

# Machine Learning for Software Engineering

---

STUDIO SULL'ACCURATEZZA DI MODELLI PREDITTIVI PER LA  
LOCALIZZAZIONE DI BUG NEL CODICE DI AMPIE APPLICAZIONI  
OPEN-SOURCE

# Agenda

---

- Introduzione ed obiettivi
- Progettazione
  - Metodologia ed assunzioni
  - Raccolta dei dati
  - Valutazione dei classificatori
  - Tecniche considerate per il confronto tra i classificatori
- Risultati
- Considerazioni e conclusioni

# Introduzione ed obiettivi

---

- Tutti i processi di sviluppo software prevedono attività di testing, il cui obiettivo è l'individuazione di eventuali malfunzionamenti presenti nel sistema, spesso causati da bug nel codice sorgente.
- Su progetti di dimensioni notevoli, l'effort necessario per l'individuazione dei bug cresce considerevolmente. Il budget a disposizione, sia in termini economici che di tempo, è sempre limitato e non tutto può essere testato.
- Come fare per migliorare l'efficacia del testing? Come fare per scovare più bug possibili?

## **OBIETTIVO n° 1**

Supportare l'attività di testing, indicando quali classi software conviene testare, stimando la probabilità che tali classi contengano dei bug.

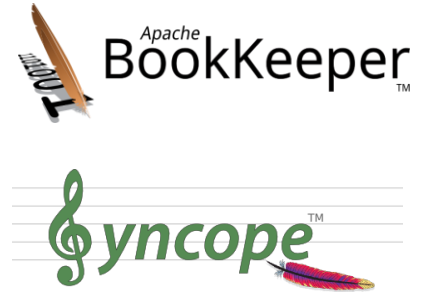
# Introduzione ed obiettivi

---

- Per perseguire l'obiettivo fissato in precedenza, è necessario un **modello predittivo**, in grado di prevedere quali classi hanno maggior probabilità di contenere dei bug.
- Un modello predittivo, in quanto tale, necessita di **dati passati** per effettuare le previsioni. Bisognerà dunque raccogliere tali dati ed effettuare delle misurazioni sul codice e sulla sua evoluzione release per release.
- A questo punto, si potrà fare ricorso a strumenti di Machine Learning, usando i classificatori considerati più opportuni e scegliendo le tecniche ritenute più adatte agli scopi che si intende raggiungere.

# Obiettivo dello studio

- Come caso di studio, sono stati selezionati due progetti open-source di Apache:
  - Apache BookKeeper
  - Apache Syncope
- Sono stati selezionati *Random Forest*, *Naïve Bayes* ed *IBK* come classificatori da analizzare.
- Le tecniche di utilizzo considerate per tali classificatori sono la **feature selection**, il **sampling**, e la **cost sensitivity**. Per non complicare eccessivamente lo studio, due di tali tecniche sono state fissate e sono stati analizzati i risultati ottenuti sperimentando l'utilizzo di diverse varianti della terza tecnica.



## OBIETTIVO n° 2

Individuare quale classificatore ha le migliori performances in termini di accuratezza delle predizioni, al variare delle tecniche di utilizzo, in particolare al variare della tipologia di cost sensitivity.

# Progettazione: metodologia ed assunzioni

---

- Per la raccolta dei dati passati, è stato necessario individuare tutte le releases dei progetti e lo storico dei bug, con le informazioni sul ciclo di vita di questi ultimi: **injected version** (IV), **opening version** (OV) e **fixed version** (FV). A tale scopo, sono state utilizzate le API REST dell'Issue Tracking System *Jira*.
- Sono stati filtrati solo i tickets relativi ad issues di tipo *Bug*, con risoluzione *Fixed* e con stato *Resolved* oppure *Closed*, collezionandoli in una lista.
- La lista di bug è stata ordinata temporalmente e sono stati scartati quei bug per cui una delle seguenti condizioni era vera:
  - La FV è indicata tra le Affected Versions
  - $IV > OV$
- Per i bug che non avevano una IV esplicitamente indicata, è stata usata la tecnica di **Proportion Incremental**, considerando solo i bug precedenti a quello in analisi per il calcolo del fattore di proporzione **p**. Tale scelta è stata dettata dal non voler utilizzare dati futuri per fare delle stime su dati passati, in modo da non influenzare la successiva valutazione dei classificatori, effettuata con la tecnica **Walk Forward**.

# Progettazione: raccolta dei dati

---

- Incrociando i tickets dei bug fixati ottenuti da *Jira* con i commit effettuati nel tempo sui progetti analizzati, ottenuti tramite il Version Control System *Git*, è stato possibile individuare i commit di fix.
- Inoltre, sempre tramite *Git*, sono stati scanditi tutti i commit dei progetti per effettuare le misurazioni necessarie sulle diverse classi software.
- In questo modo, è stato costruito un **dataset** in cui ogni riga (**istanza**) è costituita da una classe software, misurata al termine di una determinata release, caratterizzata dalle **metriche** computate ed **etichettata** con l'attributo *true* se la classe era buggy in quella release, *false* altrimenti.

**ASSUNZIONE:** per effettuare il **labeling** (buggy VS no buggy), si è assunto che tutte le classi toccate dal commit di fix del bug B, siano classificate come buggy in tutte le affected versions del bug B.



# Progettazione: problema dello Snoring

---

- Siamo sicuri che una classe etichettata come “non buggy” in una release X sia effettivamente tale?

**NO!**

- È possibile che la classe, seppur classificata come “non buggy”, avesse uno o più **bug dormienti** nella release X, che si sono manifestati solo successivamente o che addirittura devono ancora manifestarsi → fenomeno dello **Snoring**
- Per releases recenti, l’impatto dello Snoring è molto elevato ed il labeling sarebbe inaffidabile, compromettendo la bontà delle predizioni dei classificatori.
- Si decide, quindi, di sfruttare i dati di tutte le releases a disposizione per la costruzione del dataset, ma la valutazione dei classificatori verrà fatta prendendo in considerazione unicamente il dataset troncato fino alla **prima metà delle releases**, così da limitare gli effetti dello Snoring.



# Progettazione: metriche considerate

Nome metrica	Descrizione
<b>Size</b>	Dimensione in LOC della classe (cumulativa tra le releases)
<b>LOC_touched</b>	Numero di LOC aggiunte e rimosse nella release
<b>LOC_added</b>	Numero di LOC aggiunte nella release
<b>MAX_LOC_added</b>	Numero massimo di LOC aggiunte in una revisione della release
<b>AVG_LOC_added</b>	Numero medio di LOC aggiunte sulle revisioni della release
<b>NR</b>	Numero di revisioni nella specifica release
<b>NAuth</b>	Numero di differenti autori che hanno effettuato una revisione sulla classe nella release
<b>Churn</b>	LOC aggiunte – LOC rimosse nella release
<b>MAX_Churn</b>	Massimo churn di una revisione nella release
<b>AVG_Churn</b>	Churn medio tra le revisioni nella release
<b>NFix</b>	Numero di bug fixati nella classe (cumulativa tra le releases)
<b>Age</b>	Età della classe in settimane (cumulativa tra le releases)
<b>WeightedAge</b>	Età della classe pesata sulle LOC touched (cumulativa tra le releases)

# Progettazione: valutazione dei classificatori

- Come accennato in precedenza, la tecnica di valutazione dei classificatori utilizzata è stata **Walk Forward**. Si tratta di una tecnica di tipo time-series, in cui l'ordine temporale dei dati è un requisito fondamentale.
- Ad ogni run, tutte le releases di training set **NON** devono essere influenzati dalle informazioni ottenibili nelle releases successive: in altre parole, se il training set è composto dalle releases  $1, 2, \dots, n$ , allora le informazioni ottenibili dalla release  $n+1$  in poi non devono in alcun modo influenzarlo.

		RELEASE				
		1	2	3	4	5
RUN	1	X				
	2					
	3					
	4					
	5					

Legenda:



Training



Testing

# Progettazione: tecniche considerate per i classificatori

---

- Ottenuti i dataset e stabilita la tecnica di valutazione per i classificatori *Random Forest*, *Naive Bayes* e *IBK*, si è proceduto con la loro valutazione.
- In particolare, è stata osservata l'accuratezza dei diversi classificatori, mantenendo fisse le tecniche di feature selection e sampling, mentre sono state considerate le **diverse varianti della tecnica di cost sensitivity**.
- **Feature selection**: è stato applicato l'algoritmo greedy ***Backward Search***, poiché il numero delle metriche considerate nel dataset non è così elevato da causare un severo degrado delle prestazioni rispetto a *Forward Search*. Inoltre, l'obiettivo era quello di eliminare metriche superflue, piuttosto che restringere il più possibile il pool delle metriche considerate.
- **Sampling**: per BookKeeper, le istanze buggy sono risultate essere il 12.98% delle totali, mentre, per Syncope, esse ammontano al 16.03% del totale. Si è scelto di applicare ***Undersampling*** per bilanciare il dataset.
- **Cost sensitivity**: tre scenari → *No Cost Sensitivity*, *Sensitive Threshold*, *Sensitive Learning*

# Progettazione: varianti di Cost Sensitivity

- Se non viene applicata **nessuna tecnica di Cost Sensitivity**, una previsione errata ha sempre lo stesso peso, a prescindere se si sia ottenuto un falso positivo (FP) o un falso negativo (FN).
- Con **Sensitive Threshold**, si usa una matrice dei costi per ricalcolare la probability threshold usata dai classificatori; tale valore viene usato per definire quale dei due esiti predirre (buggy / non buggy).
- Con **Sensitive Learning**, si usa la medesima matrice dei costi ( $CFN = 10 \cdot CFP$ ) di *Sensitive Threshold*, ma, anziché cambiare il valore della probability threshold, si attua internamente una sorta di learning di un nuovo classificatore, con le istanze del dataset riconsiderate con il peso opportuno.
- Per l'applicazione di tali tecniche e la valutazione dei classificatori, sono state utilizzate le *Weka API*.



CTP = 0	CFN = 10
CFP = 1	CTN = 0

Matrice dei costi

# Risultati

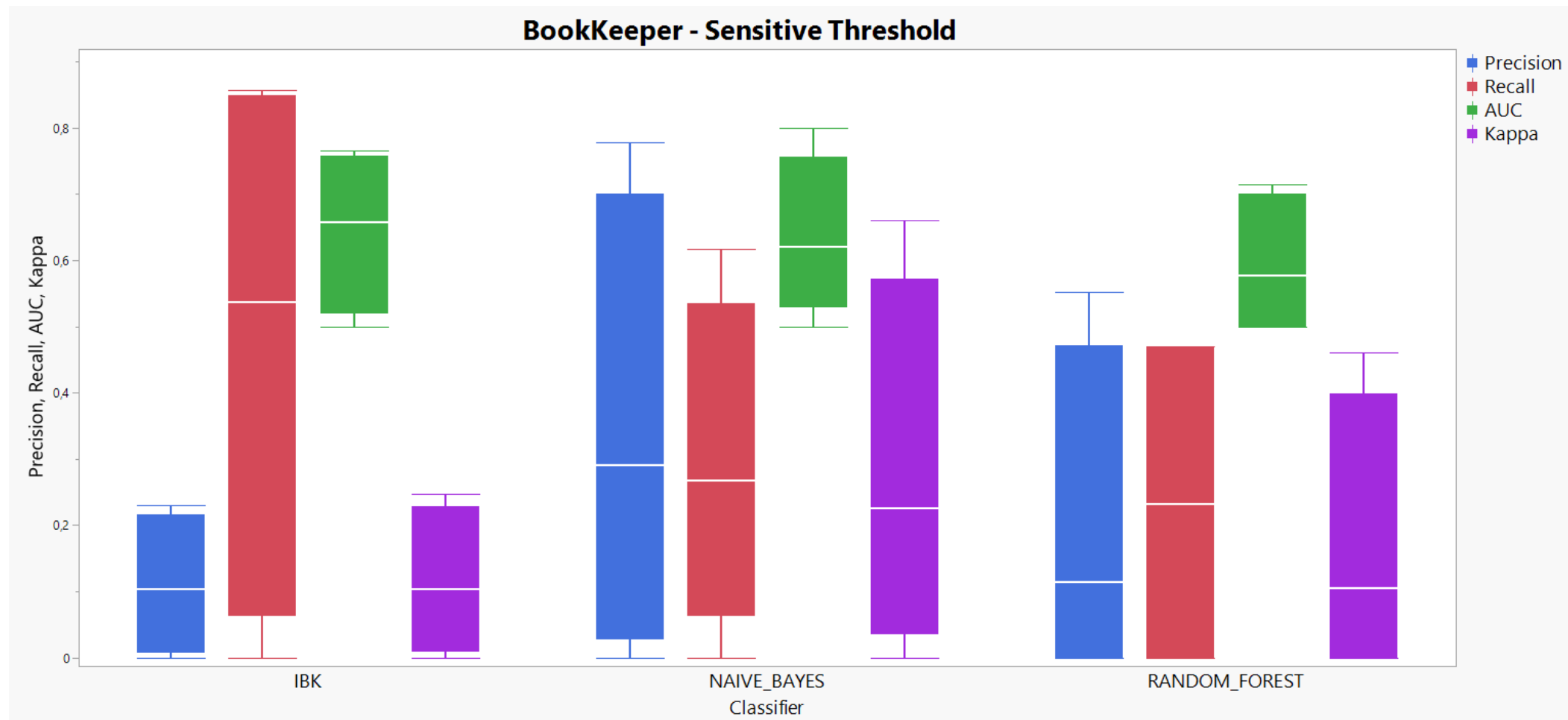
---

- Nelle slides seguenti, verranno mostrati dei box plot raffiguranti i risultati ottenuti dalla valutazione dei tre classificatori, sia per il progetto *BookKeeper*, che per il progetto *Syncope*.
- Il confronto verrà fatto sempre a parità di tecniche utilizzate.
- Le metriche prese in considerazione per valutare i classificatori sono:
  - *Precision*
  - *Recall*
  - *AUC (Area Under ROC Curve)*
  - *Kappa*

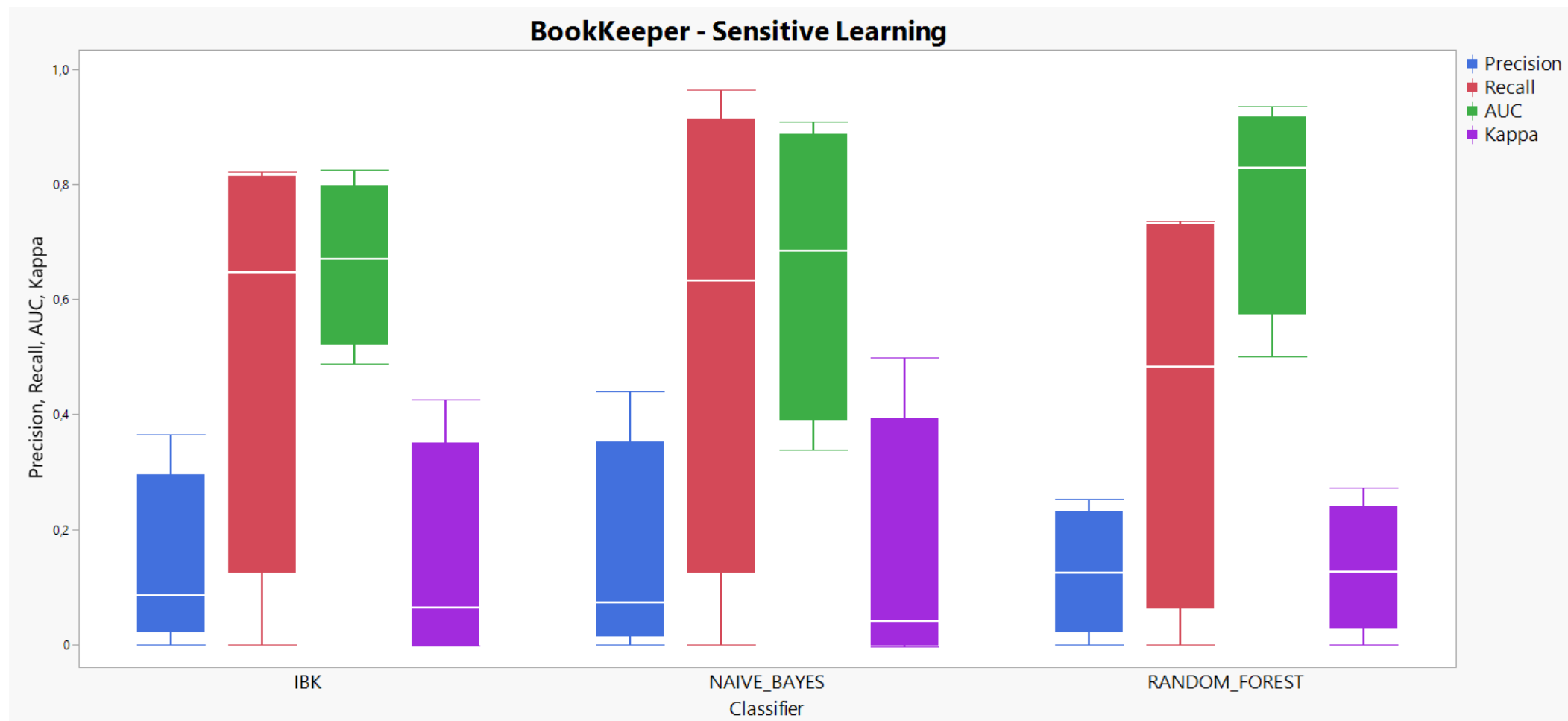
# Risultati: BookKeeper – No Cost Sensitivity



# Risultati: BookKeeper – Sensitive Threshold



# Risultati: BookKeeper – Sensitive Learning



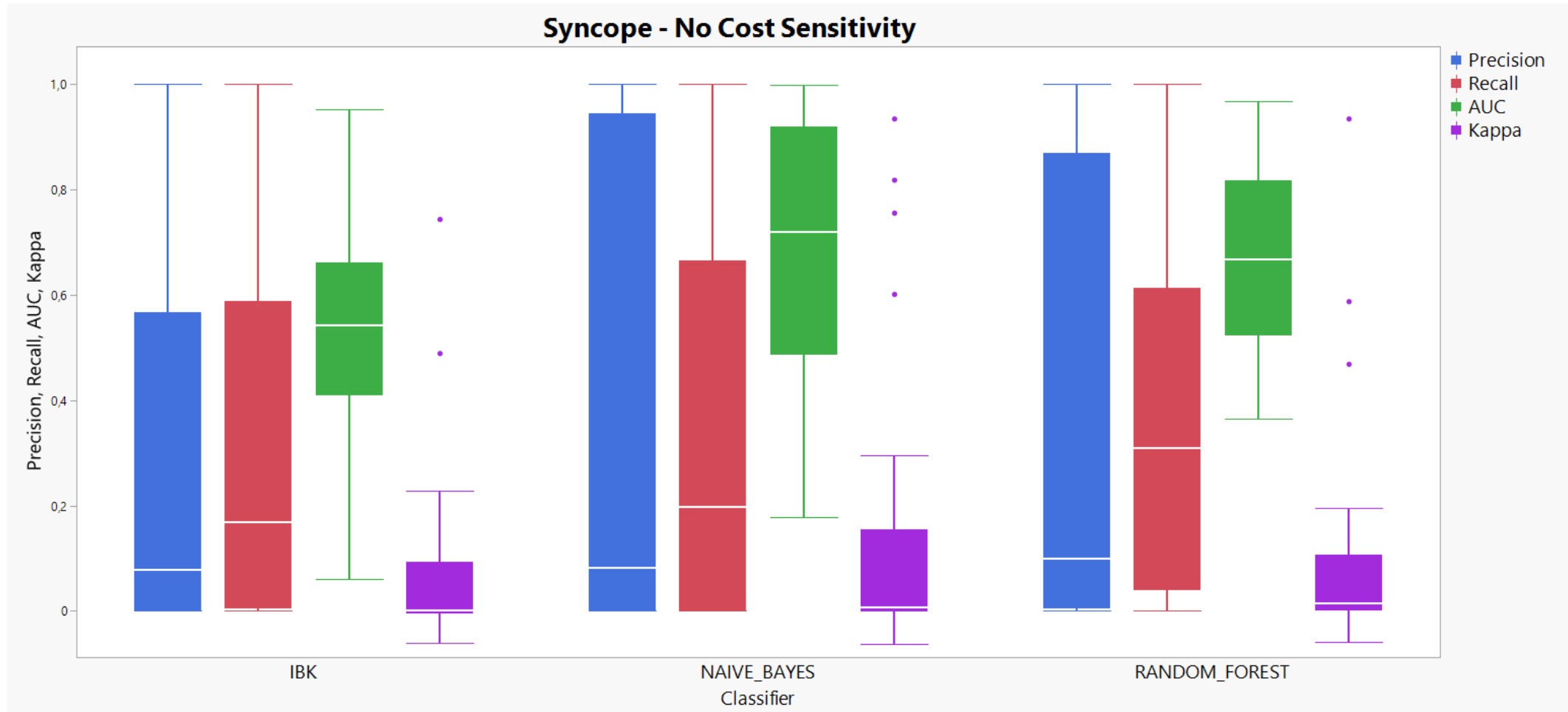


# Risultati: BookKeeper - considerazioni

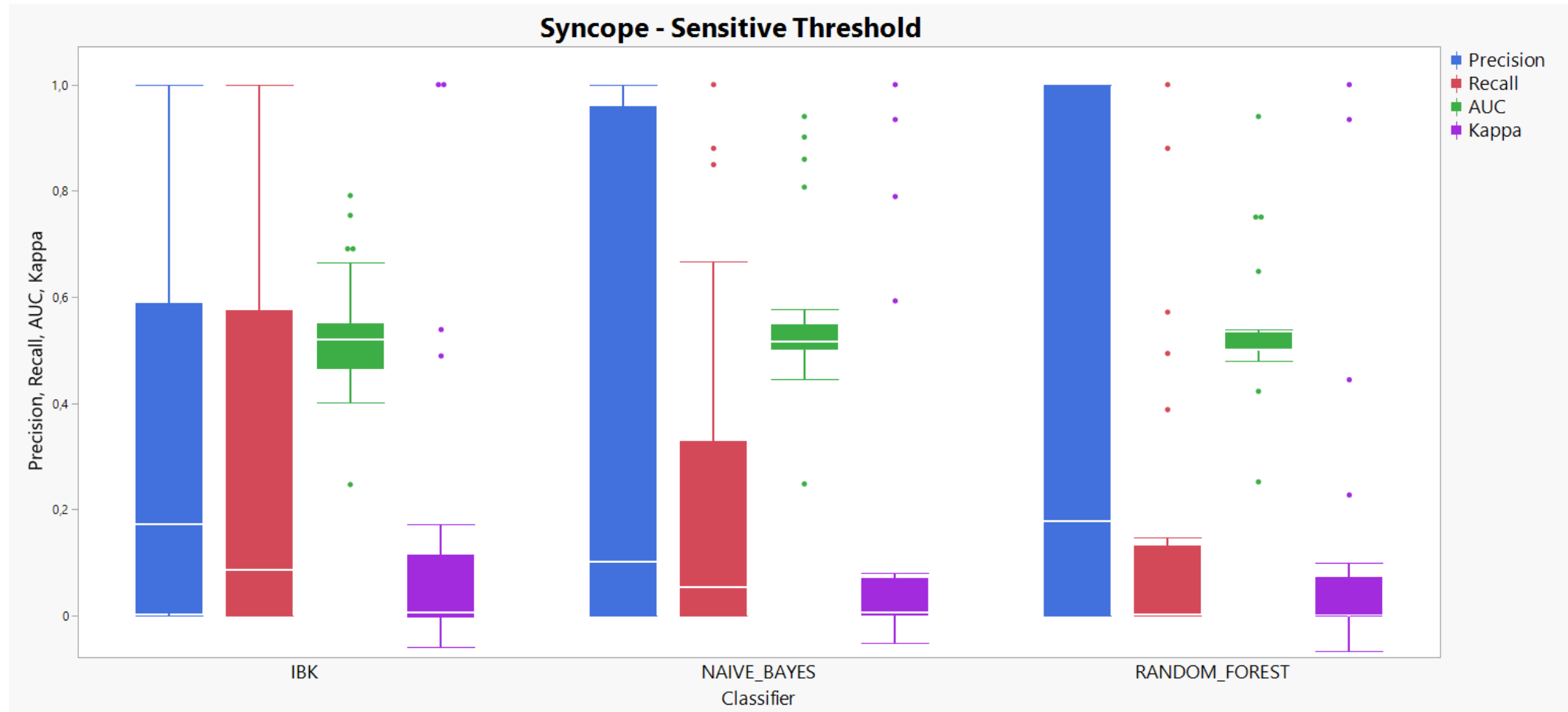
---

- **PREMESSA:** BookKeeper ha poche releases, dunque, il processo di validazione con Walk Forward è risultato nell'esecuzione di sole 5 run. Di conseguenza, le distribuzioni rappresentate dai box plot raffigurano un campione di dimensioni limitate, composto da pochi sample points.
- **No Cost Sensitivity:** tutti i classificatori hanno Recall comparabile, ma *Naive Bayes* ha la precisione più elevata, considerando la mediana. Inoltre, anche Kappa è il migliore, mentre AUC risulta essere leggermente inferiore a quella di *Random Forest*. *IBK* è sicuramente il classificatore peggiore, mentre il migliore è **Naive Bayes**.
- **Sensitive Threshold:** rispetto a *No Cost Sensitivity*, non si ha un netto miglioramento in generale. Si nota un particolare decremento della Recall per *Random Forest*, mentre *Naive Bayes* continua ad avere la *Precision* migliore. Sebbene *Naive Bayes* e *IBK* siano assolutamente comparabili, si preferisce ancora **Naive Bayes**, in quanto ha un valore mediano di Kappa superiore.
- **Sensitive Learning:** i classificatori prevedono molto più spesso positivi (buggy); ciò è visibile in un complessivo aumento della Recall e nella contestuale diminuzione della Precision. Considerando anche Kappa ed AUC, stavolta il classificatore migliore è senza dubbio **Random Forest**.

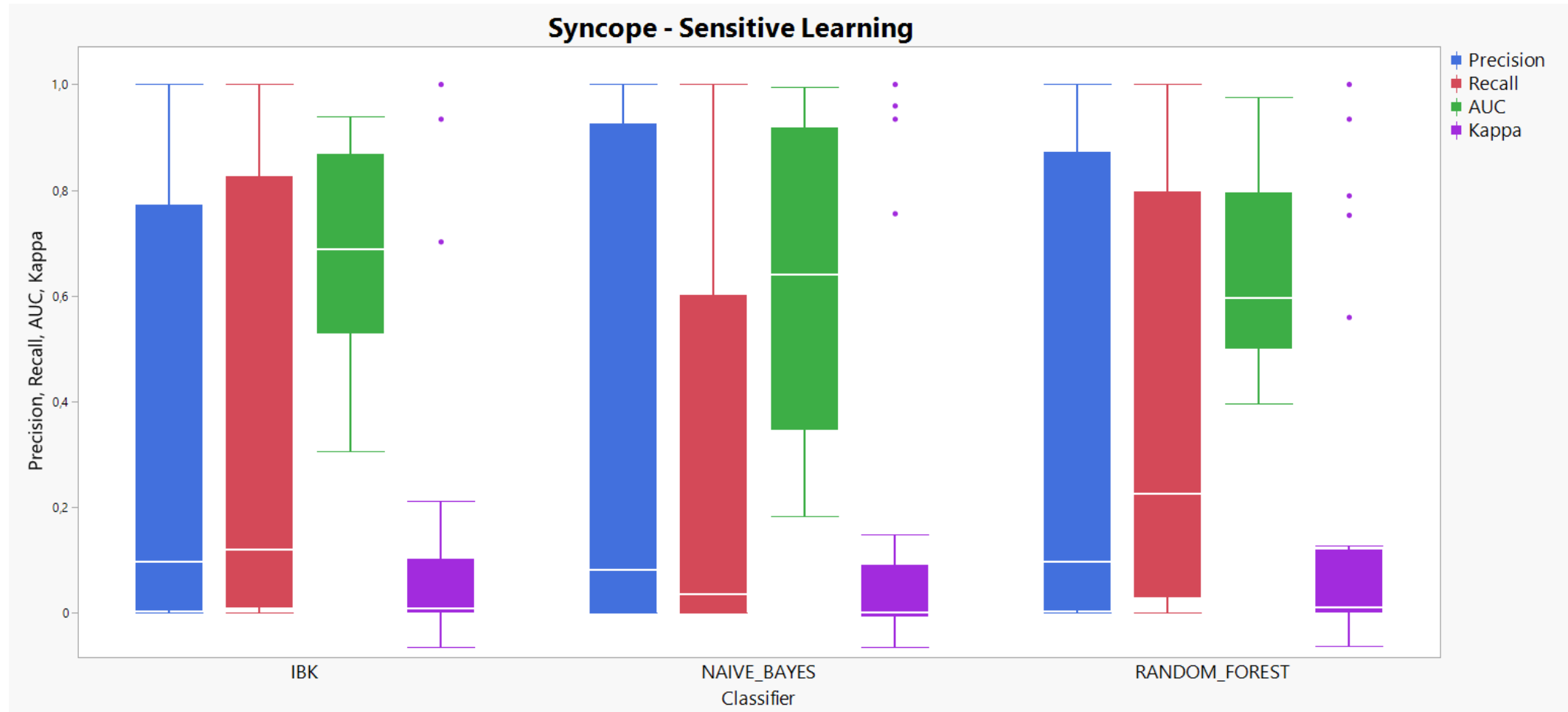
# Risultati: Syncope – No Cost Sensitivity



# Risultati: Syncope – Sensitive Threshold



# Risultati: Syncope – Sensitive Learning

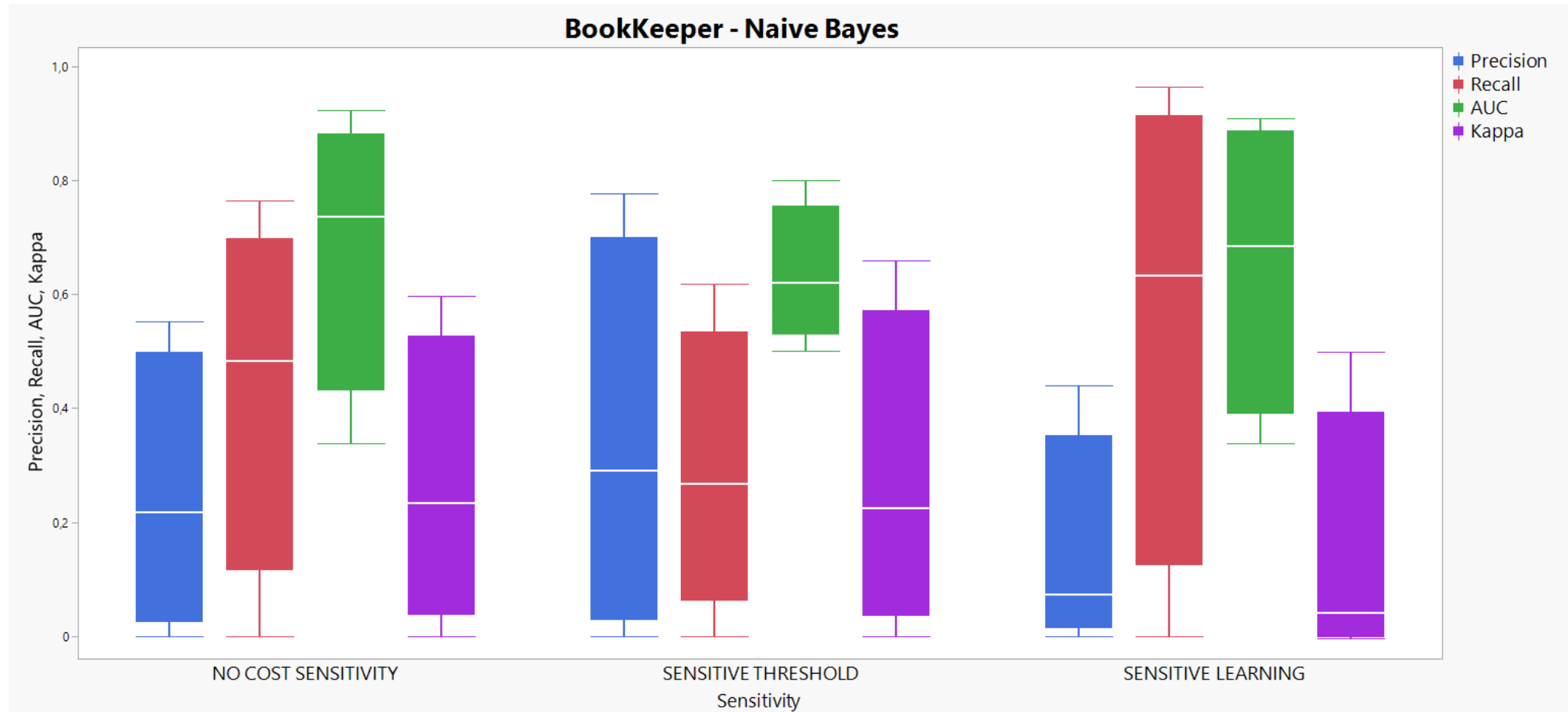


# Risultati: Syncope - considerazioni

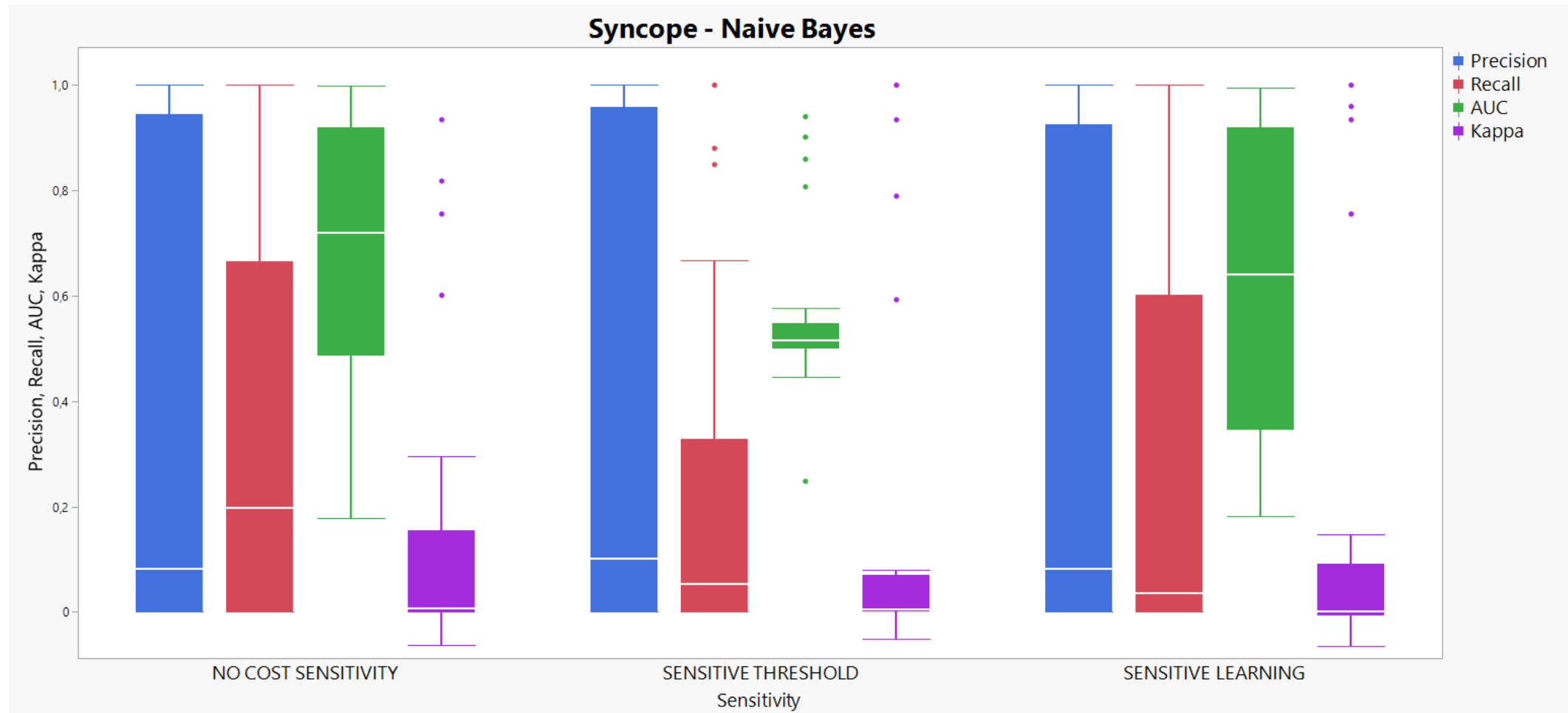
---

- **No Cost Sensitivity:** dal grafico è evidente che il classificatore peggiore sia *IBK*. La mediana della Precision dei tre classificatori coincide, ma a livello di distribuzione, quella di *Naive Bayes* è migliore, così come per l'AUC. Il Kappa è molto basso per tutti e tre i classificatori. Il migliore è **Naive Bayes**.
- **Sensitive Threshold:** si ha un inaspettato calo della Recall sia per *Naive Bayes* che per *Random Forest*, in cambio di un lieve miglioramento della Precision. Ma, complessivamente, sia AUC che Kappa calano di molto, registrando un peggioramento generale per tutti e tre i classificatori. È difficile affermare quale dei tre sia il migliore.
- **Sensitive Learning:** come atteso, c'è un aumento della Recall, anche se la mediana resta comunque abbastanza bassa. La Precision dei tre classificatori è simile, ma quello che ottiene miglioramenti sensibili grazie a questa politica di cost sensitivity è senza dubbio **IBK**, che risulta essere il migliore dei tre.

# Risultati: BookKeeper – cost sensitivity con Naive Bayes



# Risultati: Syncope – cost sensitivity con Naive Bayes



# Considerazioni e conclusioni

---

- Scelto **Naive Bayes** come classificatore, è stata valutata la tecnica di *cost sensitivity* più efficiente.
- Per *BookKeeper*, la Precision e il Kappa maggiori si ottengono con *Sensitive Threshold*, ma le stesse metriche ottenute con *No Cost Sensitivity* presentano andamenti molto simili. Si predilige quest'ultima tecnica poiché è quella tra le due che evidenzia un AUC migliore.
- Per *Syncope*, c'è da dire che in tutti gli scenari il valore di Kappa è molto prossimo allo zero, sintomo che il classificatore si comporta quasi come un classificatore "dummy" (e.g. ZeroR). Le prestazioni ottenute dall'applicazione di *Sensitive Learning* sono comparabili a quelle ottenute senza applicare alcuna tecnica di *cost sensitivity*. Tuttavia, questo secondo scenario evidenzia un valore mediano di AUC preferibile ed un Kappa leggermente maggiore, pertanto è migliore.
- In conclusione, sia per *Syncope* che per *BookKeeper*, conviene, in generale, **NON applicare alcuna tecnica di cost sensitivity**, se si applica già *feature selection* di tipo *Backward Search* e bilanciamento di tipo *Undersampling*.



# Riferimenti

---

- Repository GitHub: <https://github.com/AndreaPepe/MLforSE>
- SonarCloud: [https://sonarcloud.io/summary/overall?id=AndreaPepe\\_SyncopeDataMining](https://sonarcloud.io/summary/overall?id=AndreaPepe_SyncopeDataMining)