SE4AI & IGES

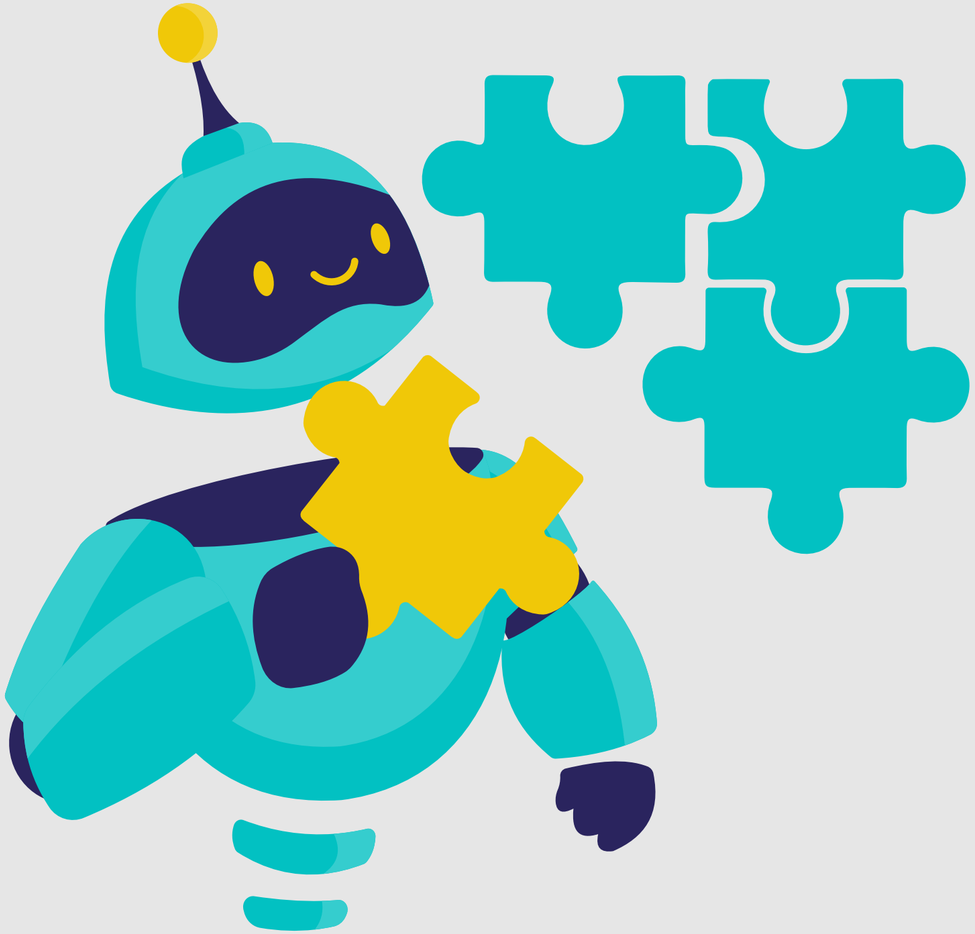
CodeSmile

ML-specific code smells detection

SOURCE CODE

DOCS

Table of Contents



Project Context

Context

Machine Learning (ML) pipelines are often **complex**, and require quality assurance

ML-specific **code smells** are suboptimal implementation patterns that lower quality, maintainability and scalability

Inefficiencies & **Technical Debt**

Code Smells for Machine Learning Applications

Haiyin Zhang
haiyin.zhang@ing.com
AI for Fintech Research, ING
Amsterdam, Netherlands

Luís Cruz
L.Cruz@tudelft.nl
Delft University of Technology
Delft, Netherlands

Arie van Deursen
Arie.vanDeursen@tudelft.nl
Delft University of Technology
Delft, Netherlands

ABSTRACT

The popularity of machine learning has wildly expanded in recent years. Machine learning techniques have been heatedly studied in academia and applied in the industry to create business value. However, there is a lack of guidelines for code quality in machine learning applications. In particular, code smells have rarely been studied in this domain. Although machine learning code is usually integrated as a small part of an overarching system, it usually plays an important role in its core functionality. Hence ensuring code quality is quintessential to avoid issues in the long run. This paper proposes and identifies a list of 22 machine learning-specific code smells collected from various sources, including papers, grey literature, GitHub commits, and Stack Overflow posts. We pinpoint each smell with a description of its context, potential issues in the long run, and proposed solutions. In addition, we link them to their respective pipeline stage and the evidence from both academic and grey literature. The code smell catalog helps data scientists and developers produce and maintain high-quality machine learning application code.

KEYWORDS

Code Smell, Anti-pattern, Machine Learning, Code Quality, Technical Debt

ACM Reference Format:
Haiyin Zhang, Luís Cruz, and Arie van Deursen. 2022. Code Smells for Machine Learning Applications. In *1st Conference on AI Engineering - Software Engineering for AI (CAIN'22), May 16–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3522664.3528620

1 INTRODUCTION

Despite the large increase in the popularity of machine learning applications [3], there are several concerns regarding the quality concerns that practitioners are eager to learn more about engineering best practices for their machine learning applications [5].

There has been a lot of interest in various machine learning system artifacts, including models and data. Researchers make efforts to improve machine learning model quality [10] and data quality [7]. However, the quality assurance of machine learning code has not been highlighted [12]. Recent work studied the code quality for machine learning applications in a general way, finding some code quality issues such as duplicated code [20] and violations of traditional naming convention [17]. These works highlighted the fact that the existing code conventions do not necessarily fit the context of machine learning applications. For example, the typical math notation in data science tasks clashes with the naming conventions of Python [20]. Thus, we argue that more research is needed to accommodate the particularities of data-oriented codebases.

As an important artifact in the machine learning application, the quality of the code is essential. Low-quality code can lead to catastrophic consequences. In the meantime, different from traditional software, machine learning code quality is more challenging to evaluate and control. Low-quality code can lead to silent pitfalls that exist somewhere that affect the software quality, which takes a lot of time and effort to discover [22]. Therefore, it is non-trivial to improve the code quality during the development process and consider code quality assurance in the deployment process.

A common strategy to improve code quality is eliminating code smells and anti-patterns. When we talk about code smells in this paper, we refer them to the pitfalls that we can inspect at the code level but not at the data or model level. We use the term "pitfall" to represent issues that degrade the software quality. Listing 1 shows an example of such pitfalls using Python and the Pandas library. In the red (-) part of the listing, an inefficient loop is created. A better alternative is highlighted in green (+), using Pandas built-in API to replace the loop, which operates faster. While some alter-

When Code Smells Meet ML: On the Lifecycle of ML-specific Code Smells in ML-enabled Systems

Gilberto Recupito
Sesa Lab - University of Salerno
Salerno, Italy
grecupito@unisa.it

Giammaria Giordano
Sesa Lab - University of Salerno
Salerno, Italy
giagiordano@unisa.it

Filomena Ferrucci
Sesa Lab - University of Salerno
Salerno, Italy
fferrucci@unisa.it

Dario Di Nucci
Sesa Lab - University of Salerno
Salerno, Italy
ddinucci@unisa.it

Fabio Palomba
Sesa Lab - University of Salerno
Salerno, Italy
fpalomba@unisa.it

ABSTRACT

Context. The adoption of Machine Learning (ML)–enabled systems is steadily increasing. Nevertheless, there is a shortage of ML-specific quality assurance approaches, possibly because of the limited knowledge of how quality-related concerns emerge and evolve in ML-enabled systems. **Objective.** We aim to investigate the emergence and evolution of specific types of quality-related concerns known as ML-specific code smells, *i.e.*, sub-optimal implementation solutions applied on ML pipelines that may significantly decrease both quality and maintainability of ML-enabled systems. More specifically, we present a plan to study ML-specific code smells by empirically analyzing (i) their prevalence in real ML-enabled systems, (ii) how they are introduced and removed, and (iii) their survivability. **Method.** We will conduct an exploratory study, mining a large dataset of ML-enabled systems and analyzing over 400k commits about 337 projects. We will track and inspect the introduction and evolution of ML smells through CodeSmile, a novel ML smell detector that we will build to enable our investigation and to detect ML-specific code smells.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Technical Debt; ML-Specific Code Smells; Software Quality Assurance; Software Engineering for Artificial Intelligence.

ACM Reference Format:
Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci,

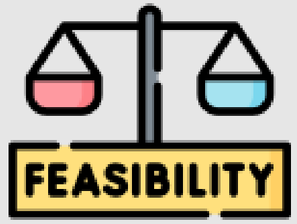
1 INTRODUCTION

Machine Learning (ML) evolved through the emergence of complex software integrating ML modules, defined as ML-enabled systems [13]. Self-driving cars, voice assistance instruments, or conversational agents like ChatGPT[1] are just some examples of the successful integration of ML within software engineering projects.

However, the strict time-to-market and change requests pressure practitioners to roll out immature software to keep pace with competitors, leading to the possible emergence of technical debt [7] *i.e.*, a technical trade-off that can give benefits in a short period, but that can compromise the software health in the long run. *Code smells* is a manifestation of technical debt. They are *symptoms* of poor design and implementation choices that, if left unaddressed, can deteriorate the overall quality of the system [8].

Sculley et al. [19] showed that ML-enabled systems are incredibly prone to technical debt and code smells, raising the need for a quality assurance process for ML components. Cardozo et al. [3] and Van Oort et al. [24] argued that while the issues in those systems are emerging, there is a lack of quality assurance tools and practices that ML developers can use. This lack of quality management assets stimulates the proliferation of code smells in ML-enabled systems [12]. Consequently, given the complex nature of those systems, new types of code smells have emerged. Considering the aspects that ML developers face when dealing with ML pipelines, Zhang et al. [29] defined a new form of code smells, *AI-specific code smells* (ML-CSs). Similarly to traditional code smells, an ML-CS is defined as a *sub-optimal implementation solution for ML pipelines that may significantly decrease the quality of ML-enabled systems.* A key example of those quality issues is using a loop operation

Study on



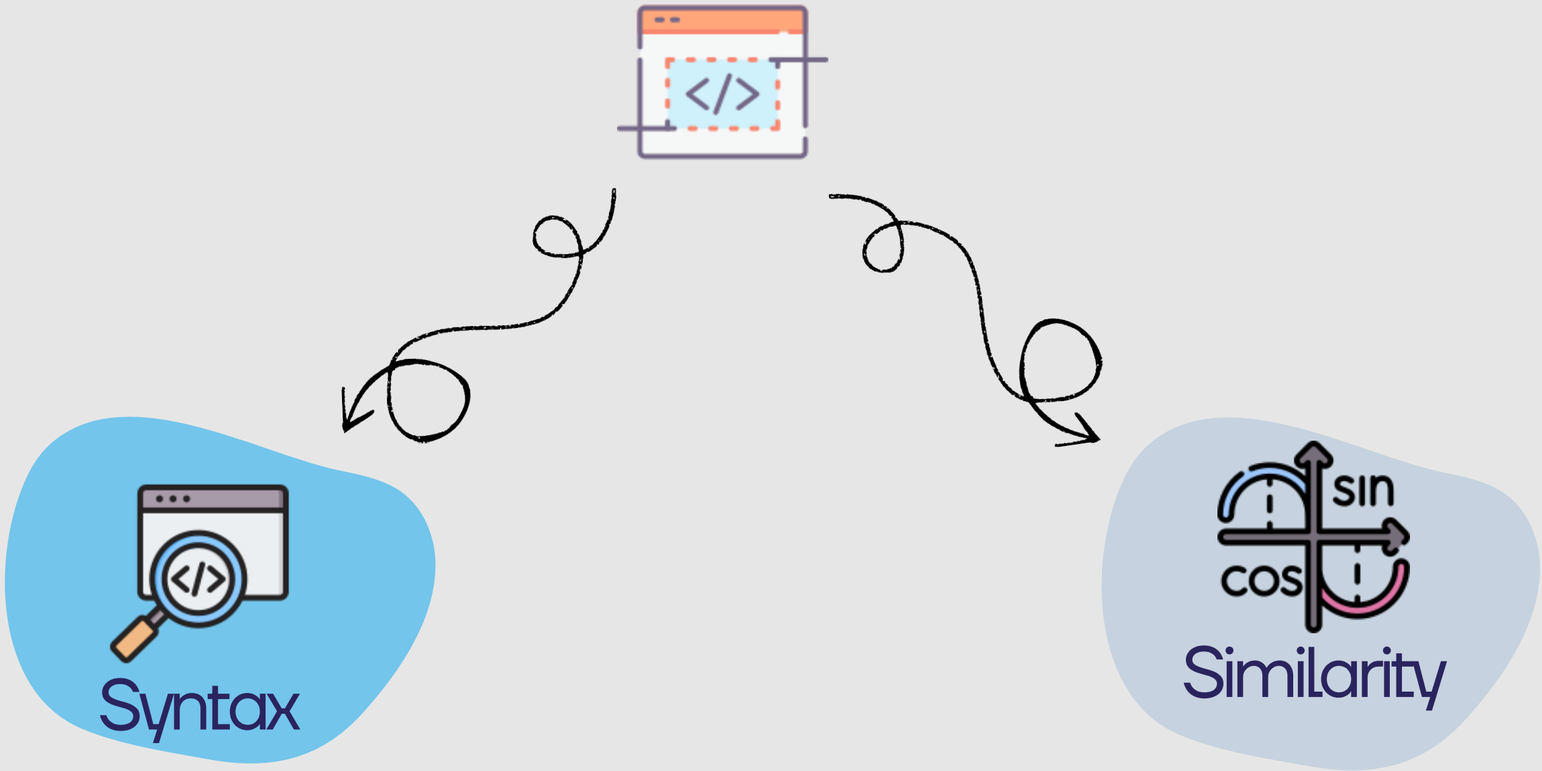
Smell Injection through LLM

Detection through LLM

Project Goals

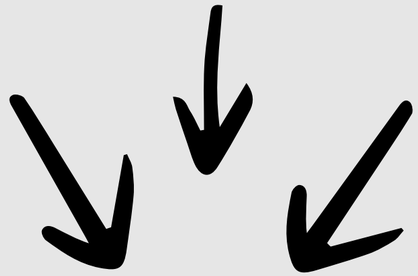
RQ1

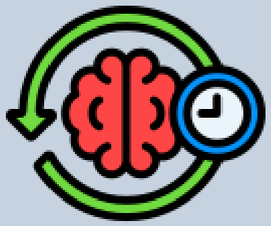
To what extent can LLMs accurately generate realistic training data with artificially injected code smells?



RQ2A

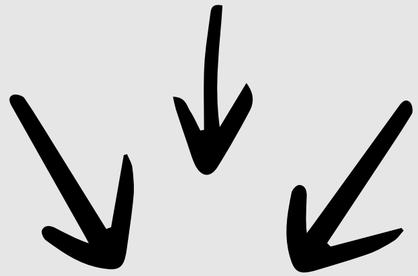
How effective is the module at identifying **individual** code smells within a single code snippet?

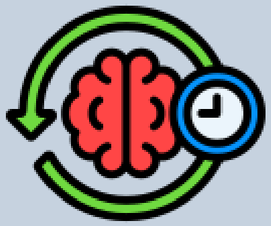


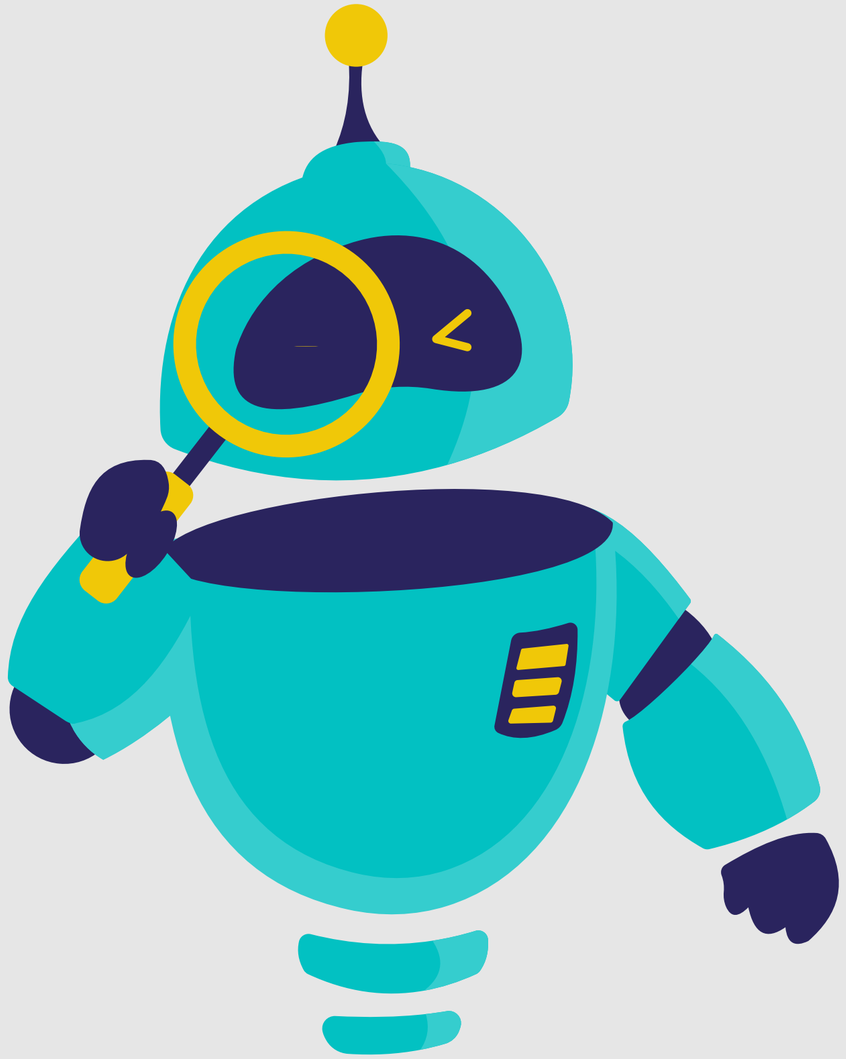


RQ2B

How effective is the module at detecting **multiple** code smells coexisting within a single code snippet?







Methodical Steps

RQ1

Qwen2.5-Coder 14B

ML-related code snippets extraction from GitHub Repos

CoT Prompt Definition for ML specific smell injection task

Balanced Dataset Construction using synthetic and real smelly snippets

Syntax Validation of each entry of the augmented dataset

Computation of cosine similarity of each entry pair, using Tf-idf

RQ2A/B

Qwen2.5-Coder xB

Balanced Scenario: 70/30 split of both real and synthetic code smells

Unbalanced Scenario: synthetic against real instances

CoT Prompt Definition for ML specific smell detection task (single label and multi-lable)

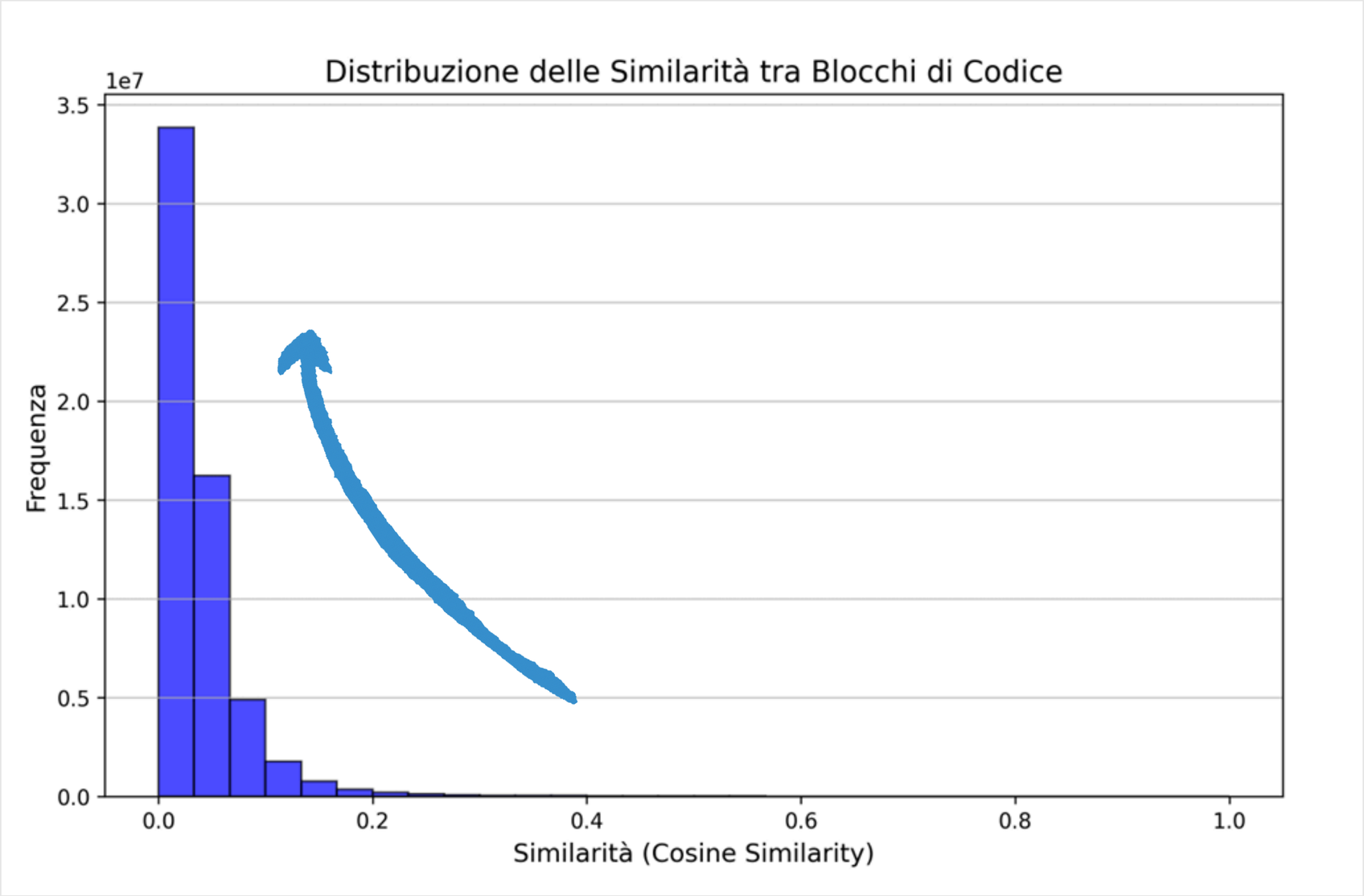
FIne-Tuning(LoRa) eacn model variant in both Scenarios

Evaluation (Precision, Recall F1-score) of each variant

Results

RQ1

97 Syntax Validation



RQ2A

Balanced Split Syntethic & Real

Classe	Precision	Recall	F1-score
Unnecessary Iteration	0.80	0.93	0.86
Hyperparameter Not Explicitly Set	0.84	0.88	0.86
NaN Equivalence Comparison Misused	0.98	0.97	0.98
Empty Column Misinitialization	0.98	0.96	0.97
Gradients Not Cleared Before Backward Propagation	0.82	0.87	0.85
TensorArray Not Used	0.94	0.93	0.94
Dataframe Conversion API Misused	0.90	0.88	0.89
Deterministic Algorithm Option Not Used	0.99	0.99	0.99
Chain Indexing	0.88	0.76	0.81
No Smell	0.98	0.98	0.98
Matrix Multiplication API Misused	0.97	0.91	0.94
Merge API Parameter Not Explicitly Set	0.93	0.93	0.93
Memory Not Freed	0.93	0.90	0.92
Broadcasting Feature Not Used	0.98	0.97	0.98
In-Place APIs Misused	0.83	0.74	0.78
Columns and DataType Not Explicitly Set	0.89	0.94	0.92
PyTorch Call Method Misused	0.94	0.92	0.93
Micro avg	0.95	0.95	0.95
Macro avg	0.92	0.91	0.91
Weighted avg	0.95	0.95	0.95
Samples avg	0.95	0.95	0.95

RQ2A

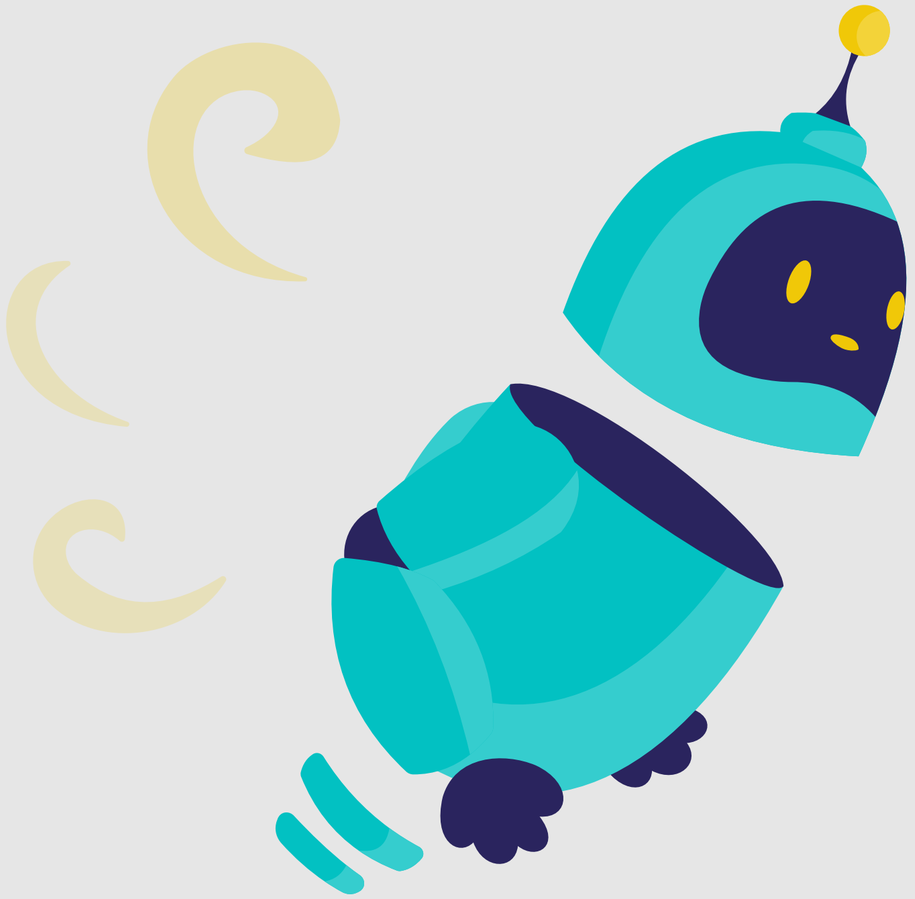
Unbalanced Syntethic Vs Real

Class	Precision	Recall	F1-score
Deterministic Algorithm Option Not Used	1.00	0.10	0.18
Chain Indexing	0.00	0.00	0.00
Empty Column Misinitialization	0.00	0.00	0.00
Matrix Multiplication API Misused	0.25	0.09	0.12
In-Place APIs Misused	0.00	0.00	0.00
Columns and DataType Not Explicitly Set	0.93	0.09	0.17
Merge API Parameter Not Explicitly Set	0.00	0.00	0.00
Dataframe Conversion API Misused	0.09	0.06	0.07
PyTorch Call Method Misused	0.00	0.00	0.00
NaN Equivalence Comparison Misused	0.00	0.00	0.00
Memory Not Freed	0.00	0.00	0.00
Gradients Not Cleared Before Backward Propagation	0.48	0.08	0.13
Hyperparameter Not Explicitly Set	0.50	0.03	0.06
Unnecessary Iteration	0.00	0.00	0.00
Micro avg	0.47	0.03	0.03
Macro avg	0.27	0.02	0.03
Weighted avg	0.54	0.03	0.04
Samples avg	0.04	0.04	0.04

RQ2B

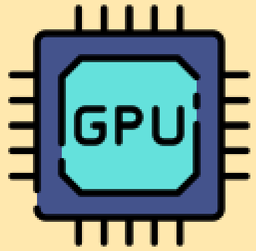
Given our limited hardware resources, even the most complex model we could afford to use (14B) was unable to properly inject multiple smells in a single code snippet, often resulting in malformed data

The lack of multi-labeled code snippets meant obviously that we could not possiply train any model variant for detecting multiple smells instances in a given code snippet



Challenges and Limitations

Hardware Issues



Limited Available Hardware meant we could not use more complex models

A narrow context window meant we could not instruct to perform harder tasks

Domain Issues

There is a very limited amount of available scientific literature in the domain of ML-specific code smells

There is not a public available dataset which contains manually labeled and validated ML-specific code smells

Implications & Future Work

Dataset

Synthetic dataset showed high diversity
and an excellent degree of syntax validity

Need for even larger datasets for improved generalization if researchers aim to leverage models with higher complexity

Functionalities

Enhancing prompt engineering could further extend model functionalities

Multi-label detection and multiple code smells injection both require more resources and further study.

Project Context

Project Goals

Methodical Steps

Results

Challenges and Limitations

Implications & Future Work

SALFORD & CO.

Thank You!

HTTPS://GITHUB.COM/XDARYAMO/SMELL_AI

SOURCE CODE

DOCS

SE4AI & IGES

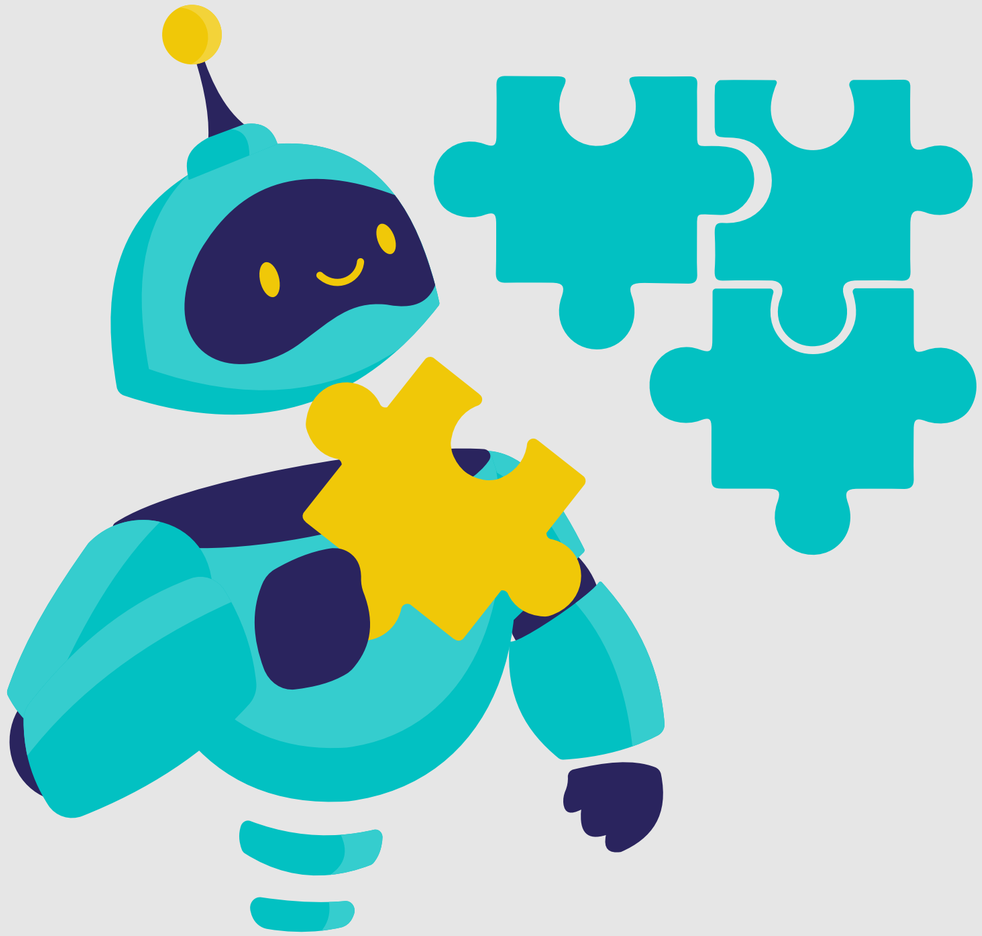
CodeSmile

ML-specific code smells detection

SOURCE CODE

DOCS

Table of Contents

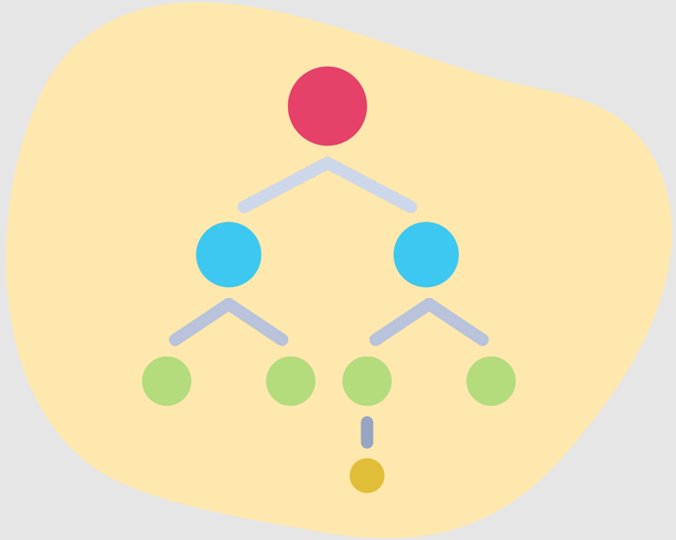


Introduzione & Contesto

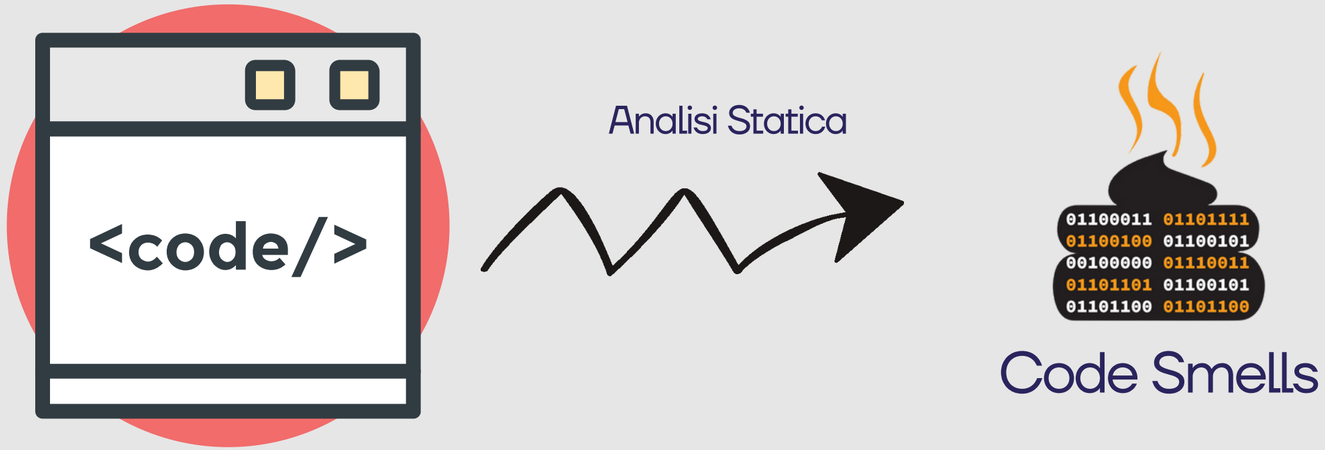
Contesto

CodeSmile si inquadra all'interno del contesto del **Technical Debt** e, in particolare, tratta della problematica dell' **AI-Specific** Technical Debt.

Attraverso l'analisi **statica** di code snippets scritti in Python, il tool mira ad individuare pattern sub-ottimali (**ML-specific Code Smells**), che potrebbero compromettere la manutenibilità, la leggibilità o la robustezza del codice analizzato.

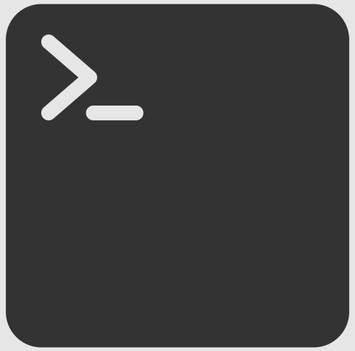


CodeSmile in Sintesi

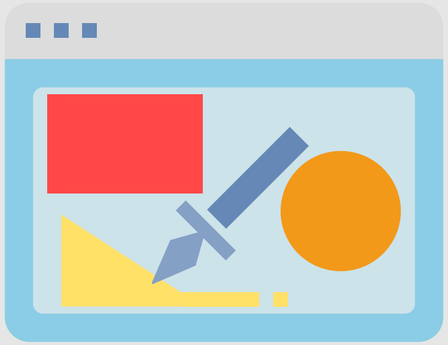


CodeSmile effettua un'**analisi statica** del codice dei moduli Python di un progetto, utilizzando Abstract Syntax Trees (AST) e regole specifiche per il rilevamento dei code smells (attualmente è in grado di identificare 16 tipologie distinte).

CodeSmile in Sintesi

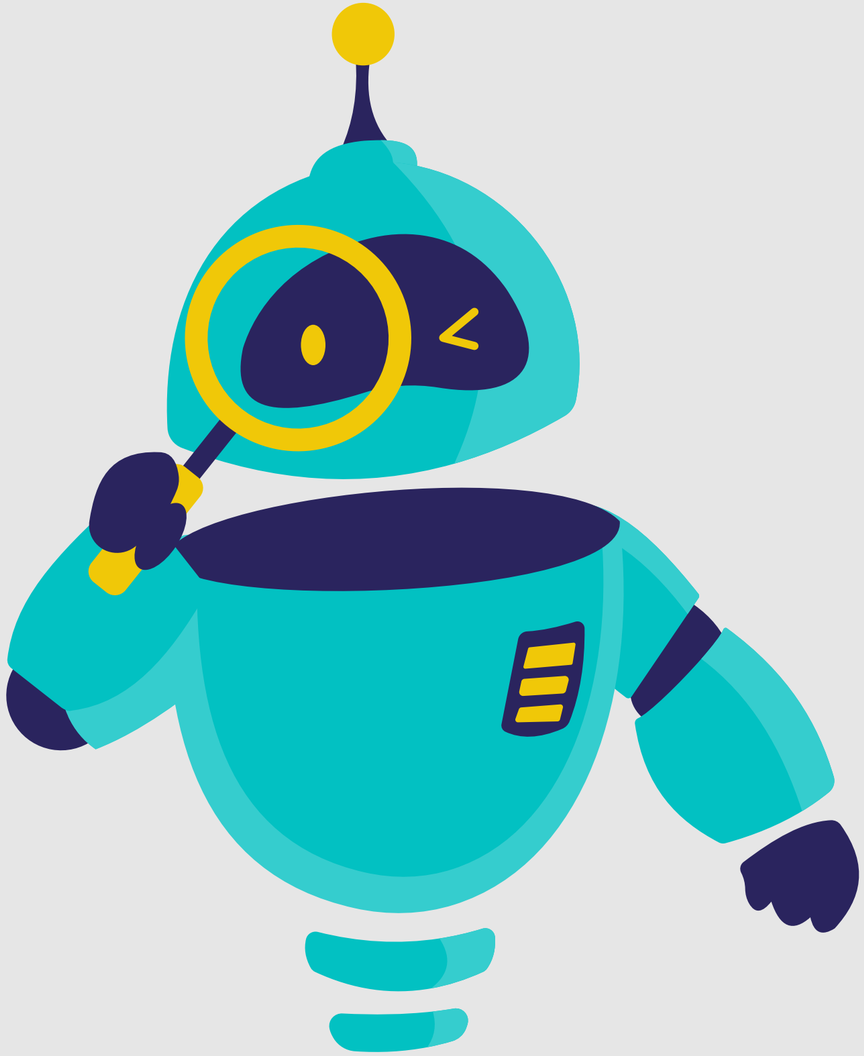


or



CLI GUI

Il tool può essere eseguito da CLI o tramite una semplice GUI e permette di specifare una serie di parametri che ne influenzeranno l'esecuzione.
I risultati, salvati come file .csv, possono essere usati di seguito per generare dei report con la stessa estensione.

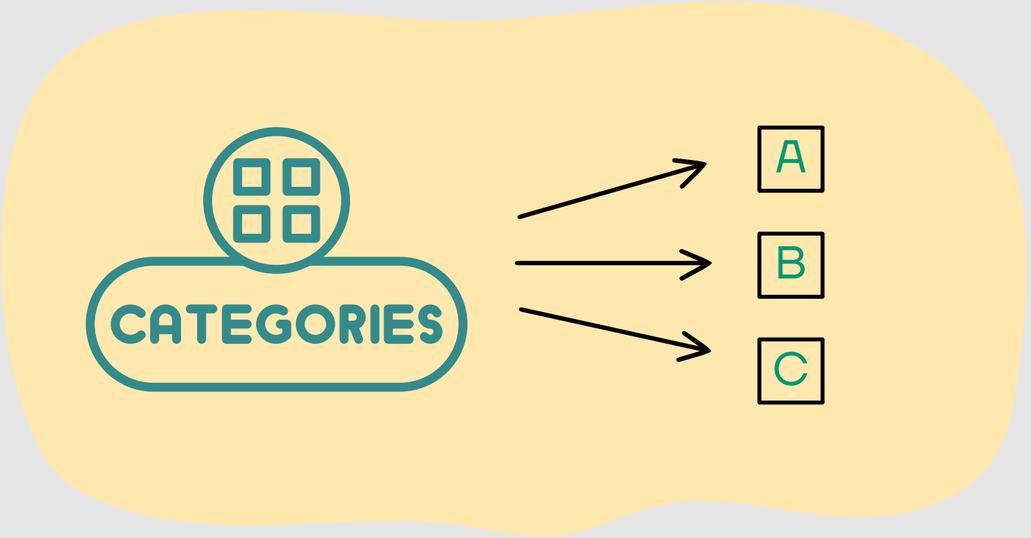


Attività pre-modifica

System Testing

Le funzionalità principali del sistema sono due: quella di **analisi** e quella di **reporting**.
In assenza di una test suite pre-esitente, è stato deciso di svilupparne una che ne valutasse il comportamento globale.

I relativi test sono stati definiti in modalità **black-box**, facendo uso della tecnica del **category partitioning**.



Parametri

I **parametri** definiti durante il testing di sistema sono i seguenti:

File:
numero & estensione

Smell:
Generico o Api-specific

Struttura:
numero proj e relativa struttura

Configurazione del tool

Incidents

#1

Descrizione

Tutti i test case falliscono quando si esegue il tool dalla root del progetto. Viene cercato un file di log in un percorso errato. E' necessario eseguirlo dalla directory **controller**.

Soluzione

Modificare il modulo **controller** per supportare esecuzioni sia dalla root che dalla directory specifica

Incidents

#2

Descrizione

Deterministic Algorithm Option Not Used non viene rilevato e non è incluso nell'output. Manca un controllo completo su gli usi di **use_deterministic_algorithms** del modulo torch

Soluzione

Modificare la funzione **deterministic_algorithm_option_not_used** in modo da riconoscere gli usi di use_deterministic_algorithms

Incidents

#3

Descrizione

Nonostante il tool analizzi dei progetti vuoti (o che contengono file non Python) o privi di smells, vengono creati dei file to_save.csv **vuoti**. Nell' esecuzione da CLI, la console stampa un **Error** non specificato

Soluzione

Aggiungere un controllo per verificare se la variabile to_save sia vuoto o meno. In caso, **evitare** di restituire la variabile to_save. **Cambiare** il messaggio stampato per evitare fraintendimenti sul significato.

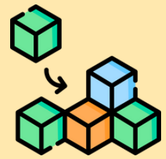
Change Requests

CR 1

Priotità Alta

Rifattorizzare il codice (attualmente procedurale) in logica OO, definendo classi e relativi metodi

FOR



Migliorare la modularità, la manutenibilità del codice e il testing grazie a funzionalità incapsulate

CR2

Priotità Media

Implementare una Web-App per consentire l'interazione con il tool via web

FOR

Migliorare l'esperienza utente e l'accessibilità. Consentire agli utenti di eseguire analisi senza conoscenze specifiche della CLI

CR3

Priotità Alta

Utilizzare Modelli di Linguaggio di Grandi Dimensioni (LLM) per creare dataset di code snippets con smells iniettati

FOR

Generare dati di addestramento per IA in assenza di dataset etichettati nel mondo reale

CR4

Priotità Alta

Addestrare un modello di IA per il rilevamento degli smells, usando il dataset generato dagli LLM e valutarne le prestazioni con metriche appropriate (accuratezza, precisione, richiamo e F1 score)

FOR

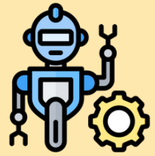
Studiare la fattibilità e le prestazioni degli LLM rispetto ai task di identificazione e di classificazione di ML-specific code smells

CR5

Priotità Bassa

Eseguire il refactoring automatico del codice al fine di correggere gli smells rilevati

FOR



Migliorare la qualità del codice rimuovendo gli smells in modo proattivo. Ottimizzare tempo e risorse degli sviluppatori

Impact Analysis

CR1

SIS	CIS	AIS	FPIS	DIS
components	*components*	*components*	∅	∅
controller	*controller*	*controller*		
cs_detector	*cs_detector*	*cs_detector*		
general_output	*general_output*	*general_output*		

Precision = 4 / 4 = 1

Recall = 4 / 4 = 1

CR 2 & 3 & 4

SIS

CIS

AIS

FPIS

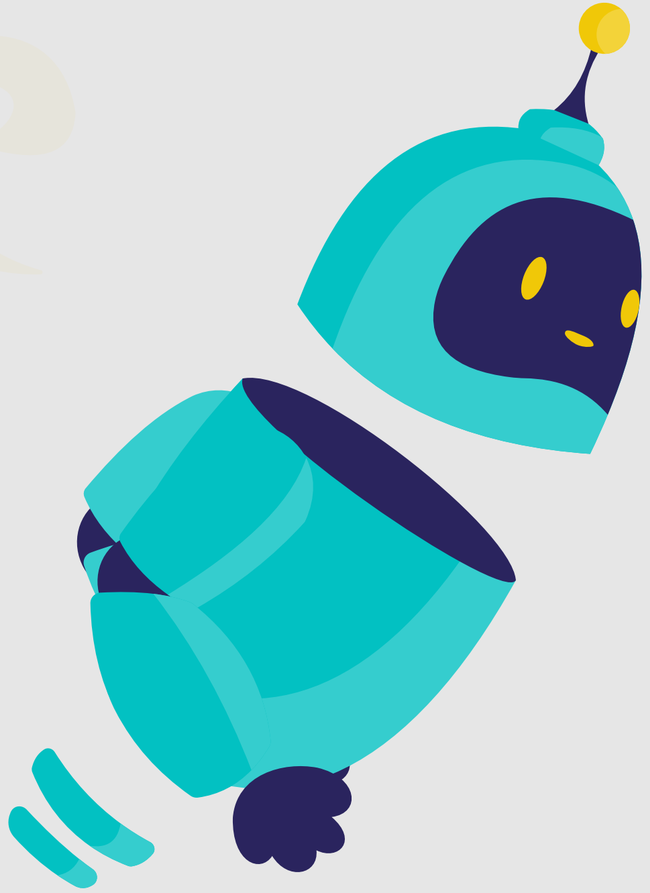
DIS

∅ ∅ ∅ ∅ ∅

Precision = 0 / 0

Recall = 0 / 0

Testing



Componenti

Incluse

Tool Ristrutturato

Web-App

Escluse

Injector

Tool IA

Unità

CR 1

Per ogni classe definita in fase di ristrutturazione, sono stati definiti dei TC che verificassero il corretto funzionamento dei metodi rispetto a **input validi** e input **non validi**. Il framework utilizzato è quello di **pytest.**

CR 2

Per ogni **componente tsx** realizzata, sono stati definiti dei TC che ne verificassero il corretto funzionamento rispetto all'interazione con l'utente.
Il framework utilizzato è quello di **jest**.

Integrazione

CR 1

pytest

Sono stati coperti diversi scenari, come il flusso di dati tra l'analyzer e l'inspector, tra CLI e analyzer , l'integrazione completa con CLI e GUI.
Inoltre, viene testata l'interazione tra GUI e analyzer, tra inspector e rule checker. Il framework utilizzato è quello di **pytest**.

CR 2

pytest

Per ogni **microservizio** realizzato con FastApi, sono stati definiti dei TC che verificassero il corretto funzionamento tra il **gateway** e il **servizio specifico** (reporting, static analysis, ai analysis)
Il framework utilizzato è quello di **pytest**.

Sistema & E2E

CR 1

Per questa attività è stata adottata la tecnica di black box. In particolare si è fatto uso del tool ristrutturato e sono stati rieseguiti i test case definiti durante le attività precedenti all' implementazione delle modifiche

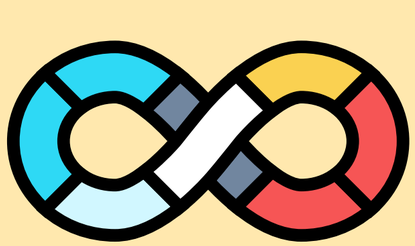
CR 2

Finalizzato a verificare il corretto funzionamento dei principali flussi utente.
Utilizzando il framework **cypress**, è stata definita una suite (specs) per ogni pagina che compone la web-app

Regressione

I test case di ogni tipologia (d' unità, d' integrazione e di sistema) definiti al termine della CR1 hanno costituito una suite di test robusta che è stata rieseguita al completamento di ogni CR.

In particolare i test d' unità e d'integrazione sono stati automatizzati tramite **GitHub Workflo**w. I test di sistema sono sono stati verificati con le stesse modalità delle attività pre-modifiche



Regressione

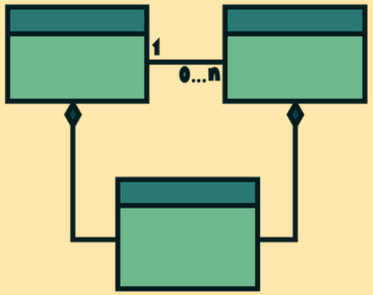
I risultati del testing di regressione per ogni CR hanno mostrato, che il sistema **non** ha risentito di anomalie o regredimenti a fronte delle modifiche. Gli **Incidents** sono stati risolti con successo.

Esito

CR 1

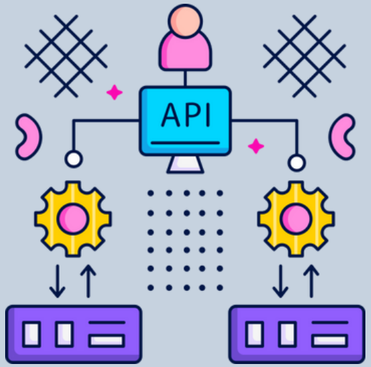
La transizione da un’architettura procedurale a una orientata agli oggetti ha portato a una struttura più **modulare e manutenibile**, migliorando l’organizzazione del codice. Il risultato finale risulta più scalabile e comprensibile rispetto alla versione precedente

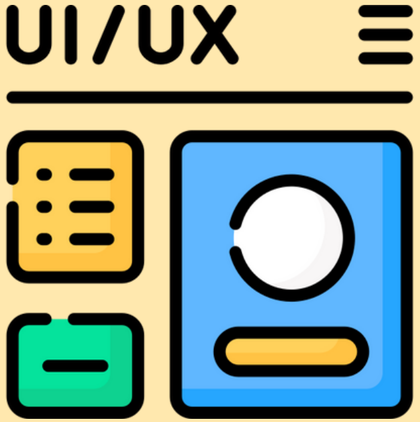
E’ stato aggiunto il **diagramma delle classi** che rispecchia la nuova architettura del tool



CR2

L’architettura dell’ applicazione web è stata progettata seguendo un approccio **orientato ai servizi**, utilizzando FastAPI. Questo ha permesso di definire servizi modulari e di facile integrazione.





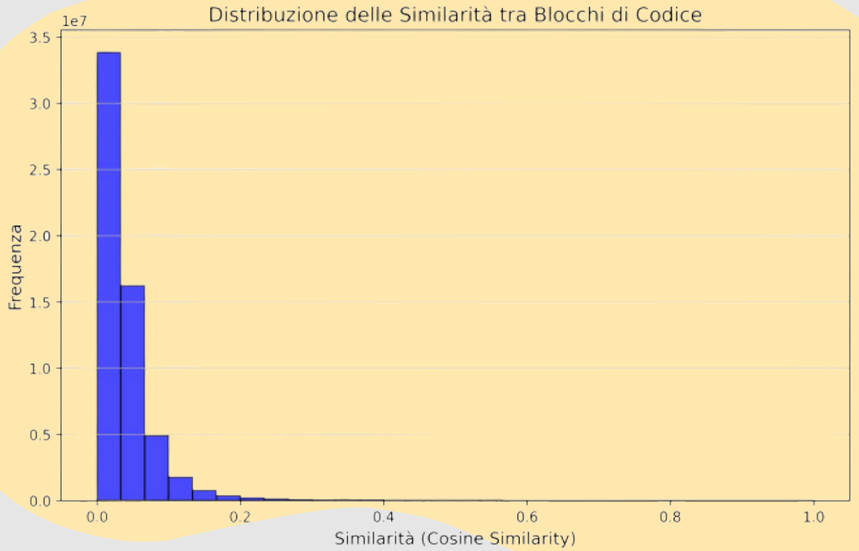
Per il frontend, è stato utilizzato il framework **Next.js**, che ha consentito di realizzare un’interfaccia utente fluida e intuitiva.

CR3

Qwen2.5-Coder

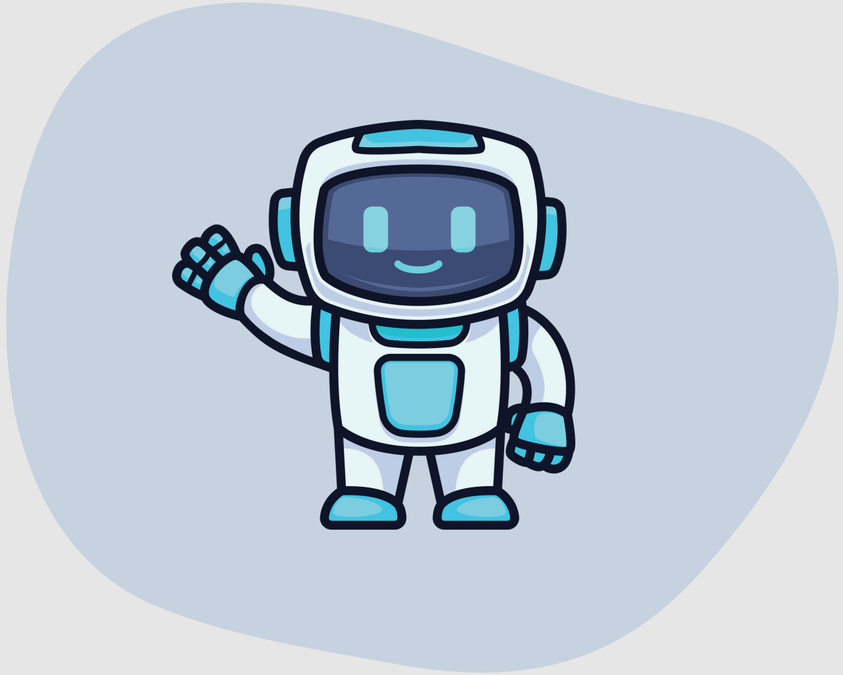
L'integrazione del modello **Qwen2.5-Coder-14B** ha permesso di creare code snippets con smells in maniera efficiente rispetto ai vincoli hardware.

Assenza di errori sintattici nel 97.38% di tutti gli snippet di codice generati e **varietà del contenuto** dei code snippets



CR4

E’ stato sviluppato un modello di IA per il rilevamento degli ML-specific code smells, sfruttando i **dataset generati nella CR3**



Il modello è stato realizzato mettendo a confronto le varianti 0.5B, 1.5B e 3B di Qwen2.5-Coder-Instruct.

In fase di valutazione, **il modello 3B** è risultato essere il più performante

Classe	Precision	Recall	F1-score
Unnecessary Iteration	0.80	0.93	0.86
Hyperparameter Not Explicitly Set	0.84	0.88	0.86
NaN Equivalence Comparison Misused	0.98	0.97	0.98
Empty Column Misinitialization	0.98	0.96	0.97
Gradients Not Cleared Before Backward Propagation	0.82	0.87	0.85
TensorArray Not Used	0.94	0.93	0.94
Dataframe Conversion API Misused	0.90	0.88	0.89
Deterministic Algorithm Option Not Used	0.99	0.99	0.99
Chain Indexing	0.88	0.76	0.81
No Smell	0.98	0.98	0.98
Matrix Multiplication API Misused	0.97	0.91	0.94
Merge API Parameter Not Explicitly Set	0.93	0.93	0.93
Memory Not Freed	0.93	0.90	0.92
Broadcasting Feature Not Used	0.98	0.97	0.98
In-Place APIs Misused	0.83	0.74	0.78
Columns and DataType Not Explicitly Set	0.89	0.94	0.92
PyTorch Call Method Misused	0.94	0.92	0.93
Micro avg	0.95	0.95	0.95
Macro avg	0.92	0.91	0.91
Weighted avg	0.95	0.95	0.95
Samples avg	0.95	0.95	0.95

Tabella 1: Metriche di valutazione del modello più performante.

Deviazioni

Le limitazioni hardware hanno impedito **l'iniezione di code smells multipli** all'interno di un singolo snippet e la realizzazione della detection con più etichette (multi-label classification)

Il tool statico fornisce la **riga specifica** in cui si manifesta il code smell e offre **suggerimenti** su come correggerlo.

Il tool AI **non dispone** di queste funzionalità a causa della **dimensione del contesto** gestibile dai modelli utilizzati

Introduzione & Contesto

Attività pre-modifica

Change Requests

Impact Analysis

Testing

Esito

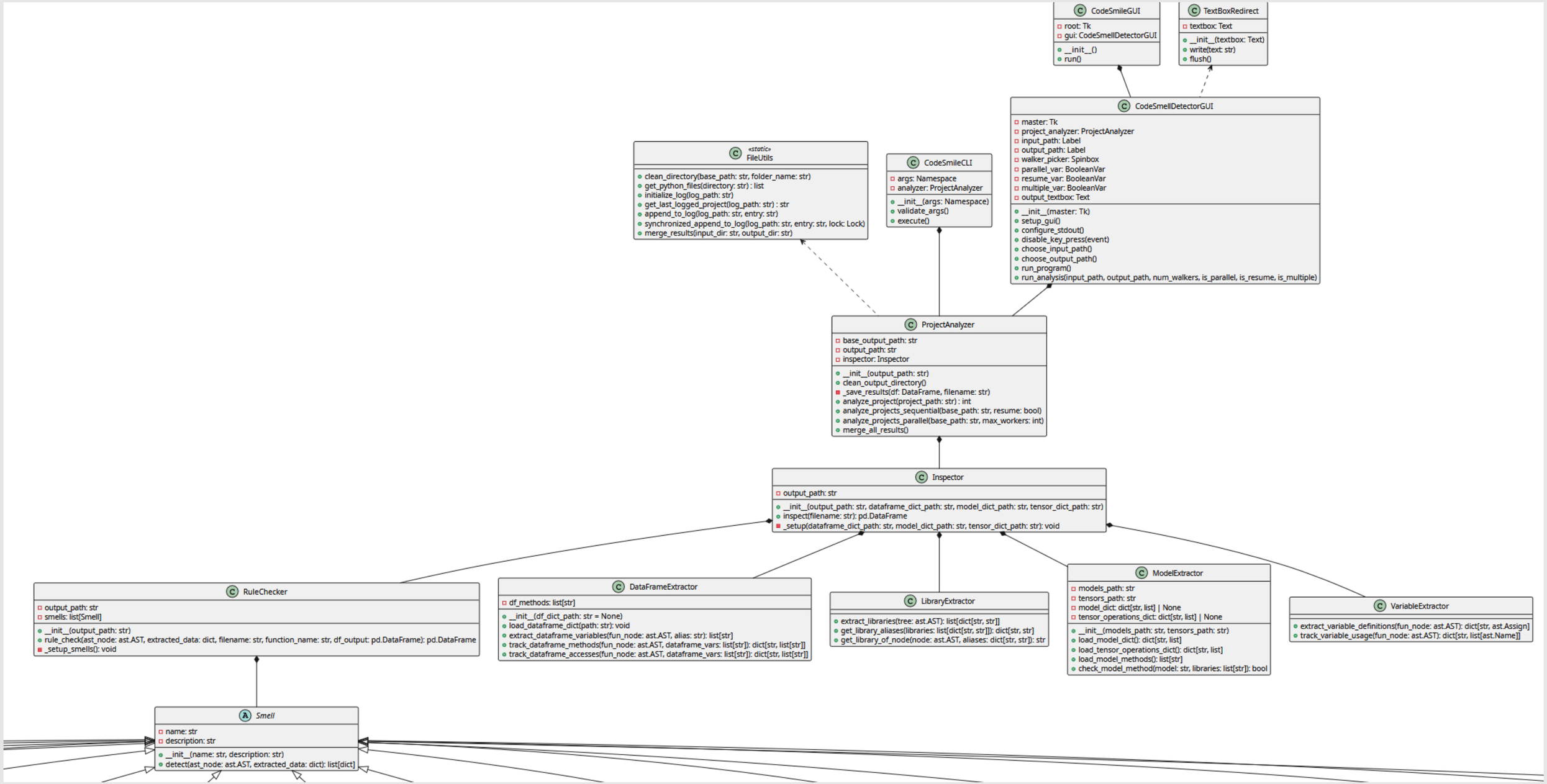
SALFORD & CO.

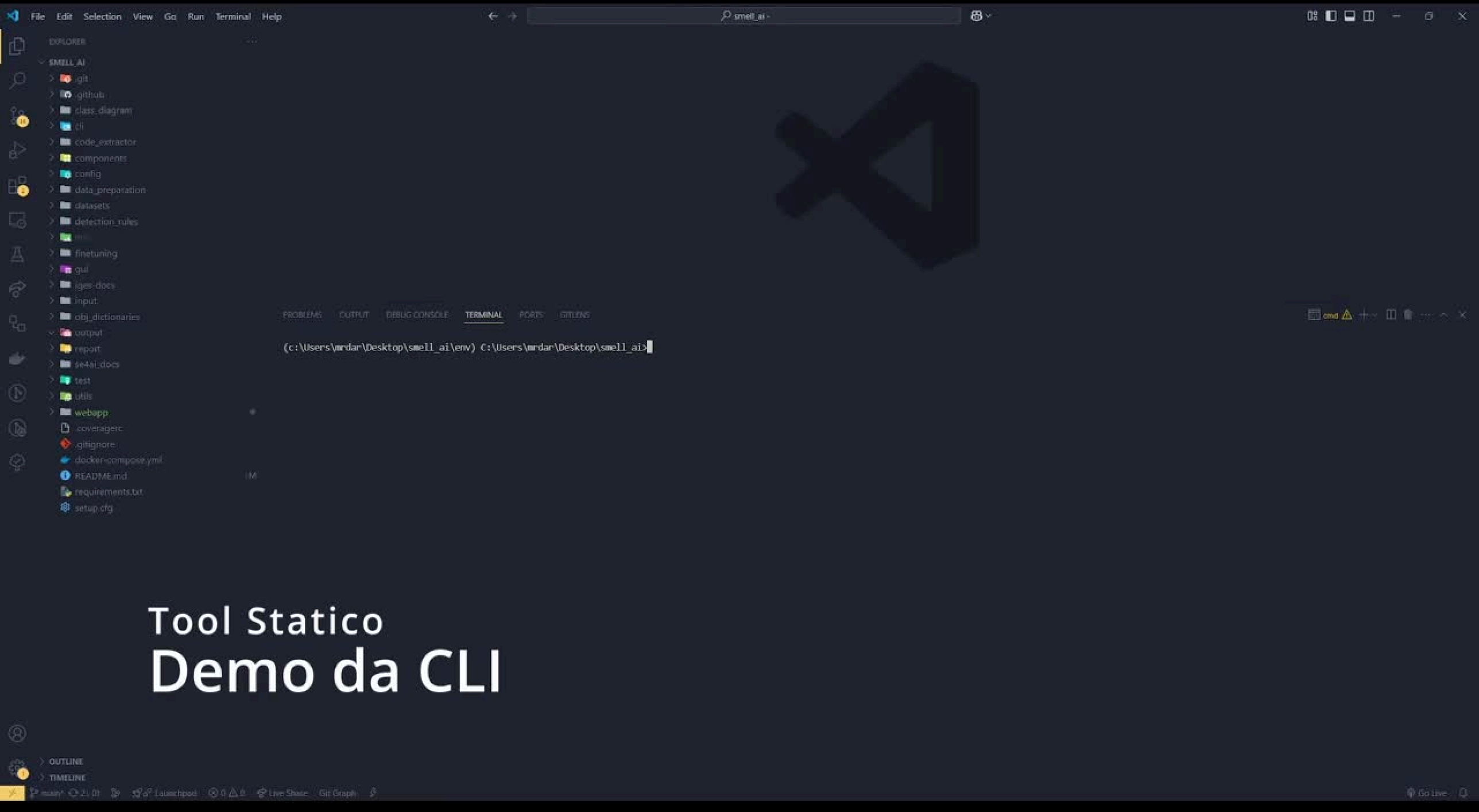
Grazie!

HTTPS://GITHUB.COM/XDARYAMO/SMELL_AI

SOURCE CODE

DOCS


CodeSmileGUI
root: Tk
gui: CodeSmellDetectorGUI
__init__()
run()
TextBoxRedirect
textbox: Text
__init__(textbox: Text)
write(text: str)
flush()
CodeSmellDetectorGUI
master: Tk
project_analyzer: ProjectAnalyzer
input_path: Label
output_path: Label
walker_picker: Spinbox
parallel_var: BooleanVar
resume_var: BooleanVar
multiple_var: BooleanVar
output_textbox: Text
__init__(master: Tk)
setup_gui()
configure_stdout()
disable_key_press(event)
choose_input_path()
choose_output_path()
run_program()
run_analysis(input_path, output_path, num_walkers, is_parallel, is_resume, is_multiple)
«static»
FileUtils
clean_directory(base_path: str, folder_name: str)
get_python_files(directory: str) : list
initialize_log(log_path: str)
get_last_logged_project(log_path: str) : str
append_to_log(log_path: str, entry: str)
synchronized_append_to_log(log_path: str, entry: str, lock: Lock)
merge_results(input_dir: str, output_dir: str)
CodeSmileCLI
args: Namespace
analyzer: ProjectAnalyzer
__init__(args: Namespace)
validate_args()
execute()
ProjectAnalyzer
base_output_path: str
output_path: str
inspector: Inspector
__init__(output_path: str)
clean_output_directory()
_save_results(df: DataFrame, filename: str)
analyze_project(project_path: str) : int
analyze_projects_sequential(base_path: str, resume: bool)
analyze_projects_parallel(base_path: str, max_workers: int)
merge_all_results()
Inspector
output_path: str
__init__(output_path: str, dataframe_dict_path: str, model_dict_path: str, tensor_dict_path: str)
inspect(filename: str): pd.DataFrame
_setup(dataframe_dict_path: str, model_dict_path: str, tensor_dict_path: str): void
RuleChecker
output_path: str
smells: list[Smell]
__init__(output_path: str)
rule_check(ast_node: ast.AST, extracted_data: dict, filename: str, function_name: str, df_output: pd.DataFrame): pd.DataFrame
_setup_smells(): void
DataFrameExtractor
df_methods: list[str]
__init__(df_dict_path: str = None)
load_dataframe_dict(path: str): void
extract_dataframe_variables(fun_node: ast.AST, alias: str): list[str]
track_dataframe_methods(fun_node: ast.AST, dataframe_vars: list[str]): dict[str, list[str]]
track_dataframe_accesses(fun_node: ast.AST, dataframe_vars: list[str]): dict[str, list[str]]
LibraryExtractor
extract_libraries(tree: ast.AST): list[dict[str, str]]
get_library_aliases(libraries: list[dict[str, str]]): dict[str, str]
get_library_of_node(node: ast.AST, aliases: dict[str, str]): str
ModelExtractor
models_path: str
tensors_path: str
model_dict: dict[str, list] | None
tensor_operations_dict: dict[str, list] | None
__init__(models_path: str, tensors_path: str)
load_model_dict(): dict[str, list]
load_tensor_operations_dict(): dict[str, list]
load_model_methods(): list[str]
check_model_method(model: str, libraries: list[str]): bool
VariableExtractor
extract_variable_definitions(fun_node: ast.AST): dict[str, ast.Assign]
track_variable_usage(fun_node: ast.AST): dict[str, list[ast.Name]]
Smell
name: str
description: str
__init__(name: str, description: str)
detect(ast_node: ast.AST, extracted_data: dict): list[dict]



Tool Statico
Demo da CLI