of true positives serves as an indicator of how effective GreenCodeAnalyzer is at detecting energy-inefficient code. However, false positives are equally important. If their count is too high, it diminishes the significance of a high true positive rate, as users may become overwhelmed by excessive code smell notifications that require manual review [40].

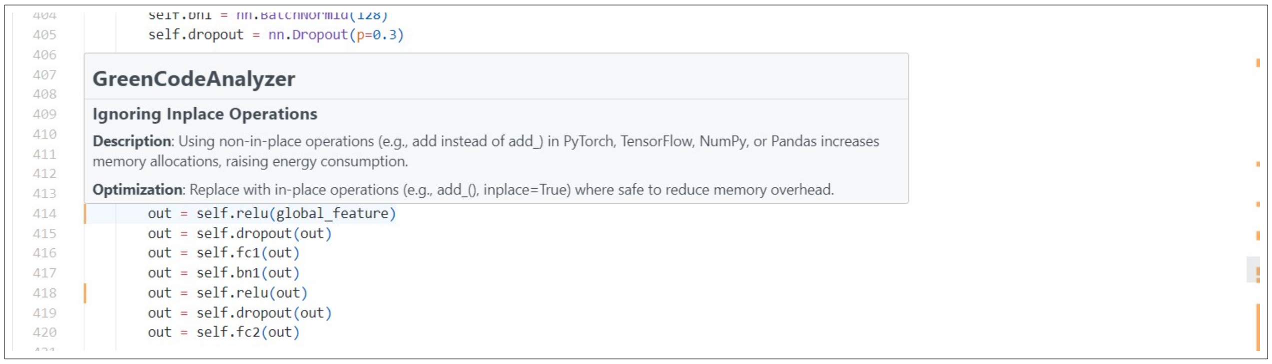
7 Results

We conduct the evaluation of GreenCodeAnalyzer in two stages: a preliminary check and the main utility experiment. The preliminary check confirms that all crafted rule violations are successfully detected and correctly identified with their respective code smells. This validation ensures that the plug-in is functioning as intended before proceeding to the utility experiment.

The results of the utility experiment are shown in Table 2. A total of 20 Python files from diverse sources are analyzed, and the total results are compiled. To collect these statistics, once each file is analyzed with the tool, we perform a manual inspection of the detected smells to classify the highlighted inefficiencies into either **true positives (TPs)** or **false positives (FPs)**. The individual results for each file are then combined to produce the final results.

As shown in Table 2, GreenCodeAnalyzer successfully detects 11 out of the 20 rules. A total of 100 energy code smells are detected across 20 files, averaging at 5 smells per file. Among these inefficiencies, we verify 78 as true positives and 22 as false positives, giving a true positive rate of 78% and a false positive rate of 22%. Only four rules contained FPs, out of which, *Excessive Training* and *Filter Operations* have the highest rates, at 40% and 39%, respectively. Additionally, *Excessive Training* is the rule with the most flagged inefficiencies, and the highest number of both TPs and FPs. In contrast, from the rules with no FPs, *Ignoring Inplace Ops* stands out for having the highest number of true positives (13) without a single misclassification.

We identify several recurring patterns that lead to false positives:



```
404         self.bn1 = nn.BatchNorm1d(128)
405         self.dropout = nn.Dropout(p=0.3)
406
```

GreenCodeAnalyzer

Ignoring Inplace Operations

Description: Using non-in-place operations (e.g., add instead of add_) in PyTorch, TensorFlow, NumPy, or Pandas increases memory allocations, raising energy consumption.

Optimization: Replace with in-place operations (e.g., add_(), inplace=True) where safe to reduce memory overhead.

```
414         out = self.relu(global_feature)
415         out = self.dropout(out)
416         out = self.fc1(out)
417         out = self.bn1(out)
418         out = self.relu(out)
419         out = self.dropout(out)
420         out = self.fc2(out)
```

Figure 3: VSCode window showcasing how the user is currently hovering over line 414, which displays the hover tooltip with details on detected code smell 'Ignoring Inplace Operations'. A description of the smell and possible optimization are also provided.

Rule	**# Smells Detected**	**# True Positives**	**# False Positives**
Blocking Data Loaders	13	10	3
Calculating Gradients	14	10	4
Chain Indexing	7	7	0
Excessive Training	28	17	11
Filter Operations	10	6	4
Ignoring Inplace Ops	13	13	0
Inefficient Caching of Common Arrays	3	3	0
Inefficient DataFrame Joins	1	1	0
Inefficient Data Loader Transfer	8	8	0
Inefficient Iterrows	1	1	0
Suboptimal Data Parallelization	2	2	0
Total	100	78	22

Table 2: Table summarizing the results of the code smells detected in the 20 test files. This table contains only rules that were matched against. For each rule we report the number of times it is detected, along with the number of true positives and the number of false positives. The last row sums the statistics for each individual smell into the total statistics.

- **Context Misinterpretation:** For example, loops flagged under *Excessive Training* are not performing actual training. These include iterations over data for computing statistics or tracking values.

- **Overgeneralized Heuristics:** The *Filter Operations* rule flags loops involving file filtering or condition-based selections, where vectorization is not practical due to I/O operations or dynamic processing logic.

- **Subtle Functional Roles:** Instances flagged under *Calculating Gradients* are actually part of non-optimization-related computations, such as diagnostics or logging.

8 Discussion

The evaluation results show that GreenCodeAnalyzer is a promising tool for detecting energy inefficiencies in Python code, particularly within data science workflows. The preliminary check demonstrates that the plug-in correctly identifies the crafted rule violations, and the utility experiment shows that the tool functions as intended when run on realistic codebases rather than specifically crafted examples.

As presented in Section 7, GreenCodeAnalyzer detects a total of 100 energy-related code smells across 20 Python files, with a 78% true positive rate. This result highlights the prevalence of energy code smells in real-world data science applications and underscores the potential for energy optimization within the field. GreenCodeAnalyzer successfully activates 11 out of the 20 implemented detection rules, demonstrating its capacity to generalize across diverse scripts. Given that data science workflows often involve complex operations on large datasets, the correct identification of inefficiencies is crucial for developers aiming to reduce energy consumption. While this coverage highlights the prevalence of energy code smells, the 22 false positives (22%) also reflect the inherent complexity of accurately identifying energy inefficiencies via static analysis.

Compared to prior work in this space, as explored in Section 3, our tool adopts a static, rule-based approach that trades depth for energy efficiency and real-time feedback within the developer's IDE. This trade-off, however, introduces challenges such as ambiguous coding contexts and overgeneralized heuristics. As shown in Section 7, instances of *Excessive Training* are often false positives. The reason being that the misclassified code lines structurally resemble training loops but serve unrelated purposes (e.g., computing means or tracking metrics). This underscores the need for more code context to distinguish these spanning structures from actual code smells. Similarly, the *Filter Operations* rule often matches against patterns that are necessary for I/O operations or dynamic logic, such as dictionary creation, but these code lines (e.g. `if file.endswith('.csv')`) are not related to filtering and cannot be optimized with boolean indexing as suggested by the rule. These kinds of errors are difficult to eliminate without deeper semantic analysis, such as program slicing, symbolic execution, or combining static analysis with lightweight runtime profiling.

From a developer's perspective, false positives introduce the risk of notification fatigue. Developers may quickly begin to ignore warnings from the tool if it regularly surfaces low-value or false alerts, which diminishes the overall trust in the system. To combat this, future iterations of GreenCodeAnalyzer could introduce confidence scores, customizable rule sets, or interactive feedback loops where developers can flag misclassifications, thereby helping the tool adapt to the project's context over time.

Related to the risk of notification fatigue is plug-in fatigue, especially for data scientists who already juggle multiple tools, extensions, and environments. For GreenCodeAnalyzer to be adopted, it needs to fit smoothly into existing workflows like VS Code or Jupyter, and avoid overwhelming users with unnecessary alerts. With a 22% false positive rate, GreenCodeAnalyzer has room for improvement in reducing misclassifications to enhance developer trust and usability.

We recognize that this is a difficult problem, and GreenCodeAnalyzer is only one piece of the solution. As far as our project goes, further refinement of the detection rules and additional testing on a broader range of Python codebases could help to reduce the occurrence of false positives and improve the tool's precision. As a community, we might benefit from a shared benchmark of annotated real-world energy inefficiencies, as well as a standardized evaluation protocol that compares tools across dimensions like recall, precision, usability, and developer trust. Encouraging open datasets and shared tooling infrastructure, similar to what the machine learning community has done for fairness and robustness, could accelerate progress.

All in all, GreenCodeAnalyzer can be an effective first step in promoting energy-efficient coding practices. However, there is room for improvement in terms of reducing false positives and further optimizing the tool's ability to detect energy inefficiencies. Moreover, the insights underscore the need for further research in context-aware static analysis and energy-aware development tools.

9 Threats to Validity

A primary limitation that affects the validity of our conclusions is the relatively small number of test files assessed. This limitation arises because evaluating the files required physical inspection for correctness. As a result, the conclusions drawn are based on a subset of files that may not fully capture the diversity of real-world projects. Consequently, the test files may not represent all the code smells or the different syntaxes associated with the code smells that our rules target. While this limitation weakens our conclusions, we still believe that the experiments provide valuable insights into the utility of the tool.

Additionally, our study does not analyze false negatives, an important aspect that could provide insight into the robustness of our rule definitions and influence user adoption. However, we opted for this approach, believing that consistently high false positives and low false negatives would be sufficiently indicative of the tool's usefulness.

Another concern arises from the manual inspection process. Given the potential for bias and errors in our evaluations, the validity of our findings could be compromised. A more robust approach would involve user studies to collect feedback or the evaluation of the plug-in's performance by a panel of external experts using a selected set of test files.

Finally, a significant threat to validity lies in the effectiveness of the optimizations in improving energy efficiency. While the optimizations were identified through a literature review, experimental validation of their effectiveness would provide stronger evidence supporting the claim that these optimizations lead to improved energy efficiency.

10 Future Work

Several directions can enhance the capabilities, effectiveness, and reliability of the GreenCodeAnalyzer static analysis tool. Recommendations for future work include:

Expanding the Ruleset. Increasing the variety and number of energy code smells identified will enhance the tool's usefulness. The current ruleset could be extended further within existing libraries, as well as expanded to additional data science libraries such as natural language processing libraries (e.g., NLTK[5]) and visualization libraries (e.g., Matplotlib[6], Seaborn[7]). Furthermore, introducing rules targeting specific machine learning model inefficiencies, such as excessively layered neural networks, would improve coverage.

Combining Static and Dynamic Analysis. While static analysis highlights potential inefficiencies early, real-world energy consumption depends on runtime execution and hardware interactions. Integrating static analysis with dynamic profiling tools can validate and refine findings.

Automated Refactoring and Suggestions. To streamline developer efforts in addressing energy inefficiencies, the analyzer could incorporate automated refactoring solutions for straightforward optimizations.

User Studies. Conducting user studies to evaluate the practical utility and usability of the tool in real-world data science workflows would provide valuable insights into its effectiveness and areas for improvement.

Quantifying Impact of Energy Code Smells. Incorporating heuristic-based energy consumption estimates or assigning severity levels to code smells would provide developers clearer guidance on the most impactful inefficiencies to address for greener coding practices.

Improving Accuracy. Enhancing the robustness and generalizability of existing rules to reduce false positives is essential for user adoption. Although achieving perfect accuracy is challenging, iterative refinement based on user feedback and additional validation can significantly improve the tool's reliability.

[5]NLTK's website: `https://www.nltk.org/`

[6]Matplotlib's website: `https://matplotlib.org/`

[7]Seaborn's website: `https://seaborn.pydata.org/`

11 Dissemination

We aim to make energy-efficient software engineering practices accessible to the community and create a broad impact, enabling data scientists, developers, and enthusiasts to adopt greener coding practices and contribute to the ongoing development of GreenCodeAnalyzer. To ensure broad reach and facilitate user engagement, we have employed several dissemination channels:

VS Code Plug-in: The core of the GreenCodeAnalyzer tool is a plug-in for Visual Studio Code, which allows users to integrate energy smells checks directly into their development environment. This provides an easy and efficient way for data scientists and software engineers to identify energy code smells in their Python code before execution.

Project Website: A dedicated website[8] for GreenCodeAnalyzer provides additional information about the plug-in, including detailed descriptions of the energy-smell rules implemented in the tool. This platform serves as a central hub for learning about the project and understanding how users can benefit from the energy-focused analysis it offers.

Open-Source GitHub Repository: The project is available as an open-source repository on GitHub[9]. The repository contains a comprehensive README file that explains how to install and use the plug-in. Additionally, a CONTRIBUTING.md file is included to encourage contributions from the open-source community. This section provides guidelines for anyone interested in improving or extending the tool, ensuring that GreenCodeAnalyzer remains a collaborative and evolving resource.

Social Media Presence: To further engage with the community and raise awareness, we have established a public Instagram page (@greencodeanalyzer). Through this platform, we share updates, best practices, and tips on energy-efficient coding, helping to build a community around sustainable software engineering.

12 Conclusion

In this paper, we presented GreenCodeAnalyzer, a static analysis tool implemented as a VSCode plug-in,

[8]GreenCodeAnalyzer's dedicated website: `https://mescribano23.github.io/GreenCodeAnalyzer/`

[9]GreenCodeAnalyzer's Github repository: `https://github.com/ianjoshi/green-code-analyzer`

designed to detect energy-related code smells in Python within the data science domain. The tool utilizes Python's Abstract Syntax Trees (ASTs) to efficiently analyze source code and identify energy inefficiencies without requiring code execution. The tool currently supports 20 static analysis rules, found through analyzing literature and practical observations. It offers actionable recommendations for optimization, promoting energy-aware development practices. This capability is crucial for enabling developers to proactively address potential energy consumption issues early in the development lifecycle.

The evaluation of GreenCodeAnalyzer on 20 Python files demonstrated its effectiveness in detecting 100 energy code smells, with a true positive rate of 78%. The findings confirm the prevalence of energy inefficiencies in real-world data science applications and highlight the potential for significant energy savings through targeted code optimizations.

While the results are promising, there is room for improvement, particularly in reducing false positives and expanding the tool's ruleset. Future work includes incorporating dynamic analysis, automating refactoring suggestions, conducting user studies, and quantifying the impact of energy code smells.

In conclusion, GreenCodeAnalyzer represents a significant and valuable contribution to the field of sustainable software engineering, specifically in the domain of data science. The project is available as an open-source repository on GitHub, encouraging contributions from the community to improve and expand the tool. By providing a practical and effective tool for identifying and addressing energy inefficiencies in Python code, it empowers developers to create more energy-efficient software, ultimately contributing to a more sustainable technological ecosystem.

References

[1] I. E. Agency, "Electricity 2024," 2024, licence: CC BY 4.0. [Online]. Available: https://www.iea.org/reports/electricity-2024

[2] E. Gelenbe and Y. Caseau, "The impact of information technology on energy consumption and carbon emissions," *Ubiquity*, vol. 2015, no. June, Jun. 2015. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/2755977

[3] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102609, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642321000022

[4] E. Masanet, A. Shehabi, N. Lei, S. Smith, and J. Koomey, "Recalibrating global data center energy-use estimates," *Science*, vol. 367, no. 6481, pp. 984–986, 2020. [Online]. Available: https://doi.org/10.1126/science.aba3758

[5] S. Team, "pyjoules: A python library to capture the energy consumption of code snippets." [Online]. Available: https://github.com/powerapi-ng/pyJoules

[6] B. Courty, V. Schmidt, S. Luccioni, Goyal-Kamal, MarionCoutarel, B. Feld, J. Lecourt, LiamConnell, A. Saboni, Inimaz, supatomic, M. Léval, L. Blanche, A. Cruveiller, ouminasara, F. Zhao, A. Joshi, A. Bogroff, H. de Lavoreille, N. Laskaris, E. Abati, D. Blank, Z. Wang, A. Catovic, M. Alencon, M. Stechły, C. Bauer, L. O. N. de Araújo, JPW, and MinervaBooks, "mlco2/codecarbon: v2.4.1," May 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11171501

[7] C. Pulido, I. García, M. Ángeles Moraga, F. García, and C. Calero, "Pypen: Code instrumentation tool for dynamic analysis and energy efficiency evaluation," *Computer Standards Interfaces*, vol. 94, p. 104000, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920548925000297

[8] United Nations, "Sustainable development goal 13: Climate action," https://sdgs.un.org/goals/goal13, 2025, accessed: 2025-04-03.

[9] C. Groza, A. Dumitru-Cristian, M. Marcu, and R. Bogdan, "A developer-oriented framework for assessing power consumption in mobile applications: Android energy smells case study," *Sensors (Basel, Switzerland)*, vol. 24, no. 19, p. 6469, 2024. [Online]. Available: https://doi.org/10.3390/s24196469

[10] K. Wood, P. Krause, and A. Mason, "Green ict: Oxymoron or call to innovation?" 10 2010, pp. 116–121.

[11] K. Eder and J. P. Gallagher, *Energy-Aware Software Engineering.* InTechOpen, 2017, pp. 103–127. [Online]. Available: https://doi.org/10.5772/65985

[12] L. de Roucy-Rochegonde and A. Buffard, "Ai, data centers and energy demand: Reassessing and exploring the trends," Ifri, 2025.

[13] M. Morisio, A. Vetro, and G. Procaccianti, "Definition, implementation and validation of energy code smells: an exploratory study on an embedded system," 03 2013, pp. 34–39.

[14] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[15] R. P. Gurung, J. Porras, and J. Koistinaho, "Static code analysis for reducing energy code smells in different loop types: A case study in java," in *2024 10th International Conference on ICT for Sustainability (ICT4S)*, 2024, pp. 292–302.

[16] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 470–481.

[17] L. N. Q. Do, J. R. Wright, and K. Ali, "Why do software developers use static analysis tools? a user-centered study of developer needs and motivations," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 835–847, 2022.

[18] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1419–1457, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09750-5

[19] J. Sallou, L. Cruz, and T. Durieux, "Energibridge: Empowering software sustainability through cross-platform energy measurement," 2023. [Online]. Available: https://arxiv.org/abs/2312.13897

[20] L. Mukhanov, D. S. Nikolopoulos, and B. R. de Supinski, "Alea: Fine-grain energy profiling with basic block sampling," 2016. [Online]. Available: https://arxiv.org/abs/1504.00825

[21] A. Schuler and G. Kotsis, "Manai – an intellij plugin for software energy consumption profiling," 2022. [Online]. Available: https://arxiv.org/abs/2205.03120

[22] Green Code Initiative, "Green code initiative: Réduisons l'impact environnemental et social des solutions numériques," 2025, accessed: 2025-04-03. [Online]. Available: https://green-code-initiative.org/

[23] GreenCode, "Greencode - visual studio marketplace," 2025, accessed: 2025-04-03. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=GreenCode.greencode

[24] ec0lint, "ec0lint: Create sustainable digital environment," 2025, accessed: 2025-04-03. [Online]. Available: https://ec0lint.com/

[25] S. Wegener, K. K. Nikov, J. Nunez-Yanez, and K. Eder, "Energyanalyzer: Using static wcet analysis techniques to estimate the energy consumption of embedded applications." Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2023.9

[26] H. Jiang, H. Yang, S. Qin, Z. Su, J. Zhang, and J. Yan, "Detecting energy bugs in android apps using static analysis," in *Formal Methods and Software Engineering*, Z. Duan and L. Ong, Eds. Cham: Springer International Publishing, 2017, pp. 192–208.

[27] A. Azzoug, "Greenpydata: Sonarqube plugin for pytorch data scientists," 2025, accessed: 2025-04-03. [Online]. Available: https://github.com/AghilesAzzoug/GreenPyData

[28] P. Carbonnelle, "Pypl popularity of programming language index," 2025, accessed: 12 March 2025. [Online]. Available: https://pypl.github.io/PYPL.html

[29] A. Finamore, "Abstract syntax trees in python," *Pybites*, 2021, accessed: 2025-04-04. [Online]. Available: https://pybit.es/articles/ast-intro/

[30] M. Haakman, "Dslinter: Pylint plugin for data science and machine learning code," 2020, accessed: 2025-04-04. [Online]. Available: https://github.com/MarkHaakman/dslinter

[31] V. Kazemi, "Effective tensorflow 2: Broadcasting the good and the ugly," 2019, accessed: 2025-04-04. [Online]. Available: https://github.com/vahidk/EffectiveTensorflow#broadcasting-the-good-and-the-ugly

[32] PyTorch Contributors, "torch.no$_grad,'' 2024, accessed : 2025-04-04.[Online].Available :$

S. O. Community, "What is the purpose of with torch.no$_grad() :,'' 2022, accessed : 2025-04-04.[Online].Available :$

R. Caruana, S. Lawrence, and C. Giles, “Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping,” in *Advances in Neural Information Processing Systems*, T. Leen, T. Dietterich, and V. Tresp, Eds., vol. 13. MIT Press, 2000. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2000/file/059fdcd96baeb75112f09fa1dcc740cc-Paper.pdf

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

C. Breshears, *The Art of Concurrency: A Thread Monkey’s Guide to Writing Parallel Applications.* O’Reilly Media, 2009. [Online]. Available: https://archive.org/details/artofconcurrency0000bres

W. McKinney, *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython.* O’Reilly Media, 2017.

T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, A. Müller, J. Nothman, G. Louppe, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Édouard Duchesnay, “Scikit-learn: Machine learning in python,” 2018. [Online]. Available: https://arxiv.org/abs/1201.0490

F. Cheirdari and G. Karabatis, “Analyzing false positive source code vulnerabilities using static analysis tools,” in *2018 IEEE International Conference on Big Data (Big Data)*, 2018, pp. 4782–4788.