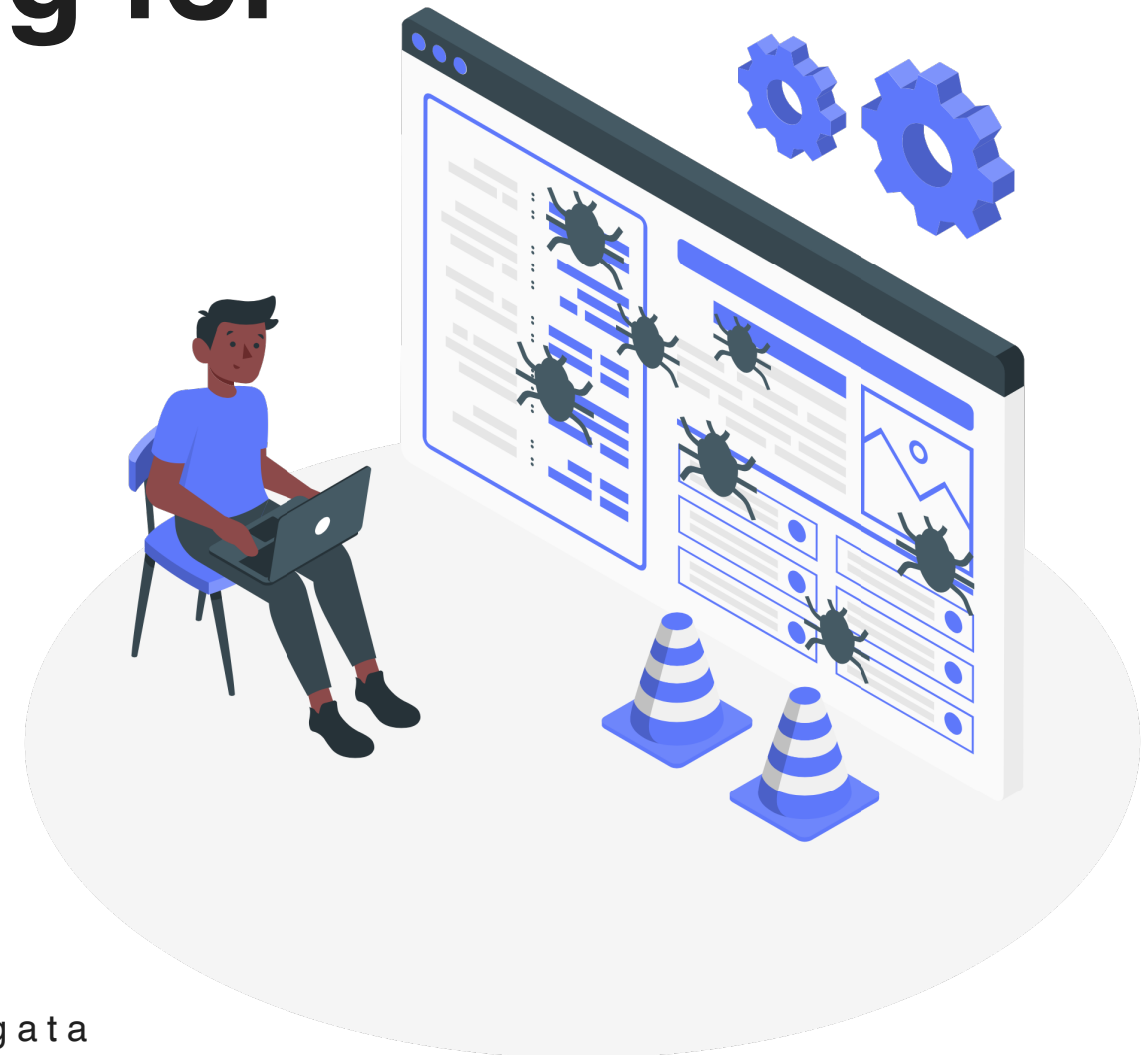


Machine Learning for SOFTWARE ENGINEERING.

Defect Prediction.

Pierpaolo Spaziani

Matricola: 0316331



Università degli Studi di Roma Tor Vergata

Indice.

- ❖ Introduzione
- ❖ Obiettivi
- ❖ Progettazione
- ❖ Analisi dei risultati: [BookKeeper](#)
- ❖ Analisi dei risultati: [OpenJPA](#)
- ❖ Conclusioni
- ❖ Link

Introduzione.

Perché?

L'attività di **software testing** è di fondamentale importanza.

Permette di far emergere **malfunzionamenti** nel sistema e di individuare **bug**.

Tuttavia testare tutto potrebbe essere troppo dispendioso.

Sapere a priori cosa testare e perchè porterebbe a numerosi vantaggi:

- ❖ **Meno ore spese a scrivere test**
- ❖ **Meno impegno economico**
- ❖ **Più tempo da dedicare a progettazione e sviluppo**

Introduzione.

Come?

Machine learning!

È possibile utilizzare i classificatori per predire la difettosità delle classi.

Le metriche che aiutano a valutare il modello sono:

- ❖ **Precision**: percentuale di quanti dei *positive* restituiti sono *true positive*
- ❖ **Recall**: percentuale di quanti *positive* ha indovinato il modello rispetto ai *positive* reali
- ❖ **AUC**: capacità del modello di classificare correttamente le istanze positive rispetto alle negative
- ❖ **Kappa**: quanto il classificatore si è comportato meglio rispetto ad un classificatore dummy

Obiettivi.

Gli obiettivi posti in questo studio sono stati quindi di analizzare le prestazioni dei classificatori utilizzati per **predire la difettosità** delle classi nei progetti open-source:

❖ **Apache BookKeeper**

❖ **Apache OpenJPA**

In particolare, i classificatori utilizzati sono:

❖ **Random Forest**

❖ **Naive Bayes**

❖ **IBk**

Progettazione.

- ❖ Fetch delle release
- ❖ Fetch delle issue
- ❖ Proportion
- ❖ Fetch dei commit e dei file
- ❖ Metriche
- ❖ Evaluation
- ❖ Balancing

Progettazione.

Fetch delle release.

La lista delle **release** per ogni progetto è stata recuperata da  Jira.

La piattaforma fornisce le Rest API che restituiscono un JSON dal quale è possibile estrarre:

- ❖ **nomi delle release**

- ❖ **date di rilascio**

In base a quest'ultimo campo è stato possibile ordinarle per avere un allineamento temporale con la lista di issue.

Per ovviare al fenomeno dello **snoring**, la seconda metà è stata scartata.

Progettazione.

Fetch delle issue.

Sempre da  Jira, è stata recuperata la lista delle **issue** in base a:

Type == “Bug” AND
(status == “Closed” OR status == “Resolved”)
AND Resolution == “Fixed”

Per ogni issue, sono state registrate (se presenti):

- ❖ **Injected Version**
- ❖ **Opening Version**
- ❖ **Fix Version**

**Issue con IV:*

BookKeeper ~ 40%

OpenJPA ~ 60%

Progettazione.

Proportion.

Dopo aver **scartato** le issue prive di Fix Version e quelle inconsistenti, ovvero con Opening Versioni $>$ Fix Version, il problema è che **non tutte le restati sono provviste di Injected Version***!


Per ovviare a questo, è stata utilizzata la tecnica dell'**Incremental Proportion** per calcolarla ove necessario.

In ogni release k , è stato calcolato P_k come $\frac{FV - IV}{FV - OV}$ con tutte le issue delle release $1, \dots, k - 1$ provvisti di tutte e 3 le versioni.

Nelle issue prive, è stata calcolata l'Injected Version: $IV = (FV - OV) \cdot P$, con P pari alla media dei P_k .

Progettazione.

Fetch dei commit e dei file.

Tramite  **GitHub**, sono stati recuperati, per ogni release, la **lista dei file e dei commit**.

In questo modo è stato possibile procedere analizzando ogni commit di ogni release e **calcolare le metriche** per ogni classe.

Essendo i **commit *bugfix*** accompagnati dal relativo numero di issue, è stato possibile etichettare le classi come ***buggy*** dall'Injected Version fino alla release precedente alla Fix Version.

Progettazione.

Metriche.

Le metriche forniscono al classificatore le linee guida per etichettare le classi come buggy o meno.

Ne sono state selezionate 10:

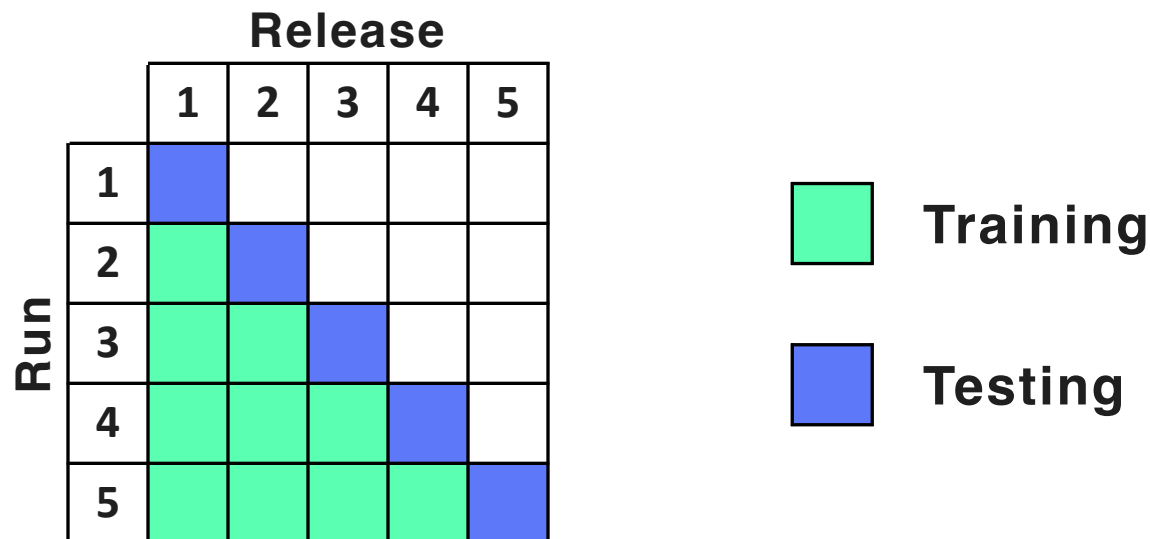
- ❖ Age
- ❖ Number of Revisions
- ❖ Number of Bugfix
- ❖ LOCs
- ❖ LOCs Touched
- ❖ LOCs Added
- ❖ Churn
- ❖ Average Churn
- ❖ Authors Number
- ❖ Average Change Set

Progettazione.

Evaluation.

Per stabilire quale dei classificatori ha le prestazioni migliori, è necessario effettuarne una valutazione.

La tecnica utilizzata è stata il **Walk-Forward**:




**Classi buggy:*

BookKeppep ~ 20%

OpenJPA ~ 5%

Progettazione.

Balancing.

Dal momento che il dataset è fortemente sbilanciato*, gli studi tramite **Weka**  sono stati effettuati analizzando i risultati dei 3 classificatori in 4 scenari di **balancing** differenti:

- ❖ No Sampling
- ❖ Undersampling
- ❖ Oversampling
- ❖ SMOTE

BookKeeper.

Recall.

BALANCING:

- No Sampling
- Oversampling
- SMOTE
- Undersampling

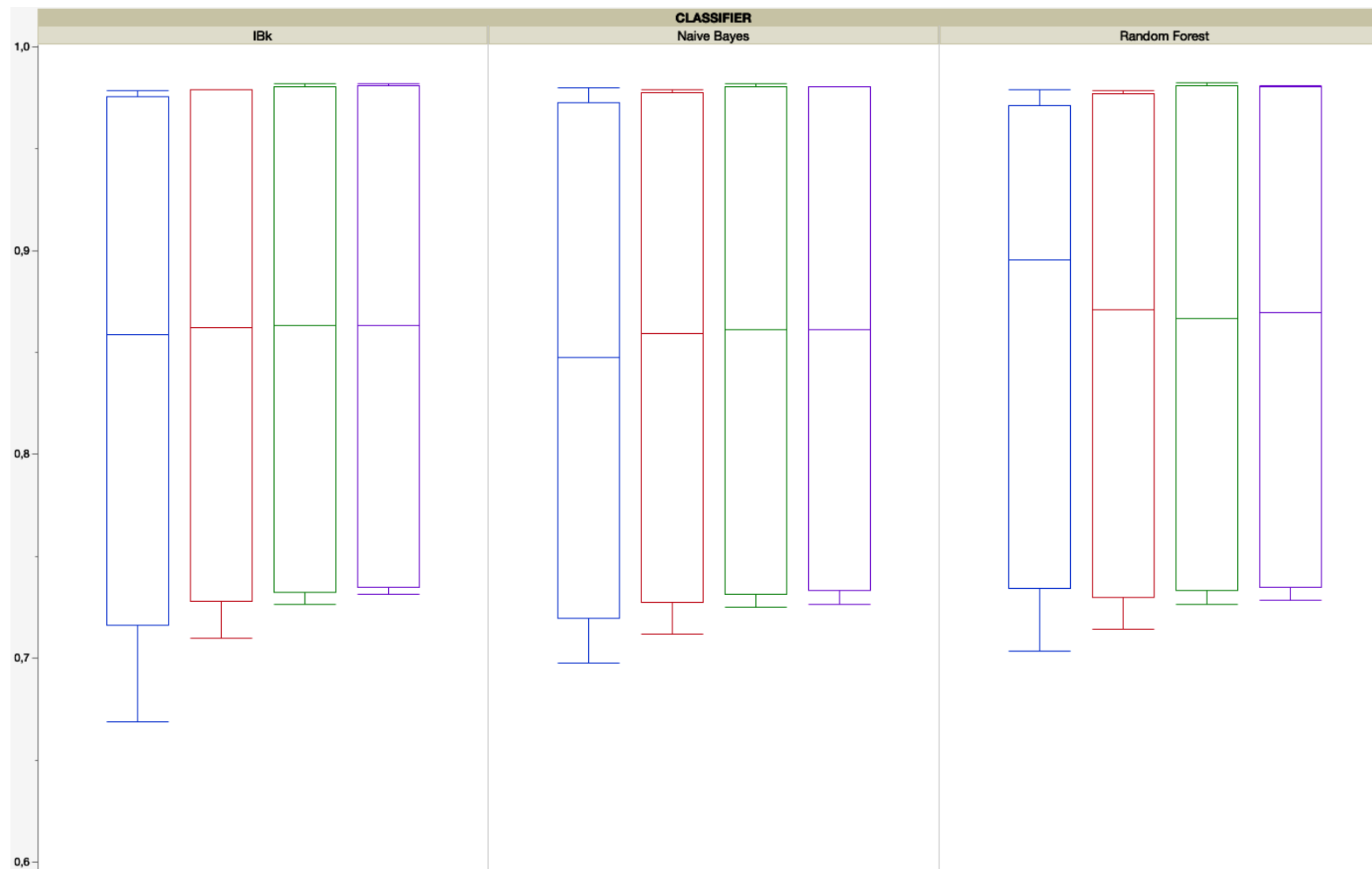


BookKeeper.

Precision.

BALANCING:

- No Sampling
- Oversampling
- SMOTE
- Undersampling



BookKeeper.

AUC.

BALANCING:

- No Sampling
- Oversampling
- SMOTE
- Undersampling

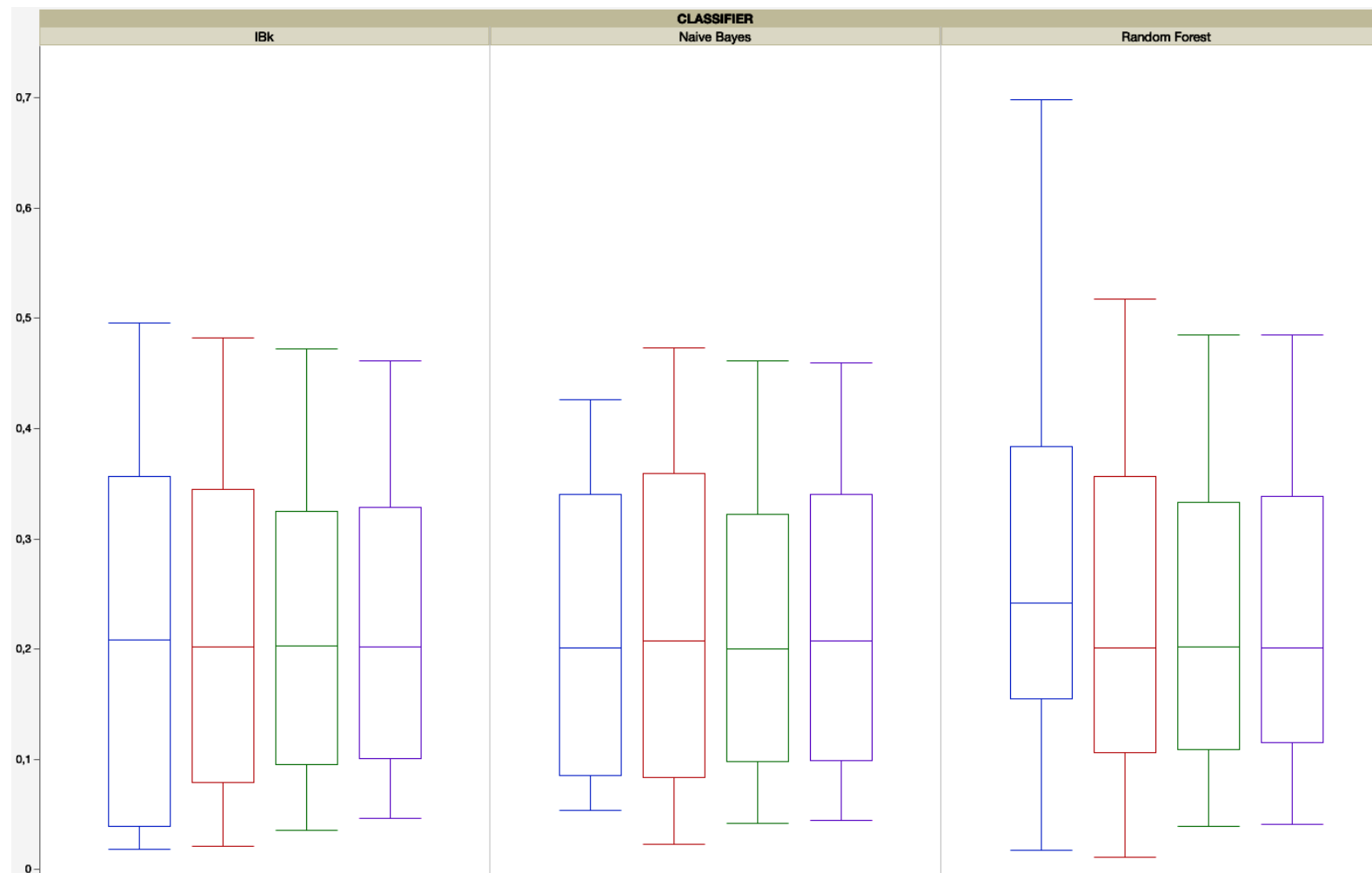


BookKeeper.

Kappa.

BALANCING:

- No Sampling
- Oversampling
- SMOTE
- Undersampling



BookKeeper.

Considerazioni.

Non si evidenziano grandi disparità tra i **classificatori**, tutti e tre presentano gli **stessi andamenti**.

Invece, al contrario delle aspettative, **non applicare sampling** sembrerebbe la **scelta migliore!**

La possibile spiegazione di questo fenomeno è che, essendo i dati molto eterogenei, applicare le tecniche di sampling non aiuta il classificatore a discriminare le classi.

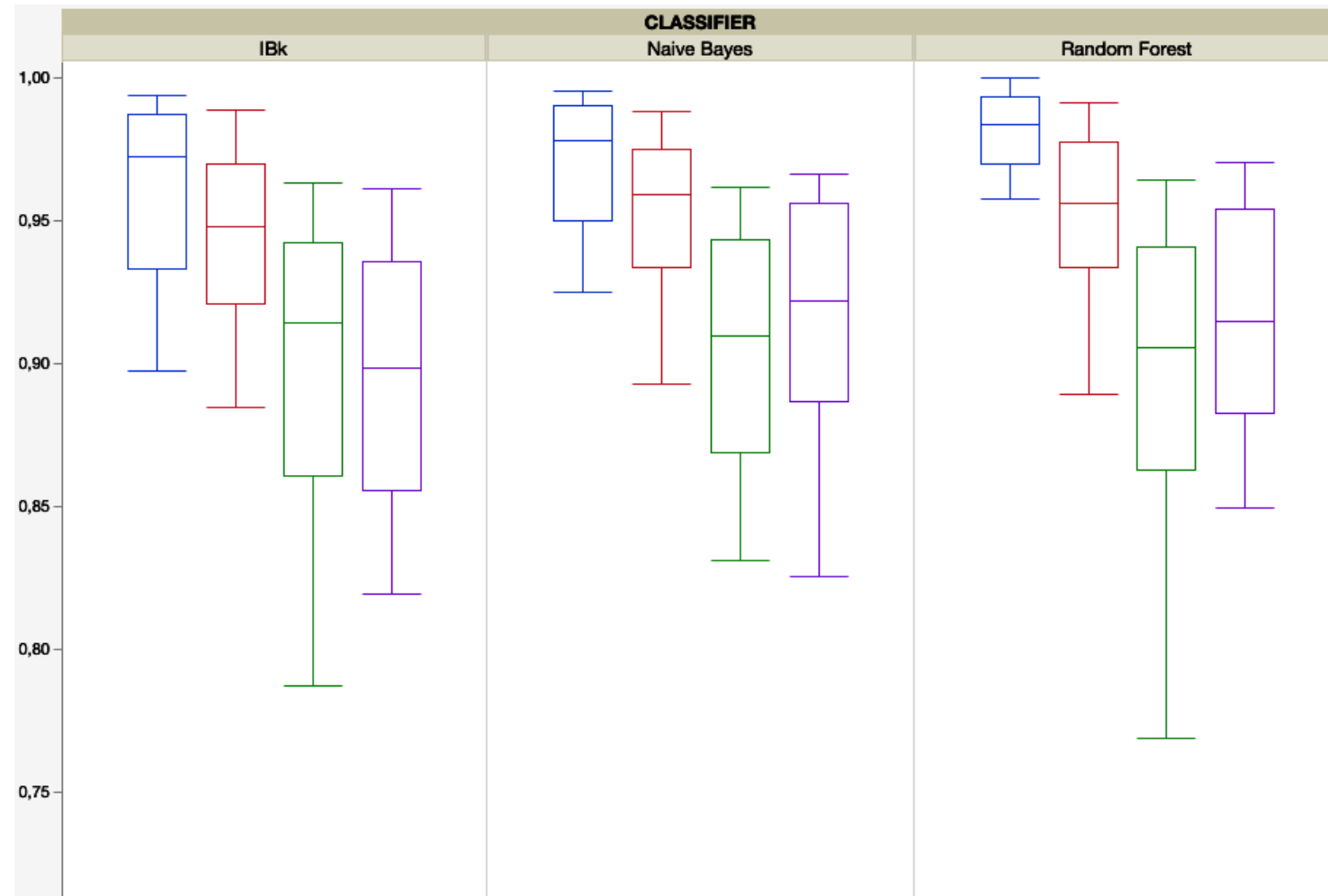
Probabilmente, sperimentarle insieme a tecniche di **feature selection** e **cost sensitive** porterebbe lo studio a loro favore.

OpenJPA.

Recall.

BALANCING:

- No Sampling
- Oversampling
- SMOTE
- Undersampling

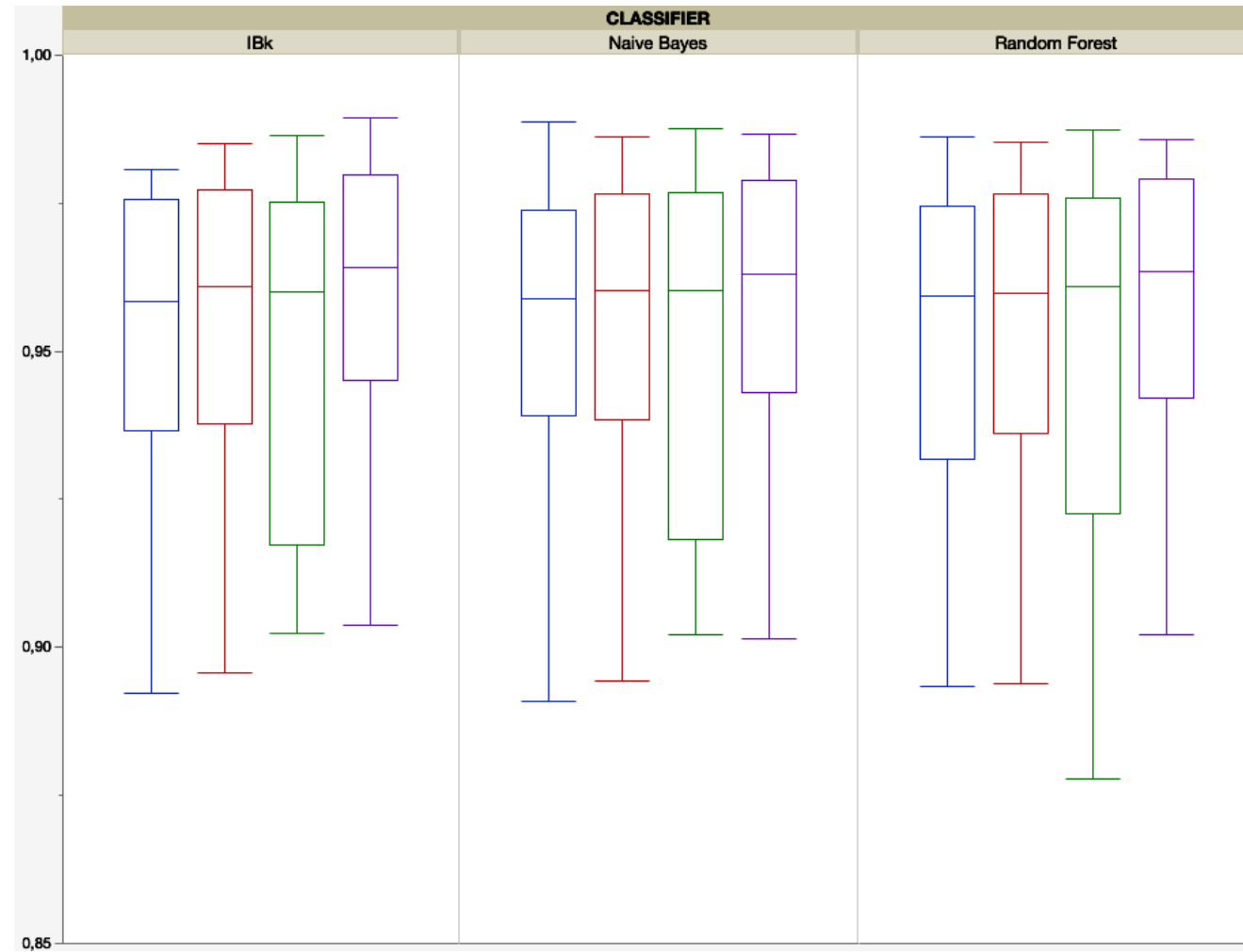


OpenJPA.

Precision.

BALANCING:

- No Sampling
- Oversampling
- SMOTE
- Undersampling

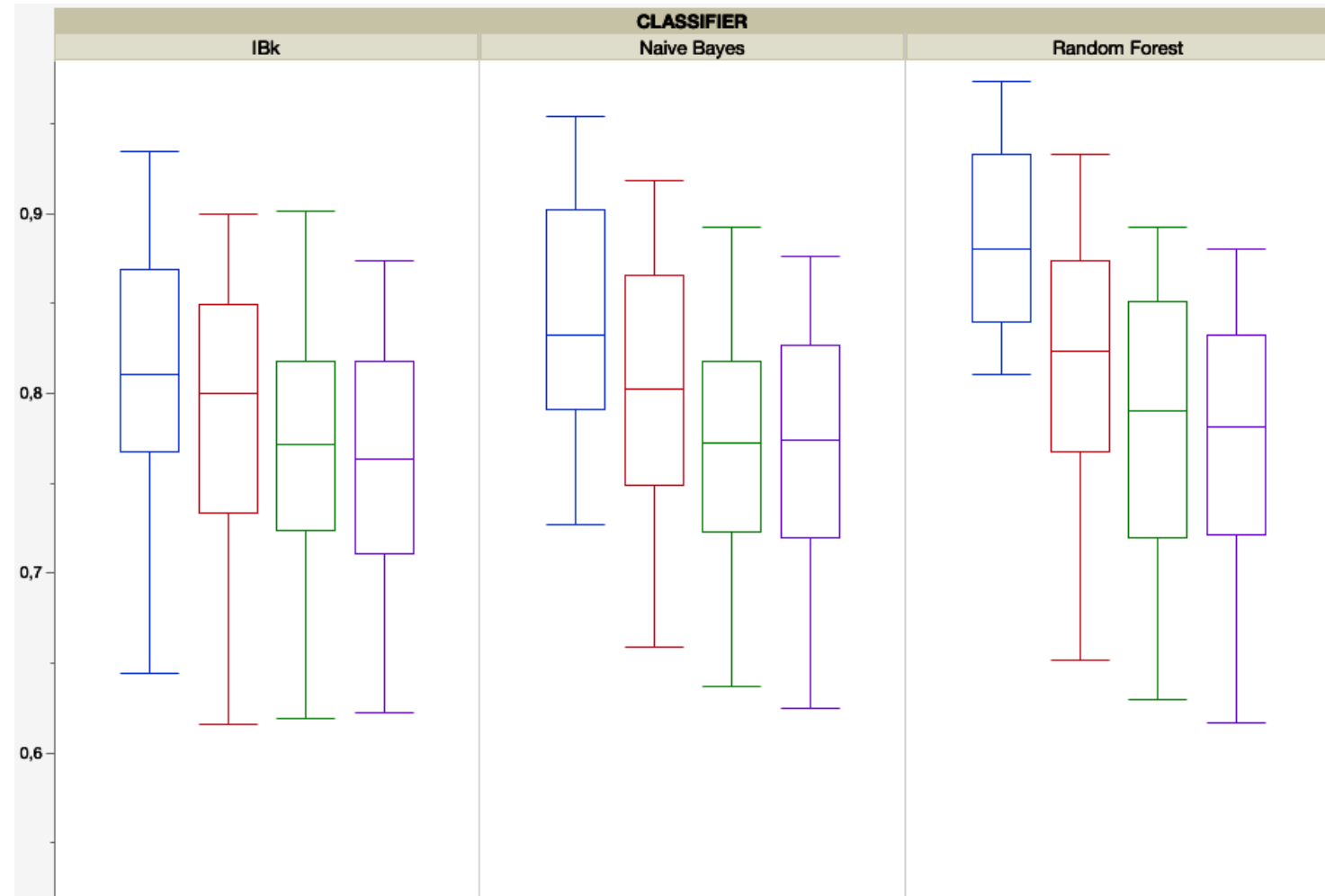


OpenJPA.

AUC.

BALANCING:

- No Sampling
- Oversampling
- SMOTE
- Undersampling

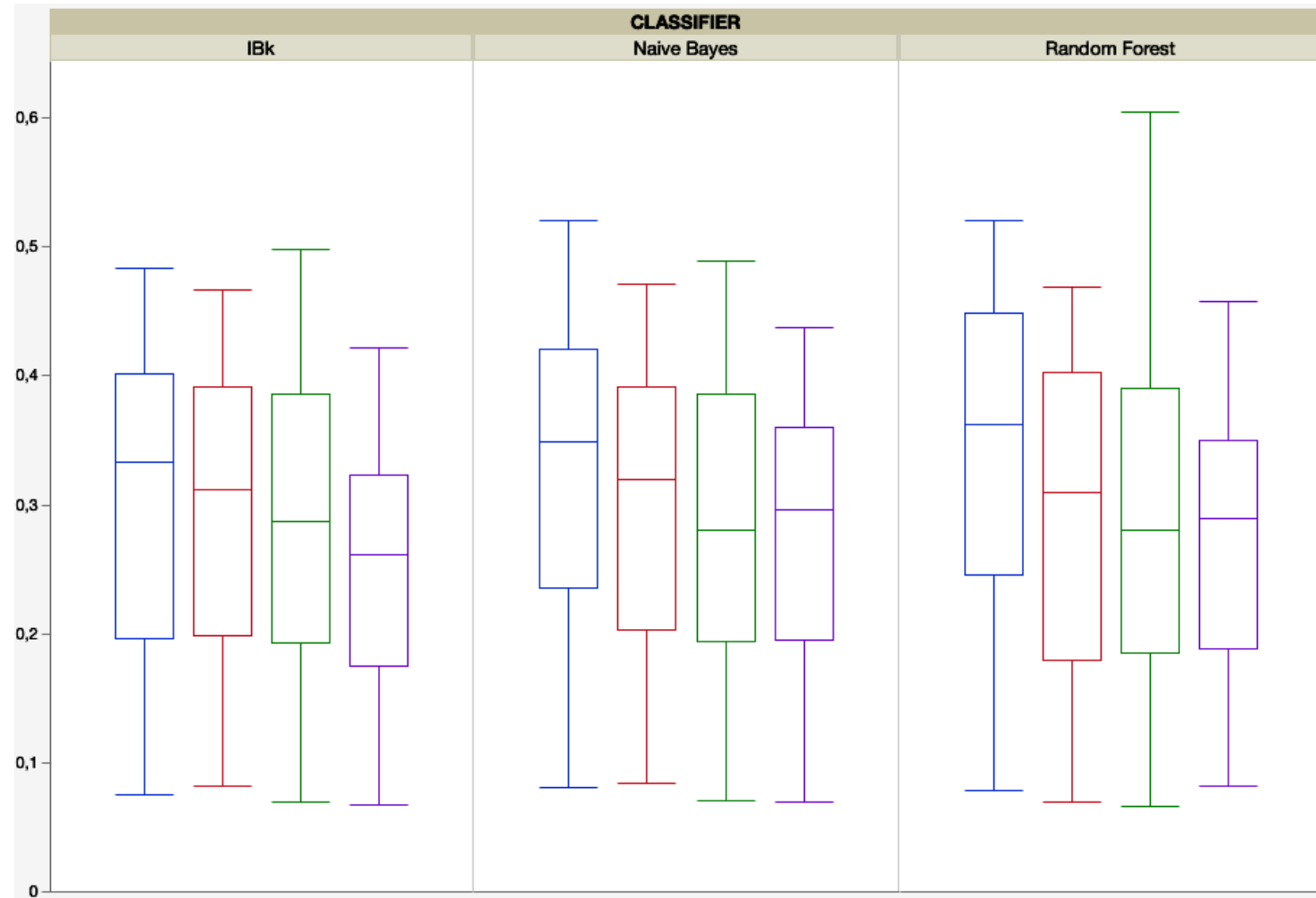


OpenJPA.

Kappa.

BALANCING:

- No Sampling
- Oversampling
- SMOTE
- Undersampling



OpenJPA.

Considerazioni.

Anche in questo caso, i **classificatori** presentano gli **stessi andamenti** e **non applicare sampling** sembra essere la **scelta migliore**.

Tuttavia, risultano molto più evidenti ed interessanti le **disparità** tra le **altre 3 tecniche di sampling!**

L'**Oversampling** si comporta meglio delle altre 2 con tutte le metriche.

SMOTE e **Undersampling** invece hanno tendenzialmente gli stessi andamenti.

Come nel caso precedente, sarebbe interessante osservare se, combinando l'oversampling con le tecniche di **feature selection** e **cost sensitive**, si riuscirebbe a superare il caso senza sampling e colmare il gap dell'eterogeneità del dataset.

Conclusioni.

I risultati ottenuti hanno evidenziato come, nonostante le caratteristiche dei dataset, i **classificatori** raggiungono valori di **affidabilità elevata**.

Affidarsi a framework di questo tipo in ambito enterprise, permetterebbe di **ridurre l'impatto economico** ed **incrementare il tasso di sviluppo**, lasciando **invariato il livello di qualità** del codice.

Avere un sistema con un tasso di bugginess basso vuol dire avere **codice riutilizzabile** e qualitativamente buono.

Non bisogna mai ignorare la relazione fra **qualità del prodotto**, **qualità del processo** e **successo dell'organizzazione**.

Il **tempo** rimane una risorsa fondamentale e limitata, poter decidere dove impiegarlo può rivelarsi una **caratteristica vincente!**

Grazie per l'attenzione!

Link.



Repositories:

Milestone 1:

<https://github.com/pierpaolospaziani/ISW>

Milestone 2:

<https://github.com/pierpaolospaziani/Weka>



Code Analysis:

Milestone 1:

https://sonarcloud.io/project/overview?id=pierpaolospaziani_ISW

Milestone 2:

https://sonarcloud.io/project/overview?id=pierpaolospaziani_weka