Machine Learning for Software Engineering

Defect Prediction

Valerio Crecco – 0320452

Università degli studi di Roma Tor Vergata

Indice

- Introduzione
- Progettazione
- Analisi dei risultati: **BookKeeper**
- Analisi dei risultati: **ZooKeeper**
- Conclusioni
- Link

Introduzione

La previsione dei difetti mira ad identificare artefatti software che possono presentare un difetto. Prevedere i difetti è fondamentale per:

- ridurre il **costo** dell'attività di **testing**;
- migliorare **il processo di sviluppo** del sw;
- usare in modo efficiente le **risorse limitate** che si hanno a disposizione;

Introduzione

L'**obiettivo** dello studio è quello di misurare l'andamento di dei **classificatori** utilizzati per **predire la difettosità** delle classi dei progetti:

- **BookKeeper;**
- **ZooKeeper;**

I **classificatori** utilizzati sono: **Random Forest**, **Naive Bayes** e **IBk**.

Introduzione

Per ogni **classificatore** utilizzato sono state analizzate:

- **Precision:** quante volte un'istanza **positiva** è stata classificata correttamente*;*
- **Recall:** quanti **positivi** ha indovinato il modello rispetto al totale dei **positivi** 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*** *;*

Progettazione – releases

Per ottenere la lista delle **release** per ciascun progetto è stata utilizzata la **RestAPI** di **JIRA**.

Questa restituisce un **JSON** dal quale si possono ottenere:

- **nomi delle release**
- **date di rilascio**

Si è utilizzato quest'ultimo campo per avere un ordinamento temporale con la lista di issues.

Per ovviare al fenomeno dello **snoring**, la seconda metà è stata scartata, quindi sono state considerate:

- **7 release** per **BookKeeper**
- **24 release** per **ZooKeeper**

Progettazione – issues

La lista delle **issues** è stata recuperata da Jira, tramite l'uso dell'API JSON di JIRA, selezionando issue:

- il cui **tipo** fosse «**Bug**»;
- il cui **status** fosse «**Closed**» o «**Resolved**»;
- la cui **risoluzione** fosse «**Fixed**»;

Per ognuna delle issues, sono state considerate: **Injected version (IV), Opening Version (OV)** e **Fix Version (FV)**

Progettazione – proportion

Dopo aver **rimosso** le issue **prive di FV** e quelle con **OV > FV**, il problema è che NON tutte le restati sono **provviste di IV**

Per questo motivo, è stata utilizzata la tecnica dell'**Incremental Proportion**.

Per ogni release ***r*** , è stato calcolato P_r come $\frac{FV - IV}{FV - OV}$

con tutte le issues delle release ***1, …., r-1*** provviste di tutte e **3** le versioni **(IV, OV, FV).**

Nelle issues prive, è stata calcolata $IV = FV - (FV - OV) * P$, con P pari alla **media** dei P_r

Progettazione – commit e file

Da **GitHub** sono stati recuperati, la **lista dei file e dei commit**, per ogni release, al fine di analizzare ogni **commit** di ciascuna 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

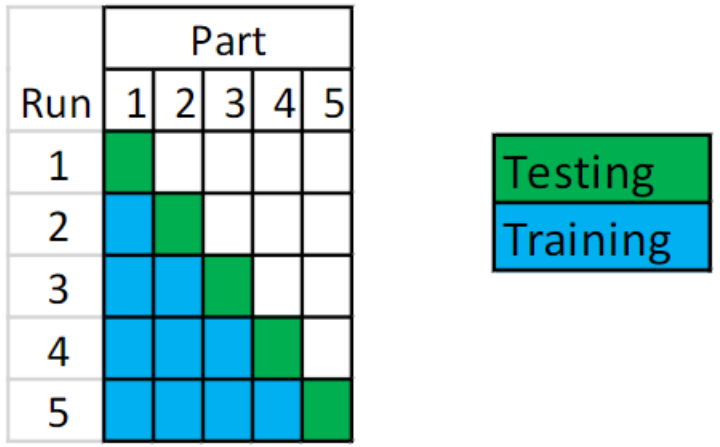
Le **metriche** che sono state considerate sono:

- **Age;**
- **Number of Revisions;**
- **Number of Bugfix;**
- **LOCs;**
- **LOCs Added;**
- **LOCs touched;**
- **Churn;**
- **Average Churn;**
- **Average Change Set**
- **Authors Number;**

Progettazione – evaluation

La tecnica utilizzata per fare **evaluation** è stata **Walk-Forward**, in quanto i dati sono correlati dal punto di vista temporale.

Tramite questa si è valutata la difettosità dei progetti nel corso delle release



Progettazione – balancing

Per ogni classificatore sono state analizzate: **Precision, Recall, AUC e Kappa;**

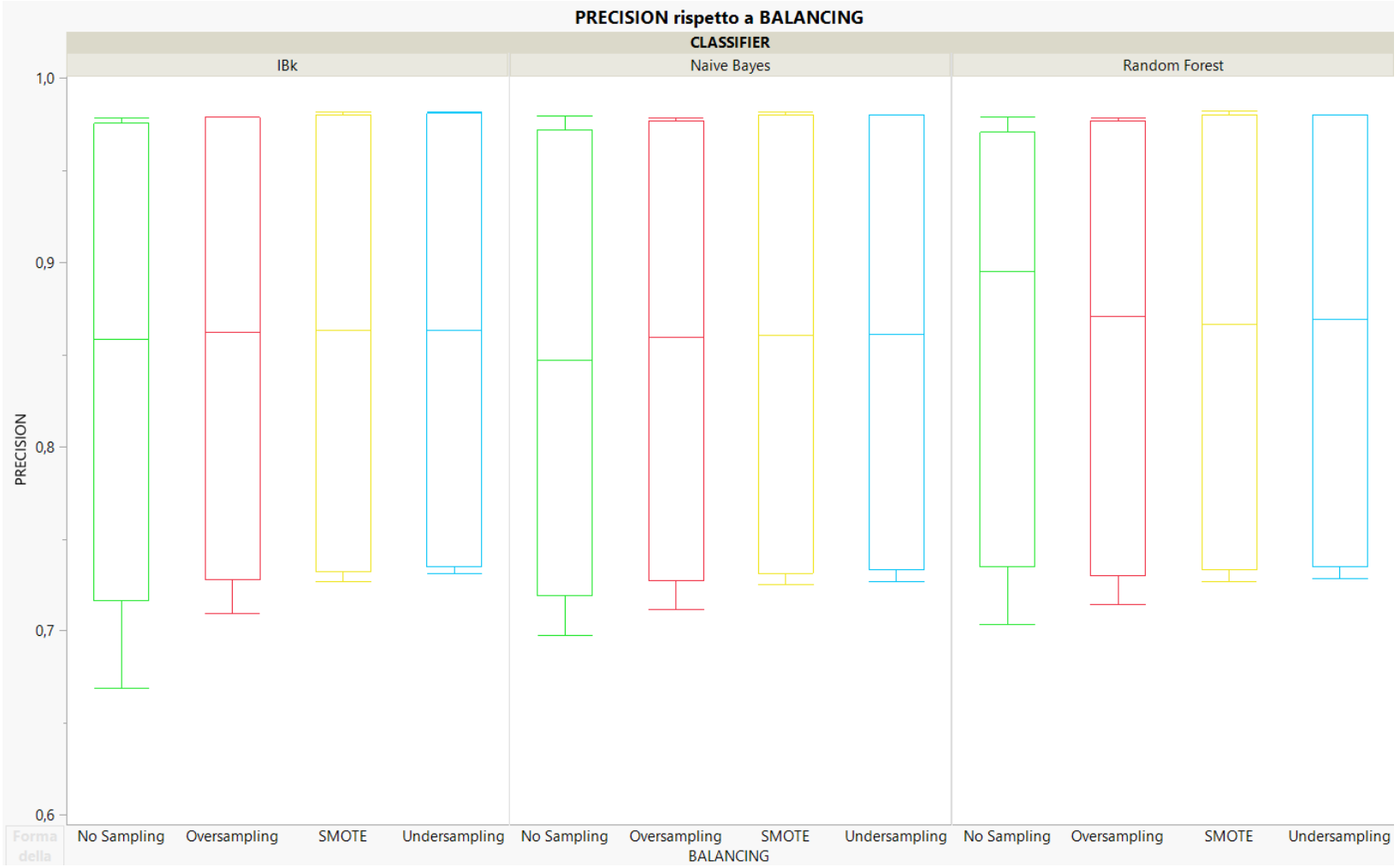
Applicando 4 scenari di **balancing** differenti:

- **No Sampling;**
- **Oversampling;**
- **SMOTE;**
- **Undersampling;**

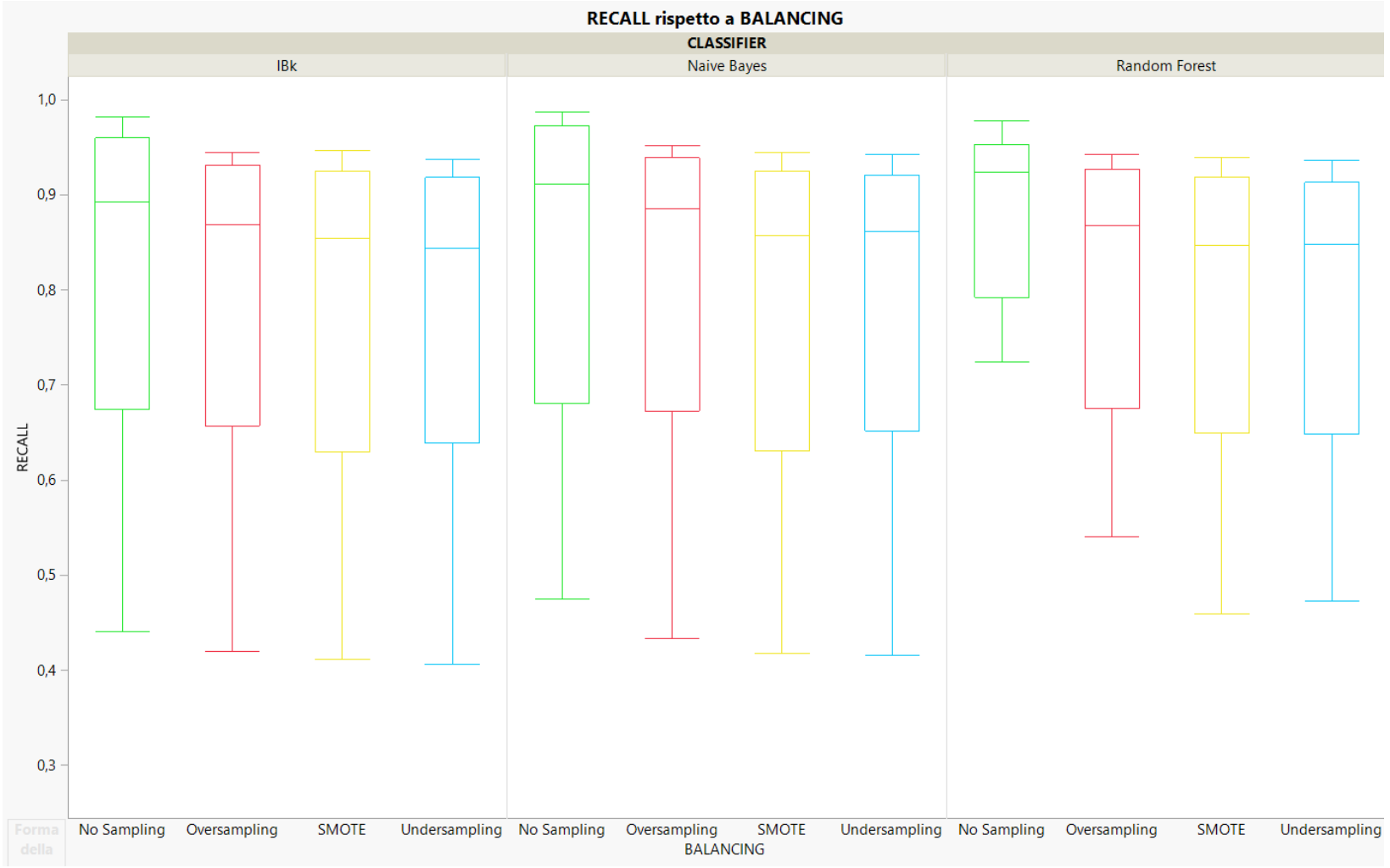
Sono state classificate come classi **buggy**:

- circa il **19%** delle classi in **BookKeeper**;
- circa il **16%** delle classi in **ZooKeeper**;

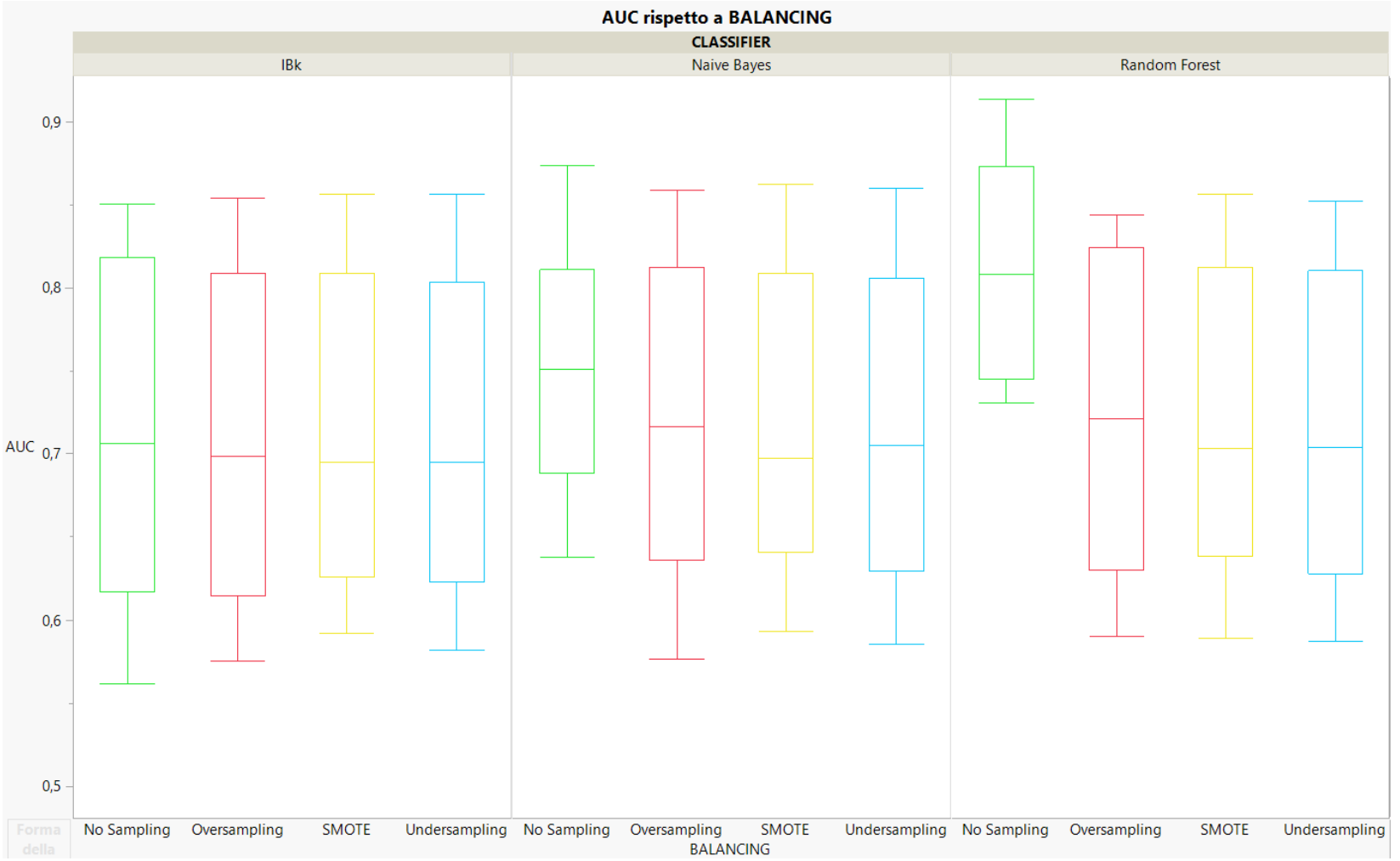
BookKeeper – Precision



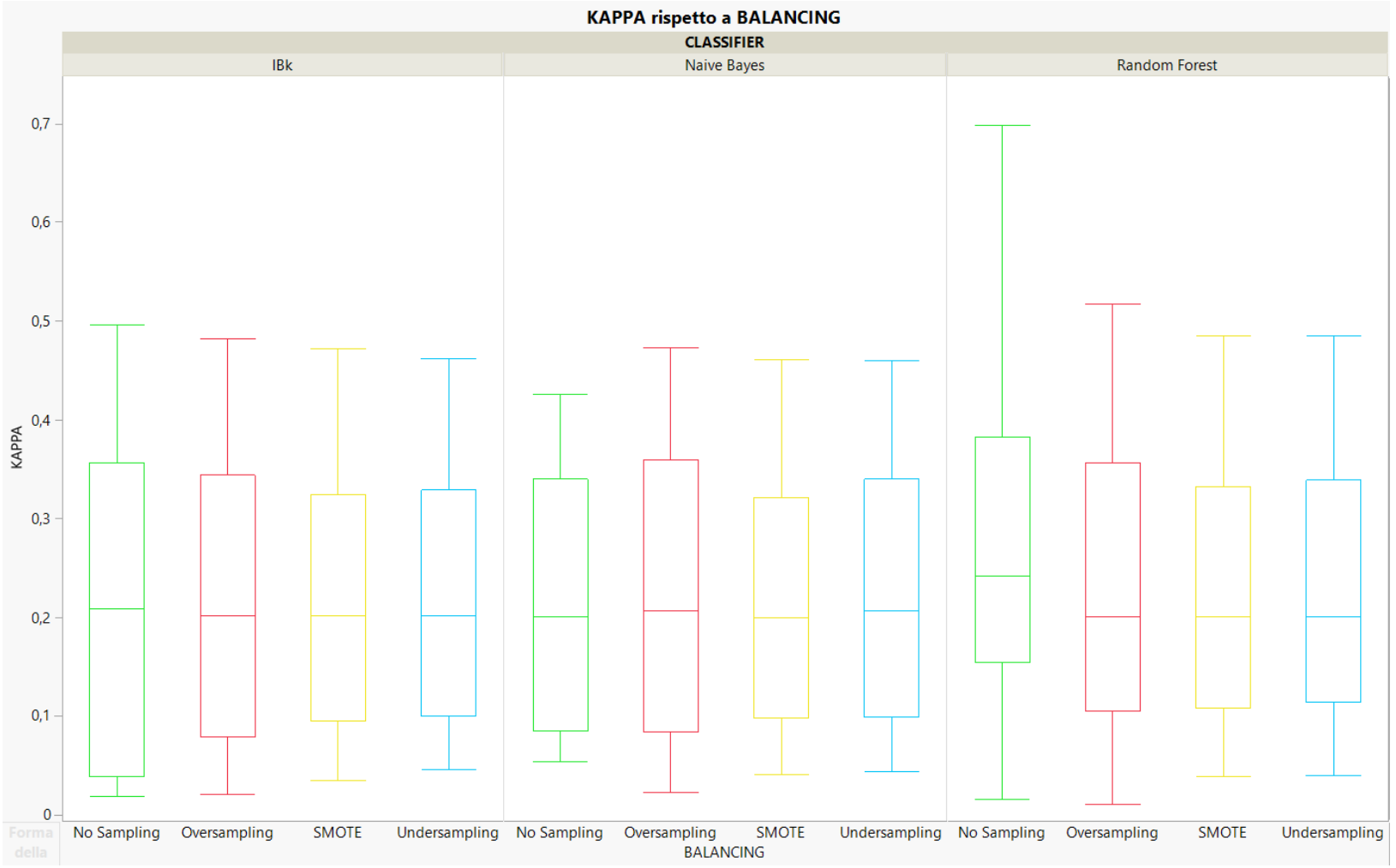
BookKeeper – Recall



BookKeeper – AUC



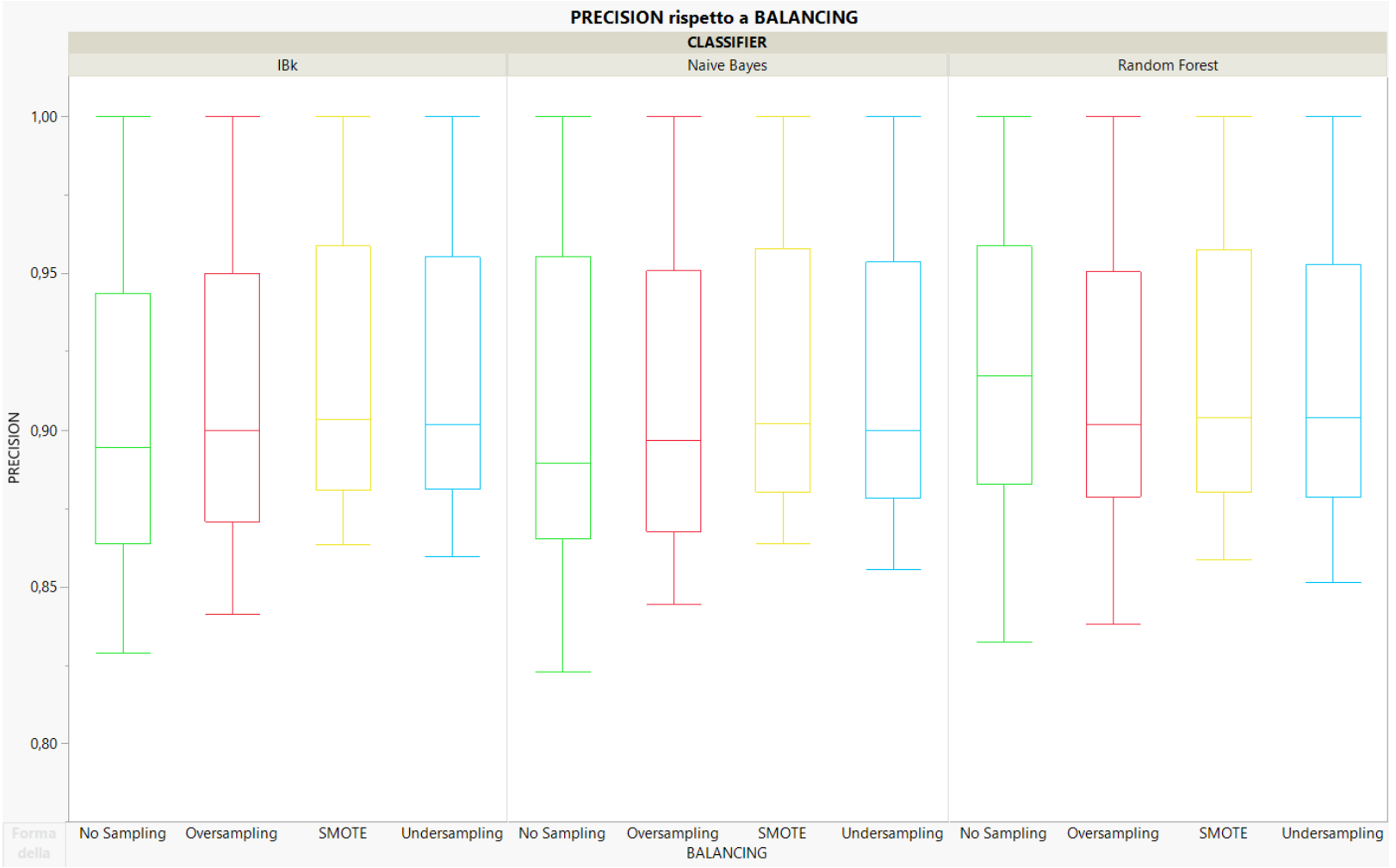
BookKeeper – Kappa



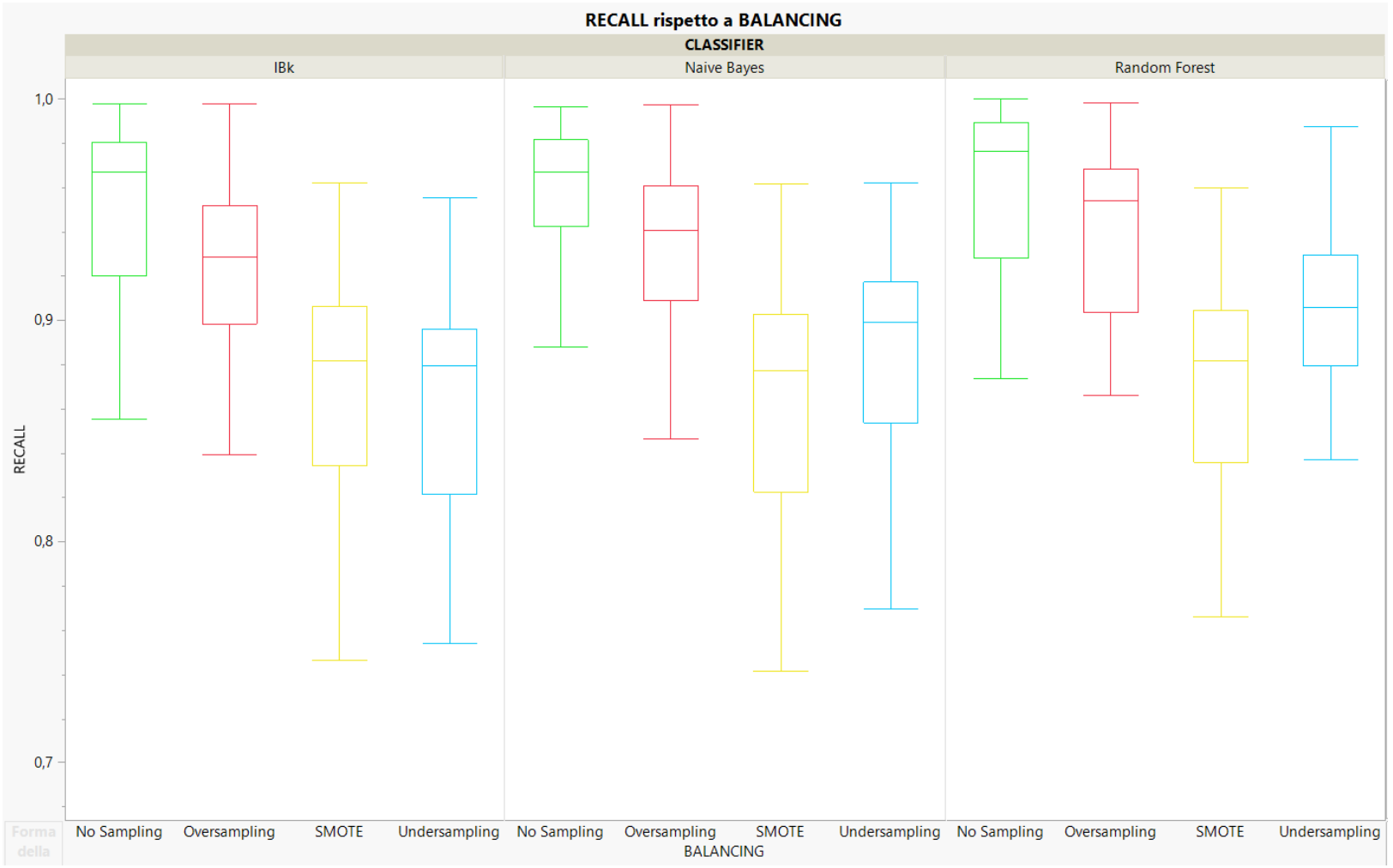
BookKeeper – analisi risultati

- Osservando i risultati NON si osservano grandi differenze tra i tre classificatori per quanto riguarda la **precision;**
- Su tutti e tre i classificatori, il **NO sampling** permette di avere una maggiore **recall**, mentre applicando **SMOTE** o **undersampling** si ha una **recall** leggermente più bassa;
- Possiamo notare che con **NO sampling**, si hanno delle performance migliori per i classificatori **NaiveBayes** e soprattutto **RandomForest** per quanto riguarda la metrica **AUC**;
- Anche per quanto riguarda **Kappa** le performance migliori si hanno con il **NO sampling** e con il classificatore **RandomForest;**

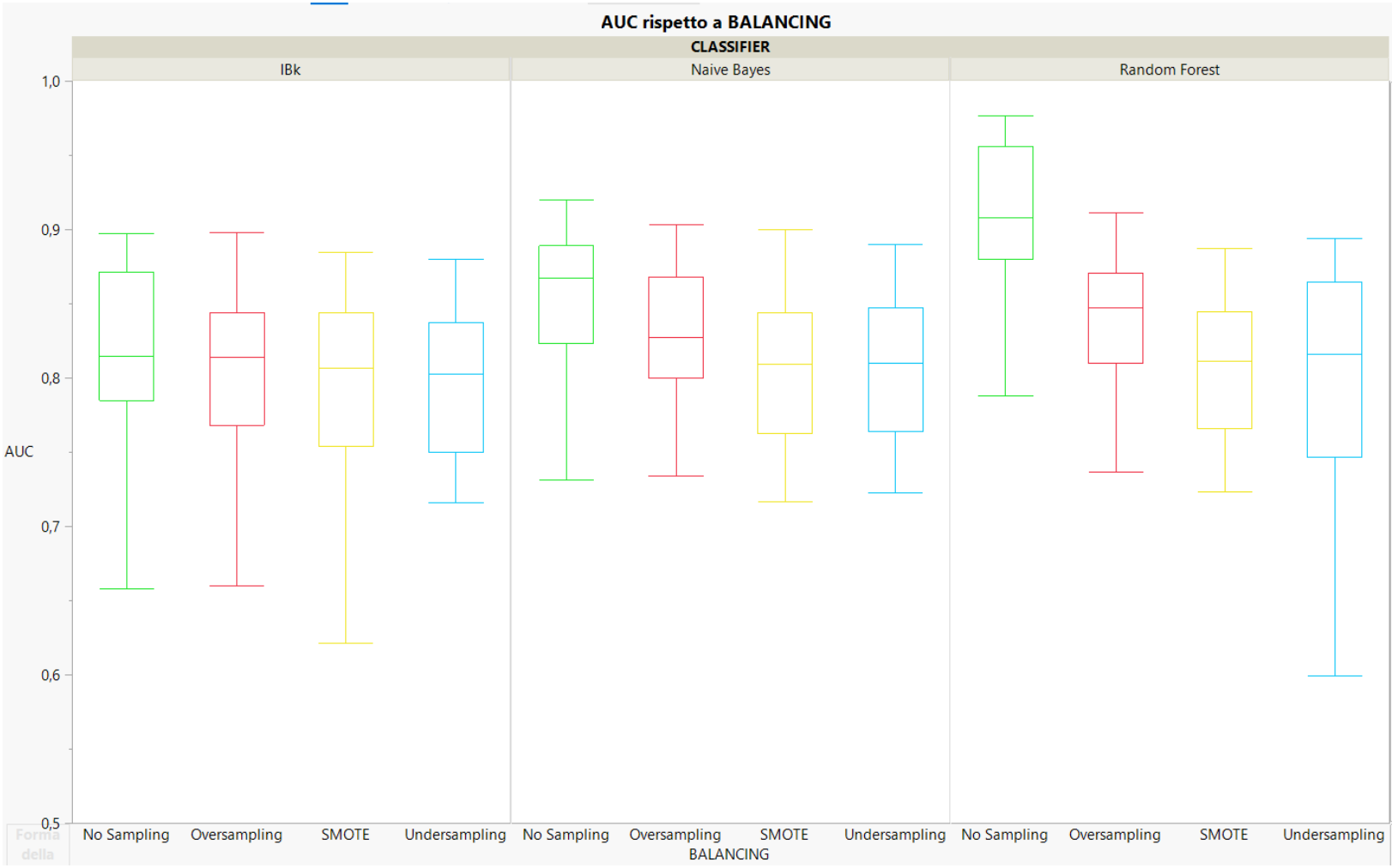
ZooKeeper – Precision



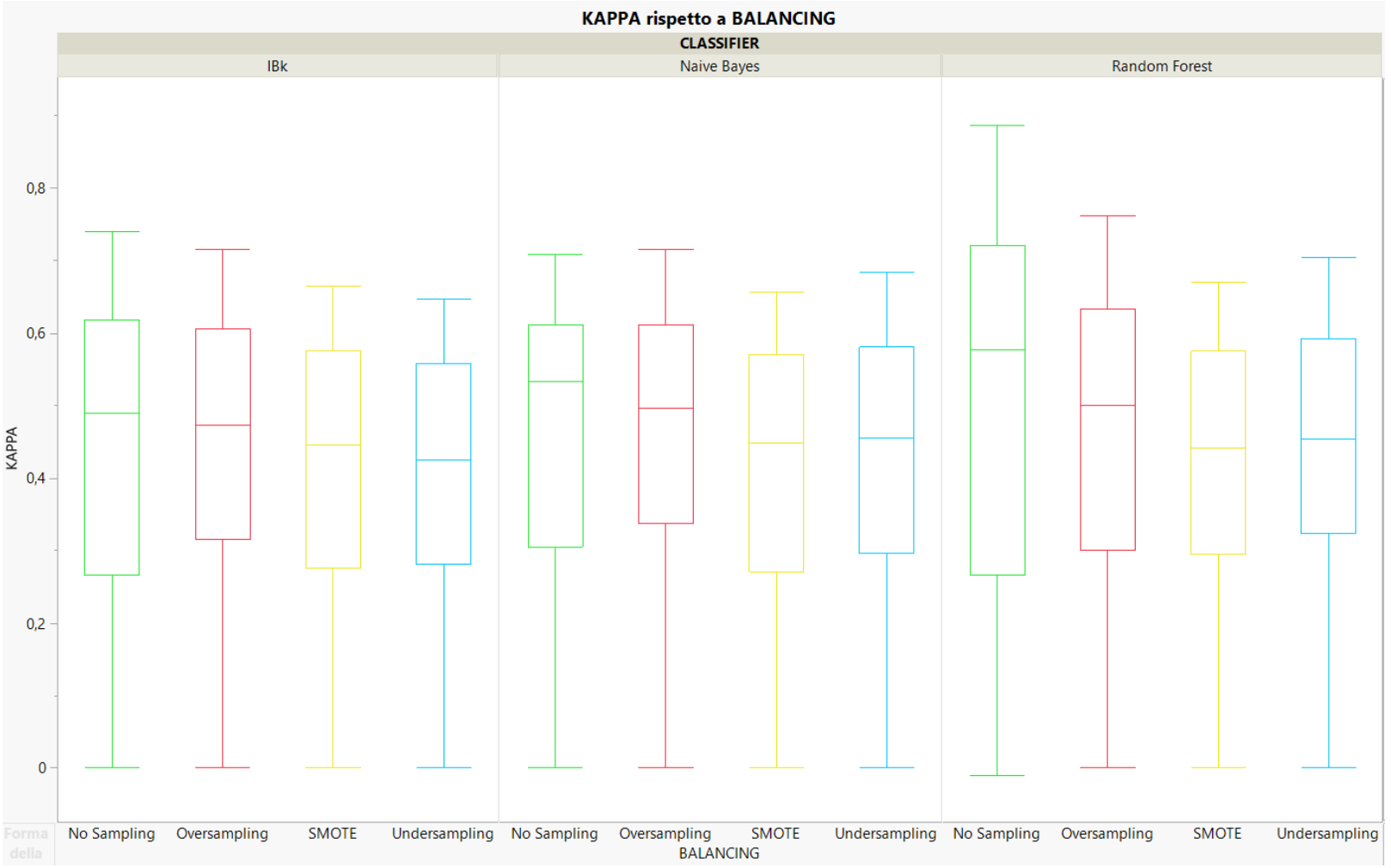
ZooKeeper – Recall



ZooKeeper – AUC



ZooKeeper – Kappa



ZooKeeper – analisi risultati

- Osservando i risultati si nota che **SMOTE** permette di avere una **precision** leggermente migliore per i classificatori **IBk** e **NaiveBayes**, mentre per **RandomForest** il **NO sampling** fa ottenere una **precision** leggermente migliore;
- Su tutti e tre i classificatori, **l'oversampling** e soprattutto il **NO sampling** permettono di avere una maggiore **recall**, mentre applicando **SMOTE** per **NaiveBayes** ed **RandomForest** si ha una **recall** più bassa;
- Possiamo notare che con il **NO sampling** si hanno delle performance migliori per i classificatori **NaiveBayes** e soprattutto **RandomForest** per quanto riguarda la metrica **AUC**;
- Anche per quanto riguarda **Kappa** le performance migliori si hanno con il classificatore **RandomForest** con il **NO sampling.**

Conclusioni

- In entrambi i progetti si può osservare come il **NO sampling** permetta di avere delle performance migliori per la maggior parte dei classificatori e per le varie metriche.
- Sia su **BookKeeper** che su **ZooKeeper**, oltre il No sampling, si ha che applicando **Oversampling** si ottengono risultati migliori per tutti e tre i classificatori e per tutte le metriche considerate.
- Per entrambi i progetti si nota che il classificatore che si comporta leggermente meglio è **RandomForest.**
- Dai risultati ottenuti si può notare che tutti i classificatori raggiungono livelli elevati di affidabilità.

Link

GitHub repositories:

- **Milestone 1:** https://github.com/creccovalerio/ISW2-Milestone1
- **Milestone 2:** https://github.com/creccovalerio/ISW2-Milestone2

Sonarcloud:

- **Milestone 1:** https://sonarcloud.io/summary/overall?id=creccovalerio_ISW2-Milestone1
- **Milestone 2:** https://sonarcloud.io/summary/overall?id=creccovalerio_ISW2-Milestone2

Grazie per l’attenzione!