Report ISW2

Lorenzo Grande 0350212

1 abstract

Questo report analizza la relazione tra metriche del codice e propensione ai difetti a livello di metodo, con un'analisi empirica su due progetti open source, Apache BookKeeper e Apache OpenJPA. L'approccio combina metriche del codice con modelli di machine learning, valutati tramite protocollo walk-forward. I risultati evidenziano che le feature maggiormente correlate alla buggyness sono le linee di codice aggiunte in un commit (*LocAdded*) e il *Fan-out*; Inoltre, la rimozione degli smell dai metodi buggy riduce la probabilità di difetto fino al 14%. Questi risultati suggeriscono che un refactoring mirato sui metodi più problematici può migliorare sensibilmente la qualità del software, fornendo indicazioni pratiche per la manutenzione e l'evoluzione dei progetti.

2 Introduzione

Contesto. La manutenzione del software è un'attività complessa e costosa. Quando la manutenibilità diminuisce, cresce la probabilità di introdurre difetti. Progetti open source come Apache BookKeeper e Apache OpenJPA, soggetti a frequenti rilasci, sono particolarmente esposti a tale rischio [Zimmermann and Nagappan(2007)]. La letteratura ha mostrato come diverse proprietà del codice, misurabili tramite *metriche statiche*, siano correlate alla fault-proneness [Khomh et al.(2012)Khomh, Penta, and Guéhéneuc, Rahman et al.(2023)Rahman, Sakib, Rashid, and Karim]. Rimane però aperta la questione di come sfruttare tali correlazioni per guidare attività di manutenzione e refactoring.

Problema e lacune. Gli studi si sono concentrati soprattutto su metriche a livello di classe o di modulo [Menzies et al.(2007)Menzies, Greenwald, and Frank], mentre il livello di metodo è meno esplorato. Inoltre, mancano evidenze empiriche sull'impatto della riduzione delle metriche più influenti nel ridurre i difetti.

Obiettivi e domande di ricerca. Questo lavoro analizza BookKeeper e OpenJPA costruendo dataset storici a livello di metodo e valutando diversi classificatori nella predizione della buggyness. In particolare, gli obiettivi sono:

- identificare le feature più influenti nella predizione della buggyness;
- valutare quale classificatore di machine learning predice meglio i metodi difettosi;
- stimare quanti difetti si sarebbero potuti evitare riducendo la feature più correlata.

Le domande di ricerca sono:

RQ1 Quali sono le features maggiormente correlate alla buggyness?

RQ2 Quale classificatore predice meglio la buggyness dei metodi?

RQ3 Quanti metodi buggy si sarebbero potuti evitare riducendo la feature più correlata?

Struttura del paper. La Sezione 3 descrive il dataset, le metriche e il protocollo sperimentale; la Sezione 4 presenta i risultati e risponde alle RQ; la Sezione 5 discute i limiti e le minacce alla validità; la Sezione 6 conclude e propone sviluppi futuri.

Link utili

- Codice sorgente su GitHub: Lollogiga/ISW2-Project
- Analisi di qualità su SonarCloud: SonarCloud

3 Measurement & Methodology

3.1 Dataset & Label Extraction

L'analisi è stata condotta su due progetti open source di larga scala: **Apache BookKeeper** e **Apache OpenJPA**, considerando più release e producendo dataset storici a livello di metodo. I dati sono stati raccolti integrando `Jira` (ticket di bug *resolved/fixed*) e `GitHub` (commit collegati), per identificare le classi e i metodi modificati nei bug fix.

Ogni metodo è inizialmente etichettato come *non buggy* e diventa *buggy* se modificato in almeno un commit di fix. Per stimare la *injected version* (IV), quando le versioni affette non erano note, è stata adottata la tecnica della **Proportion** [Vandehei et al.(2021)Vandehei, Costa, and Falessi], che assume una proporzionalità tra le release comprese fra IV e FV e quelle fra OV e FV:

$$P = \frac{FV - IV}{FV - OV} \quad \Longrightarrow \quad IV = FV - (FV - OV) \times P$$

Sono state implementate due varianti: *Incremental*, che stima P come media delle proporzioni osservate nei difetti precedenti, e *ColdStart*, usata quando i dati storici non erano sufficienti. Questo approccio consente di etichettare anche i metodi privi di informazioni esplicite nelle AV, aumentando la completezza del dataset.

Un ulteriore problema è rappresentato dallo snoring: un difetto può essere presente ma non rilevato fino alla sua correzione, generando errori di labelling. Per ridurne l'impatto, solo il primo 34% delle release di ciascun progetto è stato usato per costruire il dataset, mentre il labelling è stato effettuato considerando tutte le release. Il dataset finale contiene, per ogni coppia `(metodo, release)`, le metriche calcolate e la label binaria `isBuggy`.

3.2 Metrics & Definitions

Sono stati considerati due insiemi di metriche.

Metriche di valutazione dei classificatori. Per misurare l'efficacia predittiva dei modelli di machine learning sono state usate metriche standard:

- `avg_precision`, `avg_recall`, `avg_f1`, `avg_accuracy`, `avg_auc` — medie sulle diverse iterazioni;
- `avg_kappa` — Cohen's Kappa, misura di quanto il classificatore migliora rispetto a un predittore casuale;
- `sum_tp`, `sum_fp`, `sum_tn`, `sum_fn` — conteggi aggregati di true/false positives/negatives;
- `avg_pofb20` — metrica *effort-aware* che misura i difetti individuati analizzando il primo 20% della code base ordinata per probabilità di buggyness.

Metriche statiche e di processo. Per ciascun metodo e release sono state calcolate:

- `LOC` — linee di codice del metodo;
- `CyclomaticComplexity` — numero di percorsi indipendenti nel grafo di controllo, proxy di complessità logica;
- `Churn` — linee modificate nel metodo durante la release;
- `LocAdded` — linee aggiunte nella release;
- `Fan-in/out` — numero di metodi che invocano o sono invocati dal metodo;
- `NewcomerRisk` — presenza di sviluppatori alla prima contribuzione;
- `Auth` — numero di autori che hanno modificato il metodo;
- `WeekendCommit` — presenza di commit effettuati nel weekend;
- `nSmell` — numero di code smells individuati nel metodo.

La variabile target `isBuggy` vale `Yes` se il metodo è stato etichettato come difettoso, `No` altrimenti.

Correlazione. La relazione tra ciascuna metrica e la buggyness è stata stimata tramite il coefficiente di Spearman (ρ), che misura l'associazione monotona tra due variabili, con valori tra -1 (correlazione negativa perfetta) e $+1$ (positiva perfetta).

3.3 Experimental Protocol

L'esperimento ha seguito un protocollo *walk-forward* [Kim et al.(2008)Kim, Zimmermann, Jr., and Zeller], adatto a dati sequenziali come le release software: i classificatori sono stati addestrati su release storiche e testati sulla successiva. Sono stati confrontati tre modelli comunemente impiegati in letteratura (Naive Bayes, Random Forest, k-Nearest Neighbors), valutati sia in versione standard sia con tecniche di feature selection.

3.4 Assumptions / Shortcuts

Per garantire riproducibilità e semplicità, l'analisi è stata limitata a tre classificatori con parametri di default, a refactoring manuale di un singolo metodo target e all'uso del 34% delle release per mitigare lo *snoring*. Queste scelte possono introdurre bias che limitano la validità esterna dei risultati, come discusso in Sezione 5.

4 Risultati

Nella seguente sezione andremo ad analizzare i risultati ottenuti rispondendo alle tre domande RQ1, RQ2, RQ3.

4.1 Rationale for Selecting the Target Method

Per rispondere alla domanda RQ1 è stato calcolato il coefficiente di correlazione di Spearman (ρ) tra ciascuna metrica e l'etichetta *isBuggy*. L'analisi della Tabella 2 evidenzia che:

- In entrambi i progetti la metrica più correlata con la buggyness è *LocAdded*;
- Tra le actionable features, quella maggiormente correlata con la buggyness è *Fan-out*.

Sulla base di questo risultato, per ciascun progetto è stato scelto, nell'ultima release disponibile, il metodo con il valore più elevato della *actionable feature* identificata (di seguito indicato come *AFMethod*), sottoposto a un intervento di refactoring al fine di verificarne l'impatto sulla manutenibilità

4.2 Description of the Selected Method

AFMethod-A

- **Firma:** *bookkeeper-server/src/main/java/org/apache/bookkeeper/bookie/Journal.java::run*
- **Responsabilità:** Il metodo run() gestisce scritture sequenziali, flush e rotazioni periodiche dei log.

AFMethod-B

- **Firma:** *openjpa-kernel/src/main/java/org/apache/openjpa/kernel/jpql/JPQLExpressionBuilder.java::eval*
- **Responsabilità:** Il metodo eval(JPQLNode node) interpreta ricorsivamente nodi JPQL e li traduce in espressioni.

Le metriche pre-refactoring per i due metodi sono riportate in Tabella 3a e Tabella 3b.

4.3 Criteri per l'Identificazione degli Obiettivi di Refactoring

La scelta delle azioni di refactoring è stata guidata dai seguenti criteri:

- **Priorità 1:** preservare il comportamento funzionale e la copertura dei test.
- **Priorità 2:** ridurre la metrica principale (fan-out) senza aumentare/diminuire le metriche correlate positivamente/negativamente con la buggyness .

4.4 Resulting Refactored Method.

In entrambi i metodi sono stati applicati refactoring standard (Facade, Extract Method, Inline Temp/Replace Duplicated Code) per ridurre il fan-out. I metodi risultanti fungono da orchestratori leggeri che delegano a pochi collaboratori principali.

4.4.1 Analisi dei risultati

Nelle tabelle 4a e 4b, possiamo analizzare le variazioni in percentuali delle metriche a seguito del refactoring. Ovviamente è stato inevitabile un incremento di Churn e LocAdded, due metriche note per avere correlazione positiva con la buggyness. Questo non significa che la manutenibilità del metodo sia peggiorata, perché tali aumenti sono una conseguenza diretta della modifica del codice. L'obiettivo del refactoring è invece quello di ridurre le metriche che aumentano la complessità e che sono fortemente correlate con la buggyness del codice nel lungo periodo. In questo senso, i risultati mostrano un miglioramento:

- LOC è diminuita del 50% per AFMethod-A e del 29% per AFMethod-A;
- Fan-out è stata ridotto a poco più del 70% per entrambi i progetti;
- nSmell è stato completamente eliminato nel caso di AFMethod-A.

Questi cambiamenti vanno nella direzione di una maggiore manutenibilità, poiché riducono il rischio che il metodo diventi un punto fragile del sistema.

4.5 Analisi What-If sui Metodi Buggy Evitabili

Dai risultati della sottosezione .3, il miglior classificatore (**BClassifier**) è risultato essere la **Random Forest**, nella configurazione senza feature selection per **BookKeeper** e con feature selection *BestFirst* per **OpenJPA**. Con questo modello è stata condotta un'analisi *what-if* per stimare quanti metodi buggy si sarebbero potuti evitare se il numero di smell fosse stato pari a zero.

Procedura. Dal dataset originale (A) sono stati derivati:

- B^+ (metodi con `nSmells` > 0),
- C (metodi con `nSmells` $= 0$)
- B, ottenuto da B^+ ponendo `nSmells` $= 0$.

Il BClassifier è stato addestrato su A e applicato a A, B^+, B e C.

Riduzione attesa. Dal confronto tra B^+ e B stimiamo il numero di metodi che non sarebbero più predetti buggy dopo la rimozione degli smells. Nelle tabelle 5 e 6 sono riportati i risultati delle analisi per i due progetti. In particolare troviamo:

Tabella 1: Risultati sintetici dell'analisi What-If.

BookKeeper

Metrica	Valore
Bug evitabili (totale)	158
Proporzione su tutti i buggy	14.2%
Proporzione su buggy con smells	46.6%

OpenJPA

Metrica	Valore
Bug evitabili (totale)	442
Proporzione su tutti i buggy	4.42%
Proporzione su buggy con smells	14.46%

Assunzioni. L'analisi What-If si basa sulle seguenti ipotesi:

- l'ipotetico refactoring modifica unicamente la feature `nSmells`, mantenendo invariati tutti gli altri attributi;
- le probabilità fornite dal classificatore restano valide anche nello scenario controfattuale (assenza di smells);

5 Discussion & Threats

5.1 Discussion

I risultati ottenuti permettono di rispondere alle **RQ** definite in precedenza. Per entrambi i progetti, il classificatore migliore è risultato essere la Random Forest, sebbene in configurazioni diverse (senza feature selection per BookKeeper e con BestFirst per OpenJPA). L'analisi What-If ha mostrato che la rimozione degli smells può avere un impatto concreto: per BookKeeper circa il 14% dei metodi buggy sarebbe stato evitabile, mentre per OpenJPA il valore stimato è intorno al 4%. Queste differenze suggeriscono che la correlazione tra smells e difettosità non è universale, ma dipende dal contesto progettuale e dalle pratiche di sviluppo. Dal punto di vista della teoria della manutenibilità, i risultati confermano che la riduzione degli smells è associata a una minore propensione ai difetti, supportando l'idea che il debito tecnico si rifletta nella qualità evolutiva del software. Tuttavia, l'impatto varia e non sempre è dominante rispetto ad altre metriche.

5.2 Threats to Validity

Come in ogni studio empirico, i risultati vanno interpretati alla luce di possibili minacce:

- **Internal validity**: gli errori di misurazione (es. conteggio smells, labeling dei bug) possono introdurre rumore nei dati; inoltre, la scelta di simulare il refactoring agendo solo sulla feature `nSmells` potrebbe semplificare eccessivamente la realtà.
- **External validity**: i risultati si basano su due soli progetti (BookKeeper e OpenJPA) e potrebbero non generalizzare ad altri contesti, linguaggi o dimensioni progettuali.

6 Conclusion & Future Work

In questo lavoro abbiamo valutato l'impatto dei code smells sulla predizione della difettosità dei metodi, confrontando diversi classificatori e conducendo un'analisi controfattuale What-If. Il risultato principale è che la rimozione degli smells può ridurre in maniera significativa la propensione ai bug (con un impatto maggiore in BookKeeper rispetto a OpenJPA), confermando l'associazione tra debito tecnico e qualità del software. L'implicazione è che la gestione attiva degli smells non è solo una questione estetica o stilistica, ma può tradursi in una concreta riduzione di difetti nel ciclo di vita del software.

Prospettive future.

- Replicare l'analisi su un numero più ampio di progetti, per aumentare la generalizzabilità.
- Integrare tecniche di refactoring automatizzato, per validare l'approccio in scenari reali.
- Condurre studi longitudinali su release future, per misurare l'effetto degli smells sulla manutenzione a lungo termine.

Riferimenti bibliografici

[Khomh et al.(2012)Khomh, Penta, and Guéhéneuc] Foutse Khomh, Massimiliano Di Penta, and Yann-Gaël Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 147–156. IEEE, 2012. doi: 10.1109/CSMR.2012.30. URL https://doi.org/10.1109/CSMR.2012.30.

[Kim et al.(2008)Kim, Zimmermann, Jr., and Zeller] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 489–498. ACM, 2008. doi: 10.1145/1368088.1368157. URL https://doi.org/10.1145/1368088.1368157.

[Menzies et al.(2007)Menzies, Greenwald, and Frank] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007. doi: 10.1109/TSE.2007.256941. URL https://doi.org/10.1109/TSE.2007.256941.

[Rahman et al.(2023)Rahman, Sakib, Rashid, and Karim] Mahmudur Rahman, Kazi Sakib, Nafiul Rashid, and Md. Rezaul Karim. Do code smells induce faults? an empirical study on the relation between code smells and fault-proneness. *arXiv preprint*, 2023. URL https://arxiv.org/abs/2305.05572.

[Vandehei et al.(2021)Vandehei, Costa, and Falessi] Bailey Vandehei, Daniel Alencar Da Costa, and Davide Falessi. Leveraging the defects life cycle to label affected versions and defective classes. *ACM Transactions on Software Engineering and Methodology*, 30(2), 2021. ISSN 1049-331X. doi: 10.1145/3433928. URL https://doi.org/10.1145/3433928.

[Zimmermann and Nagappan(2007)] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects from history. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 1–10. IEEE, 2007. doi: 10.1109/ICSE.2007.39. URL https://doi.org/10.1109/ICSE.2007.39.

Appendix

.1 Correlazione fra metriche e target

Tabella 2: Correlazione (ρ di Spearman) tra features e buggyness

BookKeeper

Metrica	ρ
LocAdded	0.195545
Churn	0.195509
Fan-out	0.130310
Auth	0.129484
LOC	0.128187
CyclomaticComplexity	0.099032
nSmell	0,065228
WeekendCommit	0,041979
NewcomerRisk	0,039242
Fan-in	0,007244

OpenJPA

Metrica	ρ
LocAdded	0,109646
Auth	0,100183
NewcomerRisk	0,098151
Fan-out	0,098126
Churn	0,092386
LOC	0,085719
CyclomaticComplexity	0,065383
nSmell	0,044516
Fan-in	0,029737
WeekendCommit	0,021499

.2 Refactoring di AFMethod

Tabella 3: Snapshot delle metriche prima del refactoring

(a) AFMethod-A

Metrica	**Valore**
LOC	80
CyclomaticComplexity	10
Churn	3
LocAdded	3
Fan-in	0
Fan-out	21
NewcomerRisk	0
Auth	1
WeekendCommit	0
nSmell	1

(b) AFMethod-B

Metrica	**Valore**
LOC	452
CyclomaticComplexity	112
Churn	13
LocAdded	8
Fan-in	0
Fan-out	93
NewcomerRisk	1
Auth	2
WeekendCommit	0
nSmell	1

Tabella 4: Snapshot delle metriche dopo il refactoring

(a) AFMethod-A

Metrica	**Valore**	**Δ (%)**
LOC	40	−50.0
CyclomaticComplexity	10	0.0
Churn	104	+3366.7
LocAdded	40	+1233.3
Fan-in	0	0.0
Fan-out	6	−71.4
NewcomerRisk	1	n/a
Auth	2	+100.0
WeekendCommit	0	0.0
nSmell	0	−100.0

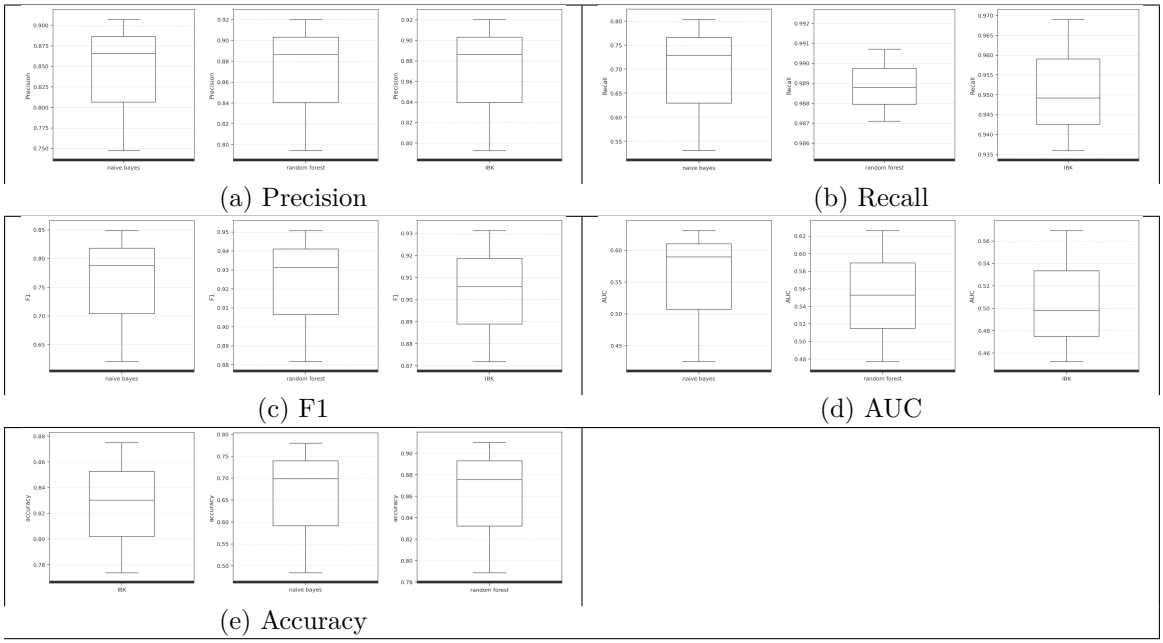
(b) AFMethod-B

Metrica	**Valore**	**Δ (%)**
LOC	320	−29.2
CyclomaticComplexity	112	0.0
Churn	240	+1746.2
LocAdded	60	+650.0
Fan-in	0	0.0
Fan-out	25	−73.1
NewcomerRisk	1	0.0
Auth	3	+50.0
WeekendCommit	0	0.0
nSmell	1	0.0

.3 Risultati classificatori

Classificatori senza feature selection

BookKeeper



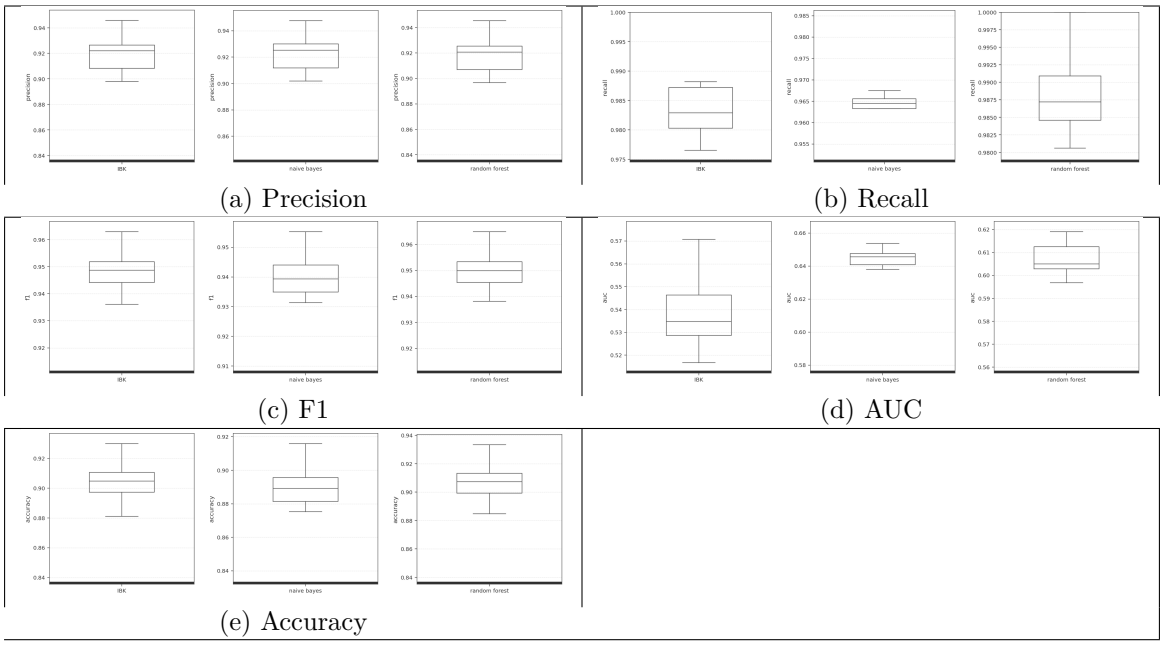
(a) Precision (b) Recall

(c) F1 (d) AUC

(e) Accuracy

Figura 1: BookKeeper: classificatori senza feature selection

OpenJPA



(a) Precision (b) Recall

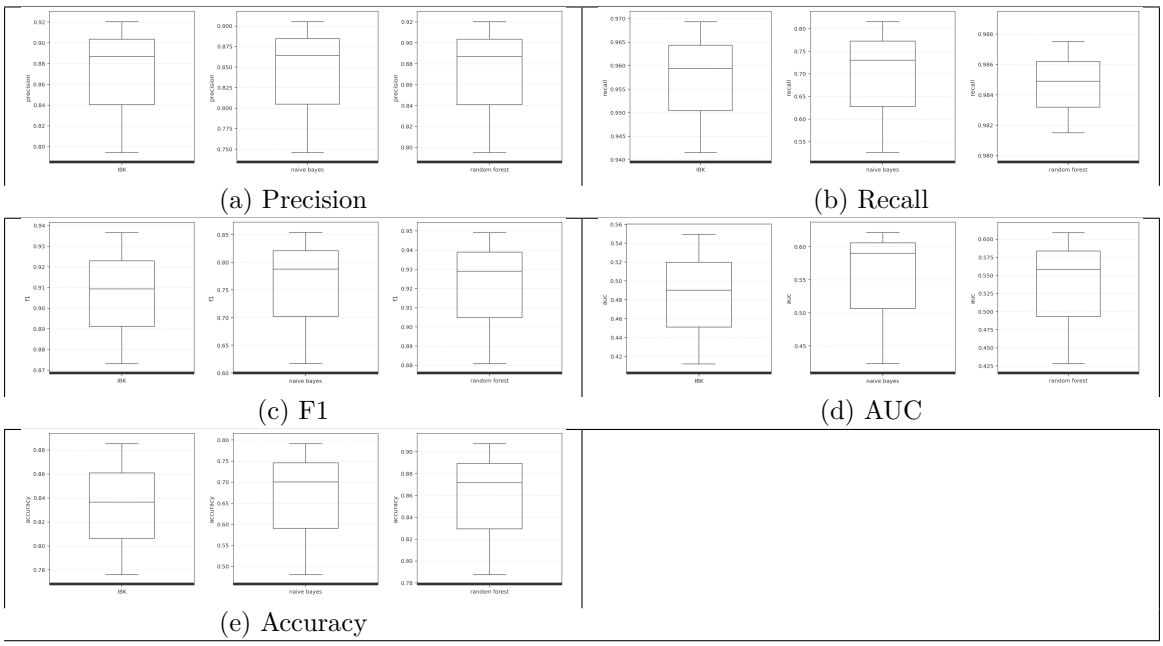
(c) F1 (d) AUC

(e) Accuracy

Figura 2: OpenJPA: classificatori senza feature selection

Classificatori con feature selection *BestFirst*

BookKeeper



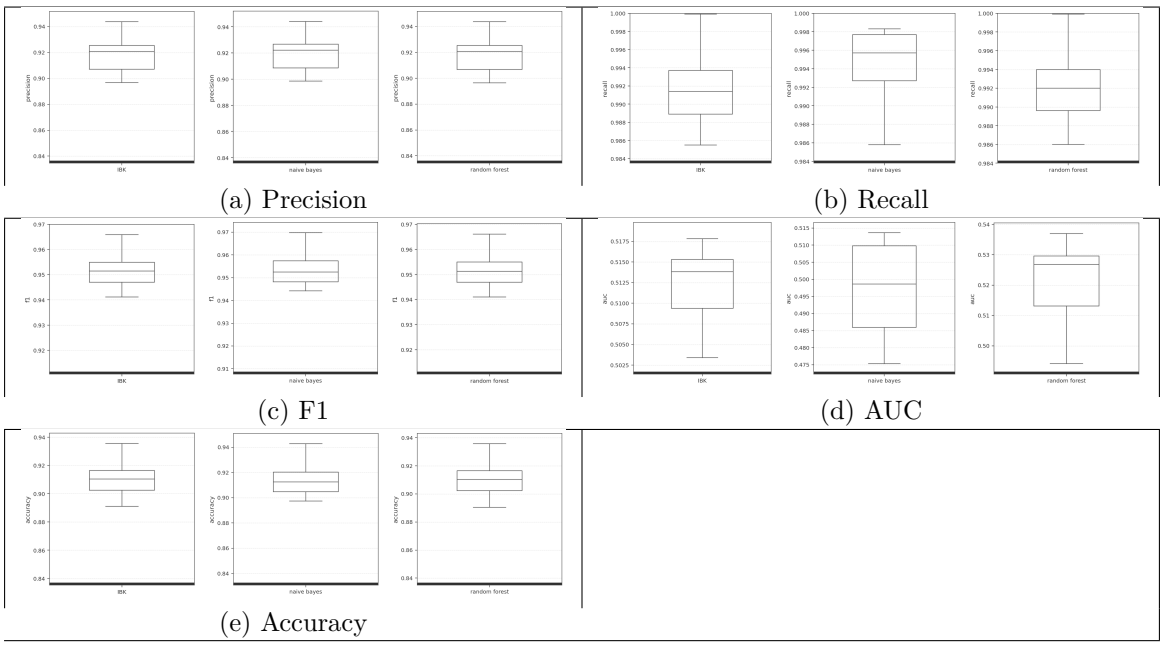
(a) Precision (b) Recall

(c) F1 (d) AUC

(e) Accuracy

Figura 3: BookKeeper: classificatori con features selection *BestFirst*

OpenJPA



(a) Precision (b) Recall

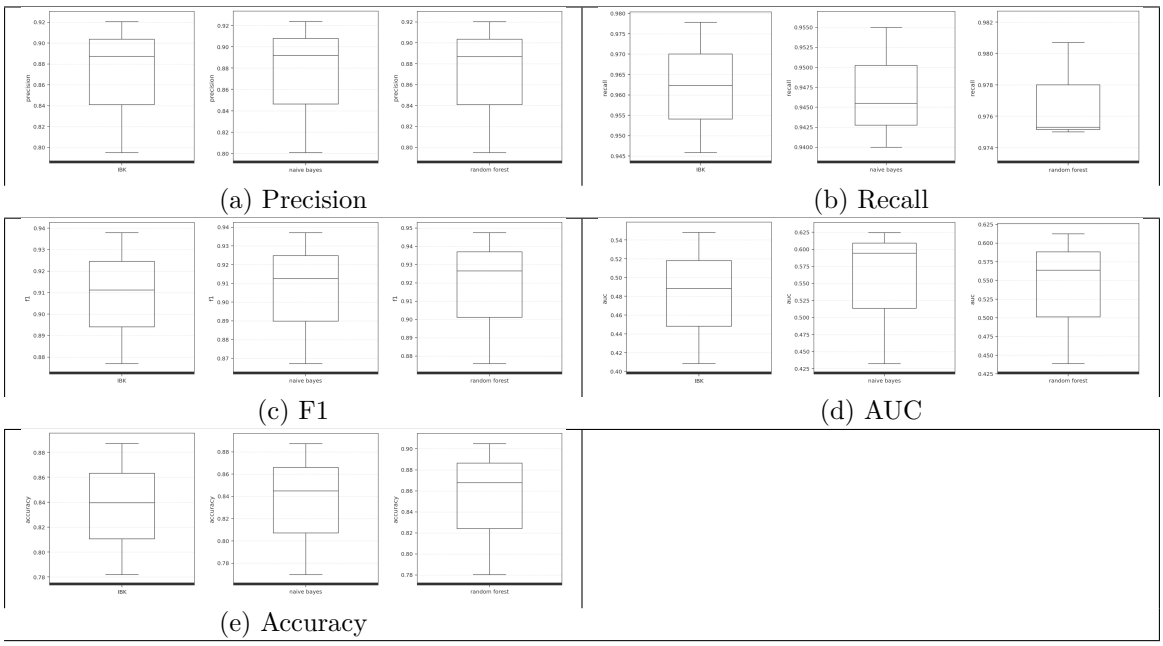
(c) F1 (d) AUC

(e) Accuracy

Figura 4: OpenJPA: classificatori con features selection *BestFirst*

Classificatori conn feature selection *GreedyStepwise Forward*

BookKeeper



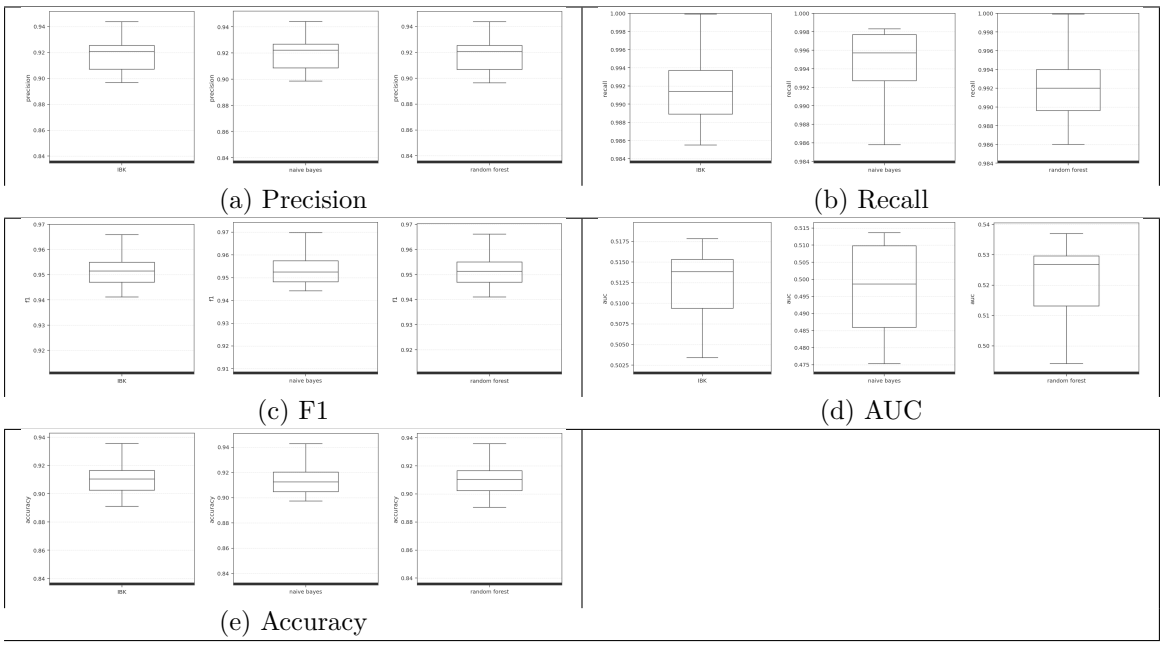
(a) Precision (b) Recall

(c) F1 (d) AUC

(e) Accuracy

Figura 5: BookKeeper: classificatori con feature selection *GreedyStepwise Forward*

OpenJPA



(a) Precision (b) Recall

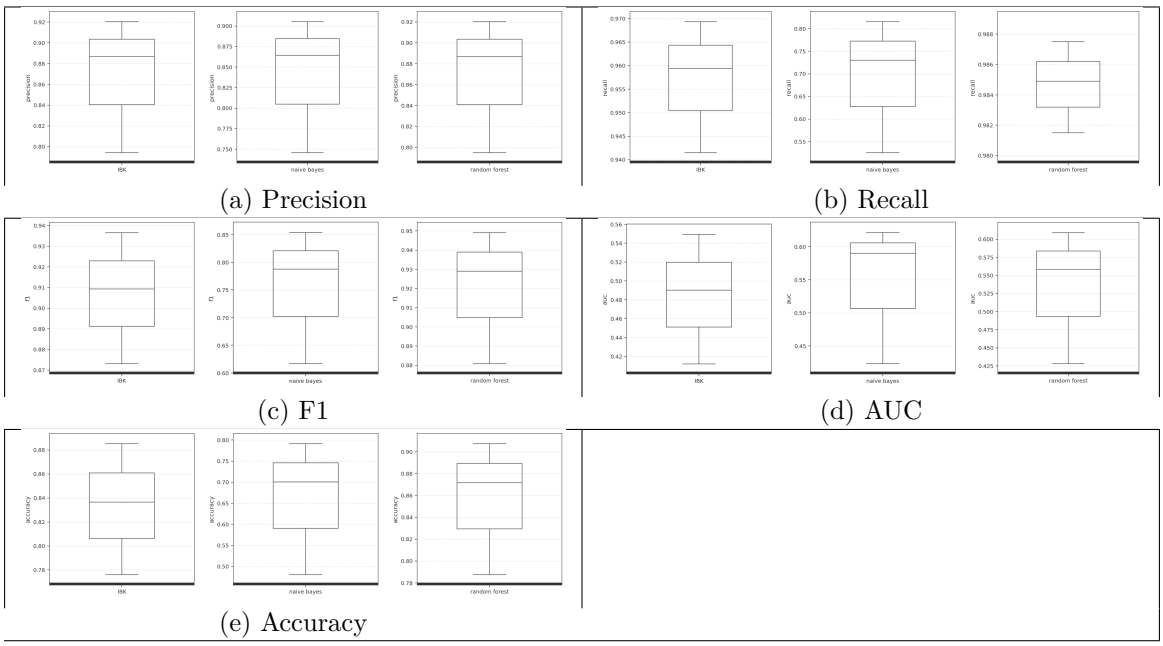
(c) F1 (d) AUC

(e) Accuracy

Figura 6: OpenJPA: classificatori con feature selection *GreedyStepwise Forward*

Classificatori con feature selection *GreedyStepwise Backward*

BookKeeper



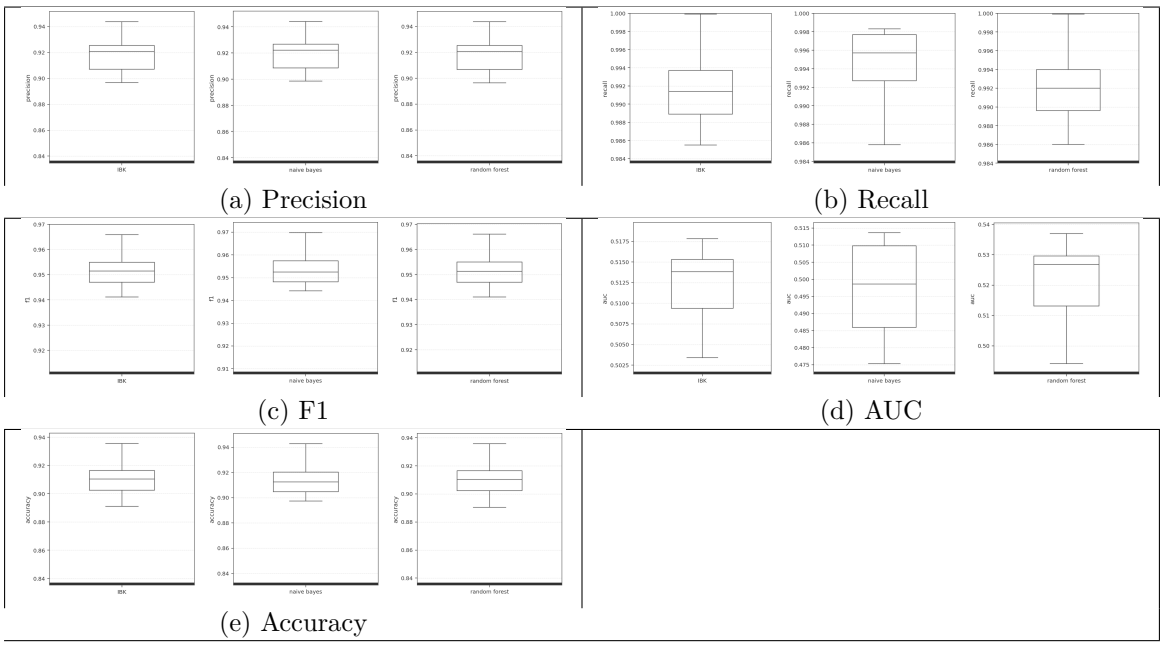
(a) Precision (b) Recall

(c) F1 (d) AUC

(e) Accuracy

Figura 7: BookKeeper: classificatori con feature selection *GreedyStepwise Backward*

OpenJPA



(a) Precision (b) Recall

(c) F1 (d) AUC

(e) Accuracy

Figura 8: BookKeeper: classificatori con feature selection *GreedyStepwise Backward*

.4 What-if Analysis

Tabella 5: Risultati dell'analisi What-If per **BookKeeper**.

	Dataset A	Dataset B+	Dataset B	Dataset C
A	1196	323	–	873
E	1114	339	181	775

Tabella 6: Risultati dell'analisi What-If per **OpenJPA**.

	Dataset A	Dataset B+	Dataset B	Dataset C
A	13699	4289	–	9410
E	10000	3057	2625	6943