Initial Report

MARK 2.0 Plus

Data	18/01/2026
Presentato da	Cerchia Giovanni (NF22500202) Medica Vincenzo (NF22500203)

Sommario

1 Descrizione del tool

1.1 Contesto e motivazione

Negli studi di **Software Engineering for AI-based systems (SE4AI)** è frequente dover costruire dataset affidabili di progetti ML a partire da GitHub. Tuttavia, una selezione manuale dei repository è spesso onerosa e **soggetta a bias**, soprattutto quando la classificazione è basata su descrizioni testuali o "buzzword" presenti nei README. In tale contesto, MARK nasce come supporto per i ricercatori, offrendo un tool di classificazione automatica di progetti ML in base alla loro natura d'uso (*addestramento* vs *utilizzo* di modelli).
MARK adotta un approccio **euristico e interpretabile**: le decisioni di classificazione sono motivate da regole esplicite applicate su elementi osservabili del codice sorgente (librerie importate e API ML invocate), scelta coerente con l'esigenza di **scalabilità** su grandi dataset e con la necessità di trasparenza nel processo decisionale.

1.2 Scopo e obiettivi

MARK è un tool di classificazione automatica di repository Python che mira a distinguere progetti "applied ML" in due categorie principali:

- **ML-Model Producer**: progetti che costruiscono/addestrano modelli ML;
- **ML-Model Consumer**: progetti che utilizzano modelli ML pre-addestrati (inference).

L'obiettivo, coerentemente con la finalità di supporto alla ricerca, è privilegiare una classificazione affidabile e ad alta precisione, riducendo al minimo il rischio di includere progetti non pertinenti nel campione selezionato per analisi successive.

1.3 Funzionamento generale

MARK opera tramite **analisi statica** dei sorgenti:

- rileva librerie ML importate (es. tensorflow, scikit-learn, keras, ...);
- rileva invocazioni di API ML nei file (distinguendo training vs inference);
- applica regole euristiche per etichettare i progetti.

Il tool integra inoltre una fase di supporto alla ricerca che automatizza:
(i) la clonazione da una lista predefinita di progetti GitHub e (ii) la classificazione dei progetti clonati.

1.4 Regole di classificazione

Le regole di classificazione di MARK possono essere riassunte come segue:

1.4.1 Classificazione Producer

Un progetto è classificato come **Producer** se contiene almeno un file che:

1. importa almeno una libreria ML del knowledge base;
2. contiene almeno una chiamata a un'API di training (es. *fit*, *train*, ...).

1.4.2 Classificazione Consumer

Un progetto è classificato come **Consumer** se contiene almeno un file che:

1. importa almeno una libreria ML del knowledge base;
2. contiene almeno una chiamata a un'API di inferenza;
3. **non** contiene API di training;
4. non presenta nel filename pattern tipici di file di test/valutazione (*test*, *eval*, *example*, *validate*).

Le condizioni aggiuntive (Rule 3 e Rule 4) servono a ridurre falsi positivi, ad esempio quando API di inferenza vengono usate solo per valutare modelli addestrati nel progetto (scenario che potrebbe far apparire un Producer come Consumer).

2 Architettura di MARK 2.0 (evoluzione object-oriented)

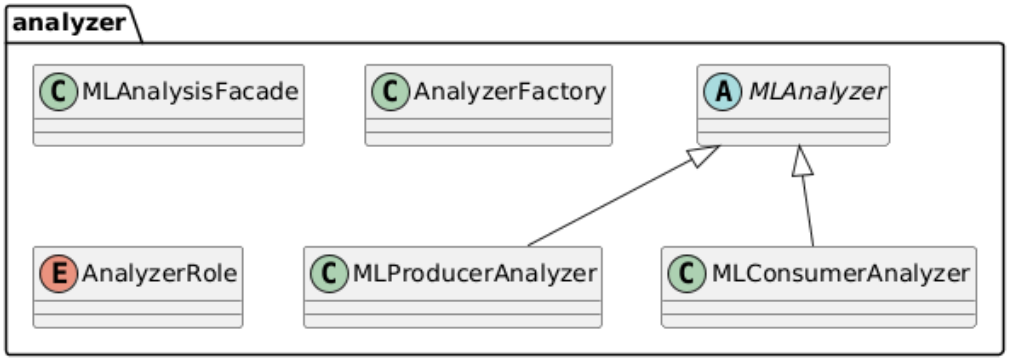
2.1 Architettura modulare

Con la versione 2.0 MARK è stato ristrutturato come un sistema object-oriented con separazione delle responsabilità in moduli "core" (cartella `modules/`):

- `analyzer/` (logica di classificazione)
- `cloner/` (acquisizione dei repository da GitHub)
- `keyword_extractor/` (estrazione delle keyword/API ML)
- `library_manager/` (gestione delle librerie Producer/Consumer da identificare)
- `scanner/` (filtraggio dei file dei progetti)
- `utils/` (logger centralizzato)
- `oracle/` (validazione dei risultati per il calcolo delle metriche di accuratezza)

2.2 Componenti principali e responsabilità

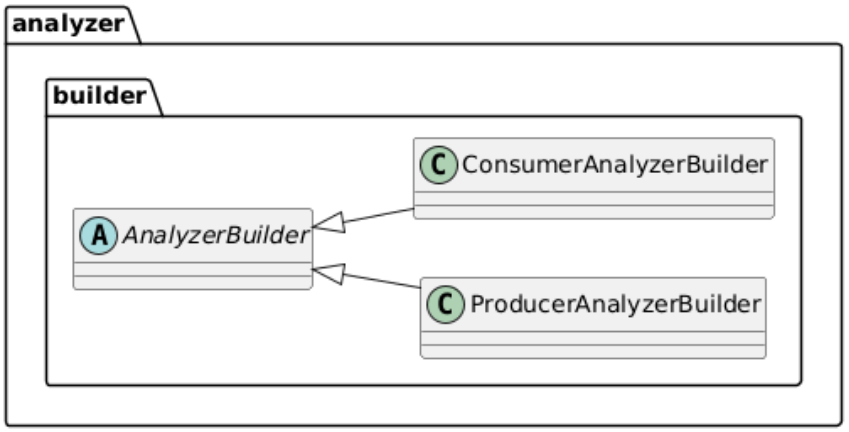
2.2.1 `modules/analyzer` — Orchestrazione dell'analisi



- `MLAnalyzer` (Abstract): Classe base astratta per tutti gli analyzer. Definisce il workflow comune per scansionare i progetti, applicare i filtri configurati, estrarre librerie/keyword ed esportare i risultati in file CSV. Fornisce i metodi template per l'analisi che le classi concrete specializzano.
- `MLProducerAnalyzer`: Specializzazione per rilevare progetti che costruiscono/addestrano modelli ML. Applica i filtri di estensione e le strategie di keyword matching per identificare l'utilizzo di API di training e salvataggio modelli.

- `MLConsumerAnalyzer`: Specializzazione per identificare progetti che utilizzano modelli ML pre-addestrati. Applica regole consumer sui file sorgente, con supporto opzionale della Rule 3 (esclusione progetti con API di training tramite parametro `rules_3`).
- `MLAnalysisFacade`: Implementa il Facade pattern che nasconde la complessità del workflow dietro un'interfaccia semplificata `run_analysis(kwargs)`. Coordina l'intera pipeline di analisi delegando alla factory la creazione degli analyzer e gestendo l'esecuzione end-to-end.
- `AnalyzerFactory`: Implementa un pattern Factory-Registry che permette la registrazione dinamica dei builder tramite decorator `@register(role)` e il metodo `create_builder(role)` per l'istanziazione. Mantiene un registro interno delle classi builder associate ai vari ruoli di analisi.
- `AnalyzerRole` (Enum): Enumerazione che definisce i ruoli disponibili per gli analyzer ML: `PRODUCER` (costruttori di modelli) e `CONSUMER` (utilizzatori di modelli).

2.2.2 `modules/analyzer/builder` — Costruzione configurata degli analyzer

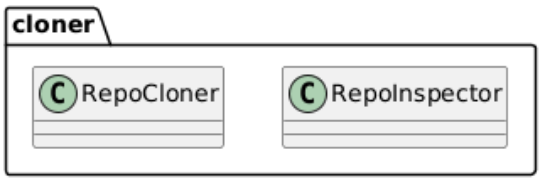


- `AnalyzerBuilder` (Abstract): Classe base astratta che implementa il Builder pattern per assemblare istanze di `MLAnalyzer` con configurazioni consistenti. Raccoglie step-by-step tutte le dipendenze richieste (role, filtri, strategie, dizionari) e produce un analyzer completo solo quando la configurazione è valida. Centralizza la logica di setup eliminando codice di inizializzazione duplicato e oggetti parzialmente configurati.
- `ConsumerAnalyzerBuilder`: Builder concreto pre-configurato per istanziare `MLConsumerAnalyzer`. Registrato automaticamente tramite

`@register(AnalyzerRole.CONSUMER)`, configura di default i filtri `.py` e `ExcludeTestFilesFilter`, i dizionari consumer/producer e la strategia `DefaultKeywordMatcher`.

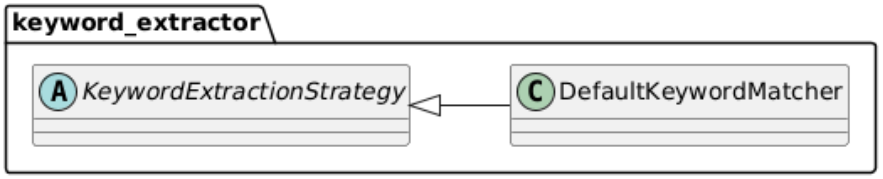
- `ProducerAnalyzerBuilder`: Builder concreto pre-configurato per istanziare `MLProducerAnalyzer`. Registrato tramite `@register(AnalyzerRole.PRODUCER)`, applica il filtro `.py`, il dizionario producer e la strategia `DefaultKeywordMatcher` per rilevare API di training.

2.2.3 `modules/cloner` — Acquisizione dei progetti



- `RepoCloner`: Gestisce la clonazione di massa dei repository GitHub tramite shallow clone paralleli con `ThreadPoolExecutor`. Registra successi ed errori su file CSV (`cloned_log.csv`, `errors.csv`) per tracciare gli esiti del processo di acquisizione.
- `RepoInspector`: Analizza i risultati del processo di clonazione verificando quali repository sono stati effettivamente clonati. Confronta il file CSV di log con il filesystem e genera report di riconciliazione (`not_cloned_repos.csv`, `effective_repos.csv`).

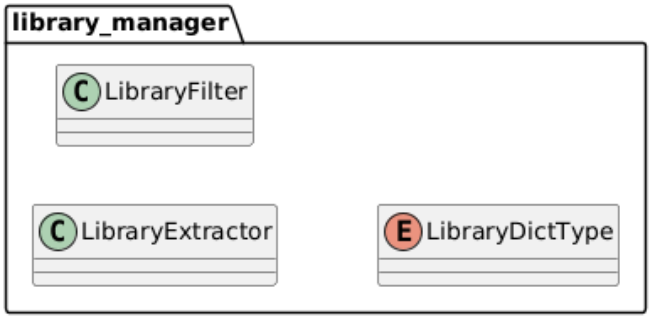
2.2.4 `modules/keyword_extractor` — Estrazione keyword/API



- `KeywordExtractionStrategy` (Abstract): Classe base astratta che definisce il contratto per l'estrazione di keyword dai file sorgente. Implementa il Strategy pattern permettendo di sostituire dinamicamente l'algoritmo di matching senza modificare le classi analyzer.
- `DefaultKeywordMatcher`: Implementazione concreta dell'estrazione keyword mediante scansione line-by-line con pattern regex. Costruisce espressioni regolari case-insensitive gestendo caratteri speciali (punti,

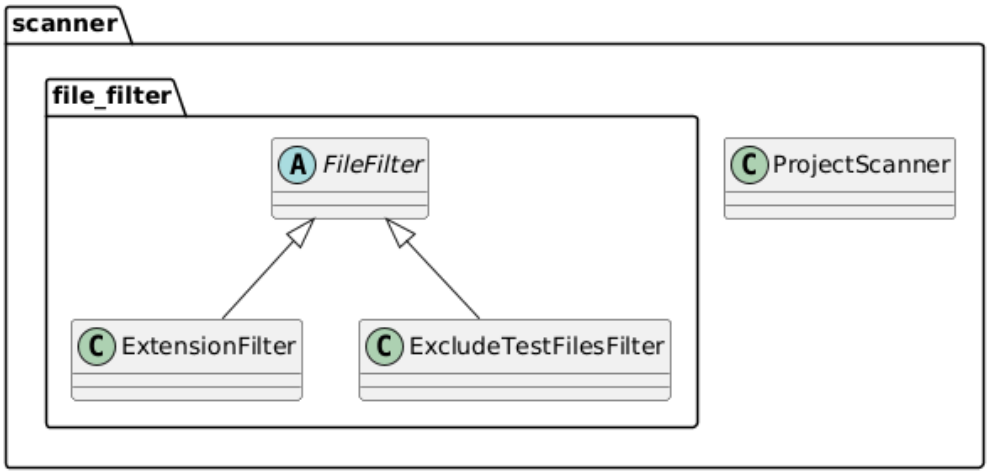
parentesi) e whitespace opzionale, restituendo le keyword trovate con il numero di riga corrispondente.

2.2.5 `modules/library_manager` — Dizionari e import ML



- `LibraryExtractor`: Utility class per identificare gli statement di import nei file sorgente Python. Esegue matching testuale line-by-line su pattern `import` e `from...import`, estraendo i nomi delle librerie utilizzate nel codice.
- `LibraryFilter`: Fornisce metodi per caricare i dizionari di librerie ML da file CSV e filtrare quelle effettivamente utilizzate nei progetti. Confronta le librerie importate con quelle presenti nei dizionari consumer/producer per classificare i progetti.
- `LibraryDictType` (Enum): Enumerazione che identifica i tipi di dizionario disponibili (`CONSUMER`, `PRODUCER`) per categorizzare le librerie ML in base al loro utilizzo tipico (training vs inference).

2.2.6 `modules/scanner` - Filtraggio e scansione file

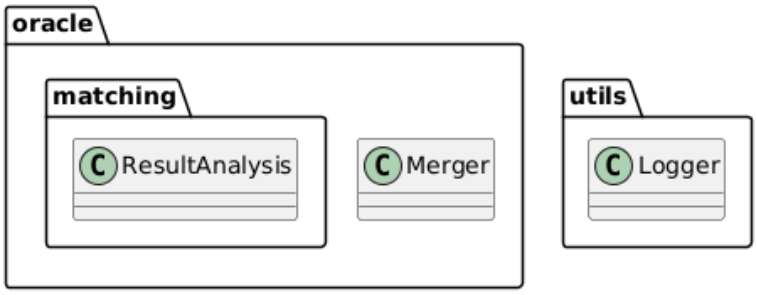


- `FileFilter` (Abstract): Classe base astratta per implementare filtri di selezione dei file durante la scansione. Definisce il metodo

`accepts(file_path)` che le classi concrete specializzano per applicare criteri specifici.

- `ExtensionFilter`: Filtro che accetta esclusivamente file con estensioni specifiche (es. `.py`, `.ipynb`). Permette di limitare l'analisi a determinati tipi di file sorgente.
- `ExcludeTestFilesFilter`: Filtro che scarta file comunemente associati a testing, esempi, valutazioni o validazioni basandosi su pattern nel nome file (es. `test`, `example`, `eval`, `validat`). Implementa la Rule 4 dei Consumer per escludere codice non rappresentativo dell'utilizzo reale.
- `ProjectScanner`: Coordina la scansione ricorsiva delle cartelle di progetto per raccogliere i file sorgente che soddisfano tutti i filtri configurati. Applica in sequenza la catena di filtri e restituisce solo i file che superano tutti i controlli.

2.2.7 `modules/oracle` e `modules/utils` – Validazione e logging

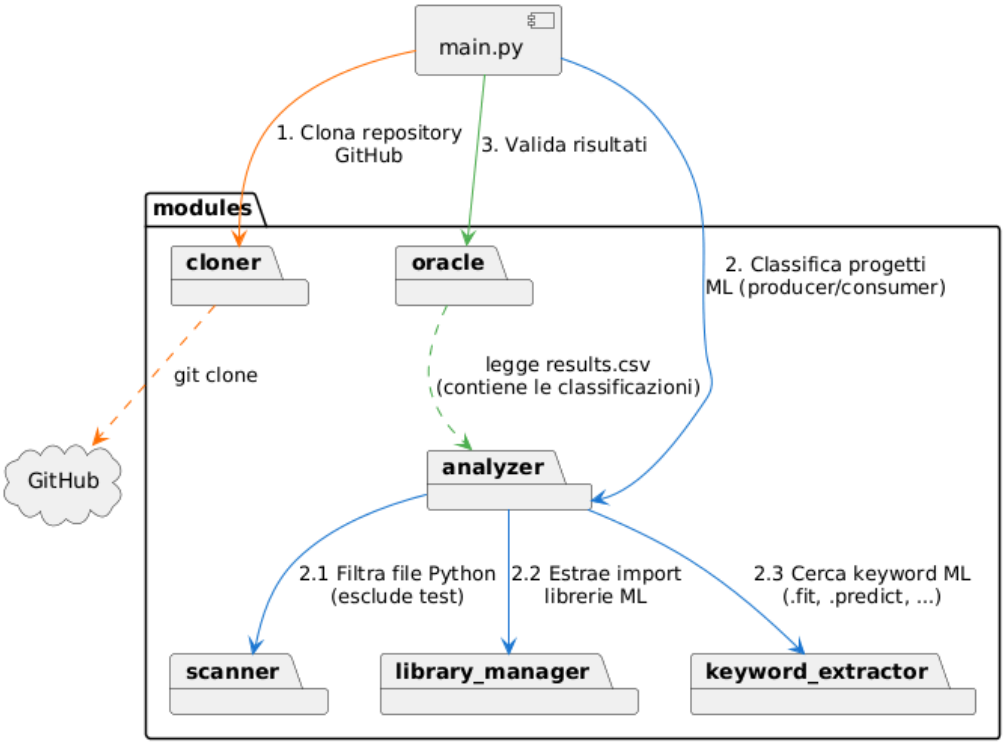


- `Merger`: Compara i risultati dell'analisi con un file oracolo di ground truth e calcola le metriche di valutazione (Precision, Recall, F1-Score, Accuracy). Esporta i false positive e false negative in file CSV separati per l'analisi degli errori.
- `ResultAnalysis`: Confronta i risultati ottenuti dai singoli progetti analizzati con il file oracolo di riferimento. Aggrega le metriche a livello di progetto ed esporta un report CSV con tutti i valori di performance per ogni repository.
- `get_logger()`: Funzione factory che crea e restituisce un'istanza configurata di logger Python. Il logger viene utilizzato trasversalmente da tutti i moduli per garantire tracciabilità uniforme delle operazioni, errori e performance del sistema.

3 Flusso principale di utilizzo

Nello stato iniziale, MARK 2.0 è eseguibile tramite una pipeline avviata dallo script `main.py`. Una prima **limitazione** notata è che i parametri di configurazione della pipeline risultano impostati direttamente nello script: l'approccio di utilizzo offerto è quindi principalmente orientato ad utenti tecnici.

3.1 Sequenza delle operazioni



Step 1 — Clonazione dei repository

`RepoCloner`:

1. legge un CSV con la lista dei progetti GitHub;
2. clona in parallelo con `ThreadPoolExecutor`;
3. genera la struttura `io/repos/<user>/<project>/...`;
4. `RepoInspector` verifica la consistenza del file CSV con il filesystem.

Step 2 — Analisi ML (classificazione)

`MLAnalysisFacade` coordina un `AnalyzerFactory` per costruire un `MLAnalyzer` specializzato; per ogni progetto:

1. filtra file (es. `.py`) con `ProjectScanner`;
2. estrae e filtra librerie con `LibraryExtractor` e `LibraryFilter`;
3. trova keyword/API con `KeywordMatcher`;

4. produce `results.csv` con una riga per evidenza rilevata (progetto, libreria, file, keyword/API, linea).

Esempio di struttura di un file CSV:

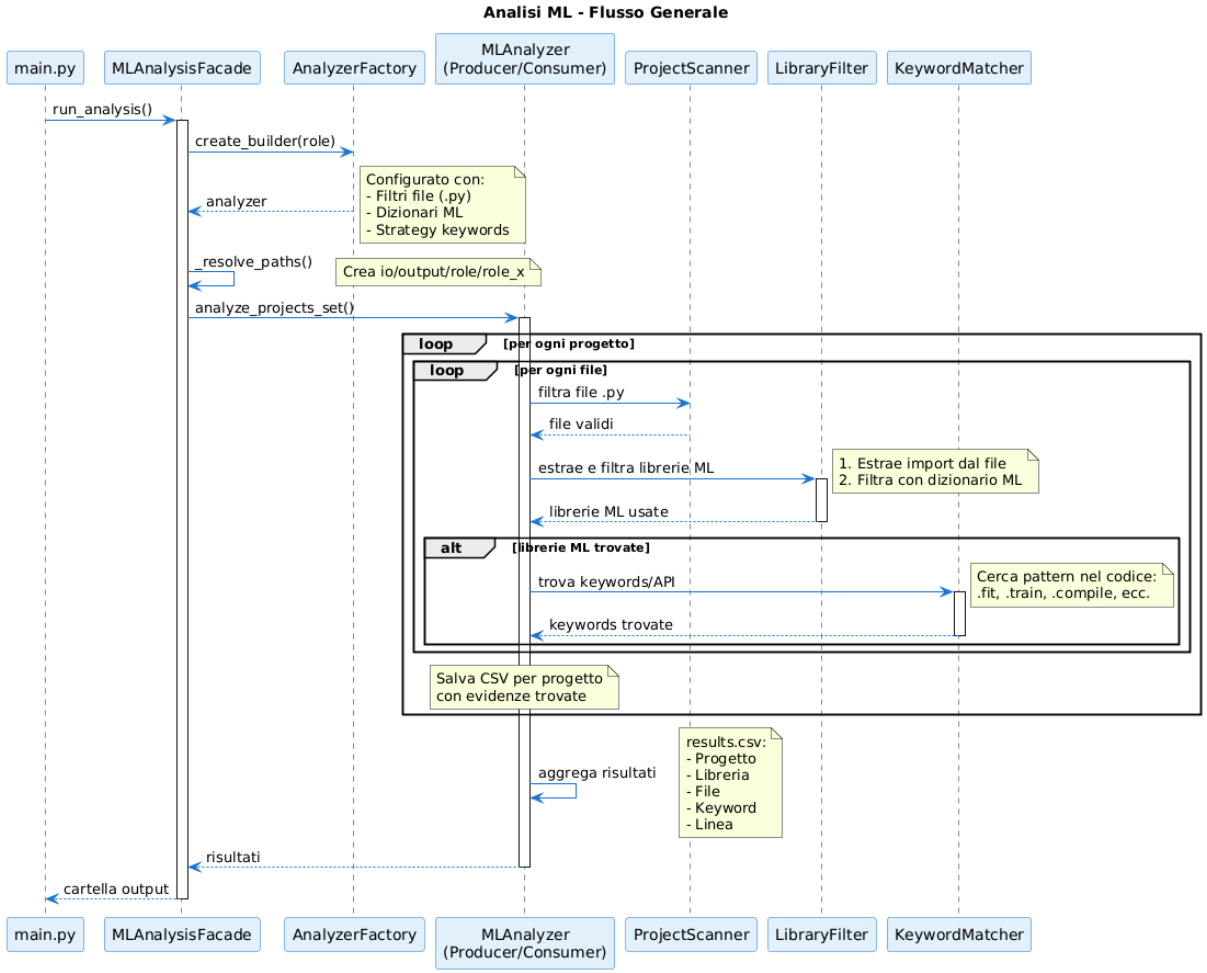
ProjectName	Is ML producer	libraries	where	keyword	line_number
user/repo-A	Yes	tensorflow	/model.py	.fit	45

Step 3 — Validazione con oracolo (se attivato)

`Merger`:

1. calcola TP/FP/TN/FN e metriche di classificazione;
2. genera `false_positives.csv` e `false_negatives.csv`.

Sequence diagram per lo Step 2 — Analisi ML



4 Input e Output

4.1 Input principali

- CSV contenente la lista di repository GitHub (se si decide di utilizzare il cloner).
- Cartella contenenti i progetti da analizzare (se si decide di eseguire i soli step di analisi)
- Knowledge base (dizionari in formato .csv) di librerie ML e API (training/inference), utilizzata per filtrare import e chiamate API.

4.2 Output principali

- Output cloner: `cloned_log.csv`, `errors.csv`, `not_cloned_repos.csv`, `effective_repos.csv`.
- Output analisi: `user_repo_ml_producer/consumer.csv` per ogni singolo progetto correttamente classificato e `results.csv` con i risultati uniti (schema riportato sopra).
- Output validazione: metriche (Precision/Recall/F1/Accuracy), `false_positives.csv`, `false_negatives.csv`.

5 Conclusioni

Il presente documento descrive lo **stato iniziale** di MARK 2.0, focalizzato su:

- acquisizione automatizzata dei repository,
- classificazione euristica Producer/Consumer tramite analisi statica,
- eventuale validazione con oracolo e metriche di accuratezza.

Le **Change Requests** sono documentate separatamente nel documento dedicato.