

# Machine Learning nell'Ingegneria del Software

---

Come prevedere i bug?

# Contenuti

---

- Contesto
- Obiettivi
- Metodologie:
  - schema di processo generale
  - collezione delle metriche da Git
  - identificazione di coppie classe/versione con bug
  - condizioni su Jira
  - calcolo dell'IV tramite proportion
  - costruzione del dataset
  - riduzione dello snoring
  - selezione del training e del testing set
- Analisi dei risultati
- Conclusioni
- Riferimenti

# Contesto

---

- Testare gli artefatti software risulta un'attività estremamente dispendiosa in termini di risorse umane, tecnologiche e finanziarie.
- Al tempo stesso però la presenza di bug nei software comporta per le aziende costi elevatissimi.
- Questi bug potrebbero essere stati introdotti al fine di risolvere bug precedenti, aggiungere nuove funzionalità ai sistemi coinvolti o per un cambiamento apportato alle tecnologie correntemente adottate.
- Per evitare ciò è necessario testare adeguatamente il codice, ma testare tutto non è possibile. Da qui sorge il problema: **come decidere cosa testare?**

# Obiettivi

---

- **IDEA:** applicare i meccanismi propri del Machine Learning alla predizione dei difetti nelle classi software.
- **OBIETTIVO:** individuare quali classi saranno più propense in futuro a presentare dei bug.
- In questo modo sarà possibile, sfruttando le informazioni della predizione, direzionare le attività di testing verso tali classi, minimizzando la quantità di test da effettuare, e quindi i rispettivi costi, e massimizzando l'efficacia nell'individuazione dei bug.
- Verranno utilizzati dei classificatori esistenti, le cui capacità di predizione saranno confrontate mediante una selezione di metriche e tecniche di valutazione.

# Metodologie: utilizzo di progetti open source

---

- I classificatori verranno confrontati su due dataset costruiti a partire dalle informazioni ricavabili su due progetti open source di Apache Software Foundation:

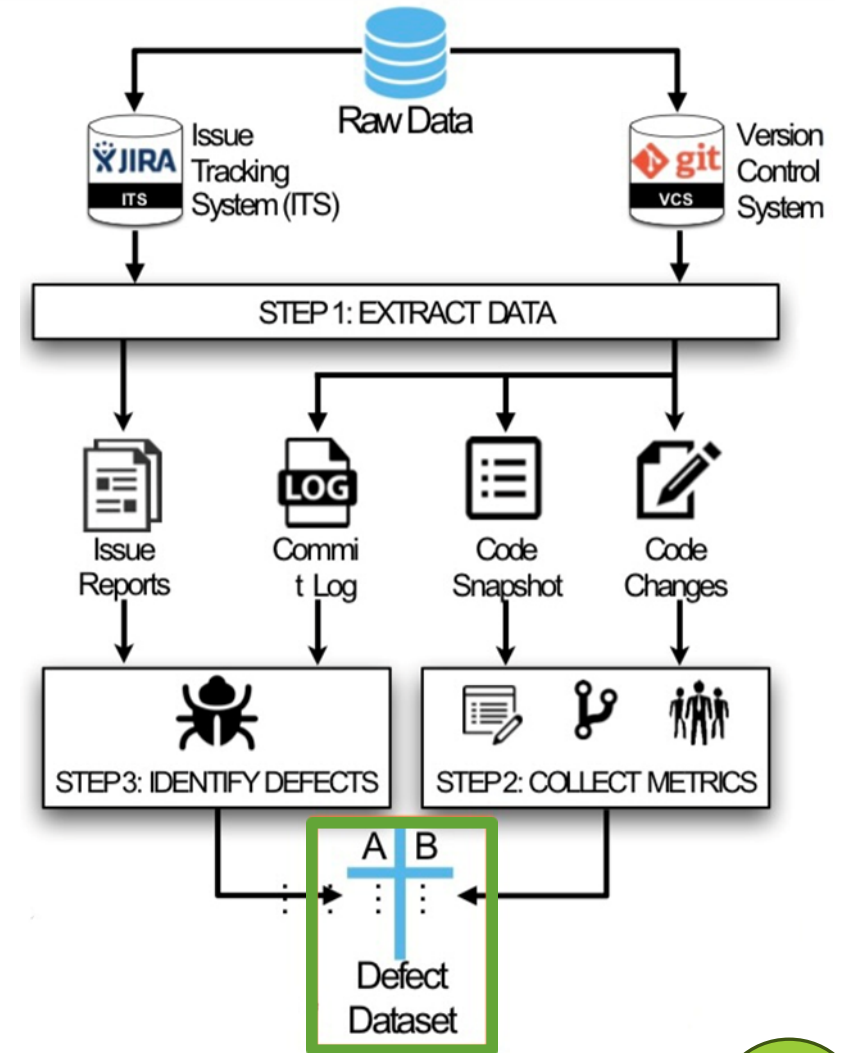


- Questa scelta è dovuta all'utilizzo per questi progetti di **Git** come sistema di controllo delle versioni e **Jira** come software per il monitoraggio di ticket, sistemi di cui avremo bisogno per la costruzione del dataset.



# Metodologie: schema di processo generale

- In primo luogo è necessario costruire un dataset che sarà dato in input ai classificatori. Questo processo consta di diverse fasi:
  1. realizzazione di un programma per automatizzare l'estrazione di dati sui progetti da Git e Jira
  2. Collezione di metriche ricavabili da Git riguardanti il codice e il processo di sviluppo del software
  3. Identificazione tramite Jira delle coppie classe/versione contenenti dei bug.



# Metodologie: collezione delle metriche da Git

- Per ogni classe in una specifica versione vengono raccolte le seguenti metriche:


| Metrica       | Descrizione                                  |
|---------------|--|
| LOC           | Linee di codice                              |
| LOC Touched   | Somma sui commit di linee aggiunte + rimosse |
| NR            | Numero di commit                             |
| Nfix          | Numero di bug risolti                        |
| Nauth         | Numero di autori                             |
| Loc Added     | Somma sui commit di linee aggiunte           |
| Max Loc Added | Massimo tra i commit di linee aggiunte       |
| Churn         | Somma sui commit di linee aggiunte - rimosse |
| Max Churn     | Massimo churn tra i commit                   |
| Average Churn | Churn medio tra i commit                     |

Per questo scopo è stata usata la libreria Java denominata **JGit**




# Metodologie: identificazione di coppie classe/versione con bug

- Si considerano su Jira i ticket per cui vale la seguente condizione: *Type == "defect" AND (status == "Closed" OR status == "Resolved") AND Resolution == "Fixed"*.

 Bookkeeper / **BOOKKEEPER-1031** ← **Ticket id** 18 of 435 ^ v

**ReplicationWorker.rereplicate fails to call close() on ReadOnlyLedgerHandle**

 Export v

▼ Details

Type: **Bug**

Priority: **Major**

Affects Version/s: **4.4.0** ← **Injected version, IV**

Component/s: bookkeeper-auto-recovery

Labels: None

Status: **RESOLVED**

Resolution: **Fixed**

Fix Version/s: 4.5.0

▼ People

Assignee: Samuel Just

Reporter: Samuel Just

Votes: 0 Vote for this issue

Watchers: 5 Start watching this issue

**Opening version, OV**

▼ Dates

Created: **10/Apr/17 18:57**

Updated: 12/Apr/17 13:39

Resolved: **11/Apr/17 18:14** ← **Fix version, FV**

▼ Description

This has the effect of permanently adding 1 listener per call into AbstractZkLedgerManager.listenerSet

▼ Issue Links



# Metodologie: condizioni su Jira

---

- Nella selezione dei ticket da considerare vengono rispettate le seguenti condizioni:

1. I ticket devono avere un commit di fix associato su Git

```
HP@LAPTOP-76KHBPGU MINGW64 ~/Desktop/bookkeeper (master)
$ git log --grep="BOOKKEEPER-1031:"
commit 48aa69dd0ba41f5ba7bb2b04f31172c919be4391
Author: Samuel Just <sjust@salesforce.com>
Date: Tue Apr 11 11:14:19 2017 -0700
```

Commit ID

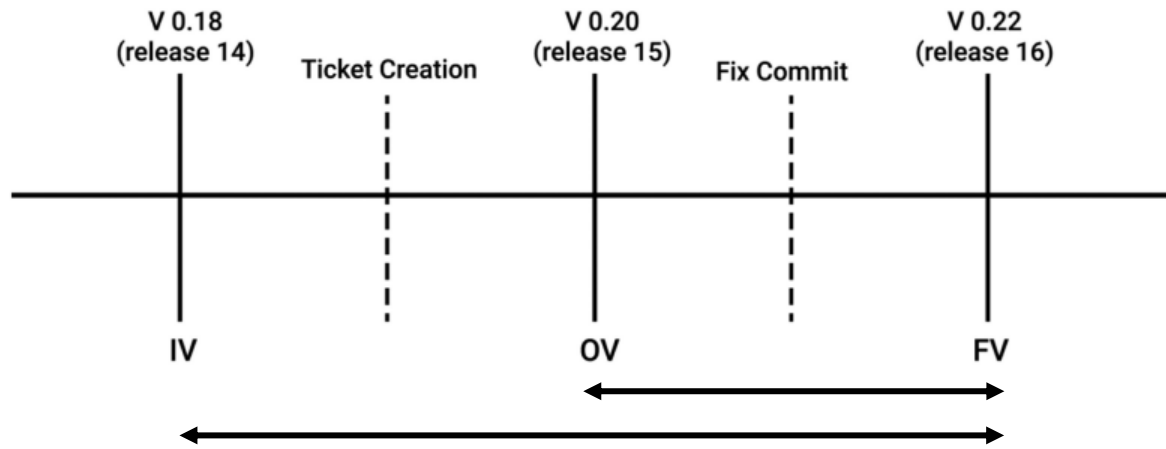
```
BOOKKEEPER-1031: close the ledger handle in ReplicationWorker.rereplicate
```

2. Si escludono i difetti che non sono post-release, ossia deve valere **IV != FV**
3. IV e OV devono essere **consistenti**, ossia **IV <= OV**. In questo caso il ticket non viene scartato ma si considera senza IV indicata.

# Metodologie: calcolo dell'IV tramite proportion

- In molti casi l'indicazione sull'**IV non è presente** su Jira, oppure non è valida. Per calcolarla si utilizza un approccio denominato **Proportion**<sup>1</sup> basato sulla seguente intuizione:

c'è una **proporzione stabile**, tra i difetti di uno stesso progetto, tra il numero di versioni che intercorrono tra IV e FV e il numero di versioni tra OV e FV.



$$P = \frac{(FV - IV)}{(FV - OV)} \rightarrow IV = FV - (FV - OV) * P$$

Nel nostro caso si è utilizzato **Proportion Incremental**, calcolando, per un certo bug, P come media sui P dei difetti precedenti dello stesso progetto. Se tali difetti erano in numero < 5 si è usato **Cold Start**, calcolando P come mediana tra i P medi di altri progetti.

# Metodologie: costruzione del dataset

- A questo punto sono noti per ogni difetto IV e FV, con informazioni sulle classi modificate dal commit di fix su Git, perciò è possibile etichettare ogni classe come **buggy** oppure no in una certa versione del progetto. Si ottiene un dataset con le seguenti informazioni:

| Version | File Name   | LOC | LOC_touched | NR | NFix | NAuth | LOC_added | MAX_LOC_added | Churn | MAX_Churn | AVG_Churn | Buggy |
|---------|---|-----|-------------|----|------|-------|-----------|---------------|-------|-----------|-----------|-------|
| 1       | hedwig-server/src/main/java/org/apache/hedwig/se... | 35  | 37          | 2  | 0    | 2     | 36        | 35            | 35    | 35        | 17.5      | No    |
| 1       | hedwig-client/src/main/java/org/apache/hedwig/cl... | 231 | 255         | 5  | 0    | 2     | 243       | 224           | 231   | 224       | 46.2      | Yes   |
| 1       | bookkeeper-server/src/main/java/org/apache/bookk... | 92  | 1576        | 6  | 0    | 3     | 834       | 763           | 92    | 763       | 15.333333 | No    |
| 1       | bookkeeper-server/src/main/java/org/apache/bookk... | 50  | 56          | 2  | 0    | 2     | 53        | 50            | 50    | 50        | 25.0      | No    |
| 1       | hedwig-server/src/main/java/org/apache/hedwig/se... | 29  | 31          | 2  | 0    | 2     | 30        | 29            | 29    | 29        | 14.5      | No    |
| 1       | hedwig-server/src/main/java/org/apache/hedwig/se... | 37  | 43          | 2  | 0    | 2     | 40        | 37            | 37    | 37        | 18.5      | No    |
| 1       | bookkeeper-server/src/main/java/org/apache/bookk... | 180 | 210         | 4  | 0    | 2     | 195       | 168           | 180   | 168       | 45.0      | No    |
| 1       | bookkeeper-server/src/main/java/org/apache/bookk... | 147 | 147         | 1  | 0    | 1     | 147       | 147           | 147   | 147       | 147.0     | No    |
| 1       | hedwig-server/src/main/java/org/apache/hedwig/zo... | 98  | 100         | 2  | 0    | 2     | 99        | 98            | 98    | 98        | 49.0      | No    |
| 1       | hedwig-client/src/main/java/org/apache/hedwig/cl... | 149 | 155         | 2  | 0    | 2     | 152       | 149           | 149   | 149       | 74.5      | Yes   |
| 1       | hedwig-server/src/main/java/org/apache/hedwig/se... | 45  | 45          | 1  | 0    | 1     | 45        | 45            | 45    | 45        | 45.0      | No    |
| 1       | hedwig-server/src/main/java/org/apache/hedwig/se... | 105 | 191         | 5  | 0    | 3     | 148       | 103           | 105   | 103       | 21.0      | Yes   |
| 1       | hedwig-server/src/main/java/org/apache/hedwig/se... | 229 | 331         | 3  | 0    | 2     | 279       | 226           | 227   | 226       | 75.666664 | No    |

# Metodologie: riduzione dello snoring

---

- Un altro problema che si incontra durante il labeling delle classi è lo **snoring**<sup>2</sup>, dovuto alla presenza dei **bug dormienti**, ossia difetti che vengono scoperti (OV) diverse release dopo la loro introduzione (IV).
- Una classe si definisce affetta da snoring se:
  1. Contiene **almeno un difetto dormiente**
  2. Non presenta **nessun difetto non dormiente**
- Lo snoring porta quindi ad etichettare una classe come non buggy quando invece lo è, introducendo degli errori nel dataset. Poiché tale fenomeno è diffuso in misura maggiore nelle release più recenti, è stato dimostrato empiricamente che eliminare queste release comporta un miglioramento nelle performance dei classificatori.

Nei progetti considerati sono state **eliminate metà delle release** per ridurre lo snoring.

# Metodologie: selezione del training e del testing set (1/2)

- Il dataset costruito è caratterizzato da **dati sensibili al tempo**, le classi sono buggy in specifiche release, ordinate temporalmente.
- Per valutare le capacità predittive dei classificatori, non sarebbe realistico testarli su dati antecedenti a quelli usati per l'addestramento. Quindi nel **suddividere il dataset in training e testing set è necessario preservare l'ordine temporale dei dati**.
- A tale scopo viene adottata una tecnica di validazione time-series: **Walk-Forward**<sup>3</sup>.

Il dataset viene diviso in parti, scelte come le più piccole **unità ordinabili temporalmente**, come le **release**. Se diviso in n parti si eseguono n run, in ognuna delle quali i dati in una unità/release sono usati come **testing set**, e tutti i dati precedenti temporalmente come **training set**.

| Run | Part |   |   |   |   |
|-----|------|---|---|---|---|
|     | 1    | 2 | 3 | 4 | 5 |
| 1   | ■    |   |   |   |   |
| 2   | ■    | ■ |   |   |   |
| 3   | ■    | ■ | ■ |   |   |
| 4   | ■    | ■ | ■ | ■ |   |
| 5   | ■    | ■ | ■ | ■ | ■ |

Testing  
Training

# Metodologie: selezione del training e del testing set (2/2)

- Per rendere i risultati della valutazione dei classificatori più attendibili, nel labeling delle classi si è tenuto conto delle seguenti considerazioni:

1. il **training set** deve essere **realistico**, perciò al momento dell'*n*-esima iterazione del walk-forward, che considera le prime *n* release nel training set, le classi in esse vengono etichettate sulla base dei ticket noti al momento dell'*n*-esima release.



|              |             |                    |
|--------------|-------------|--------------------|
| Training set | Testing set | Release successive |
|--------------|-------------|--------------------|

2. Il **testing set** deve essere **accurato**, quindi le classi vengono etichettate usando tutte le informazioni disponibili, e quindi considerando tutti i ticket delle release esistenti, anche quelle successive.

La **macchina fotografica** rappresenta il **punto di osservazione** rispetto al quale vengono considerati i ticket con cui etichettare le classi.



|              |             |                    |
|--------------|-------------|--------------------|
| Training set | Testing set | Release successive |
|--------------|-------------|--------------------|

# Metodologie: confronto tra classificatori

---

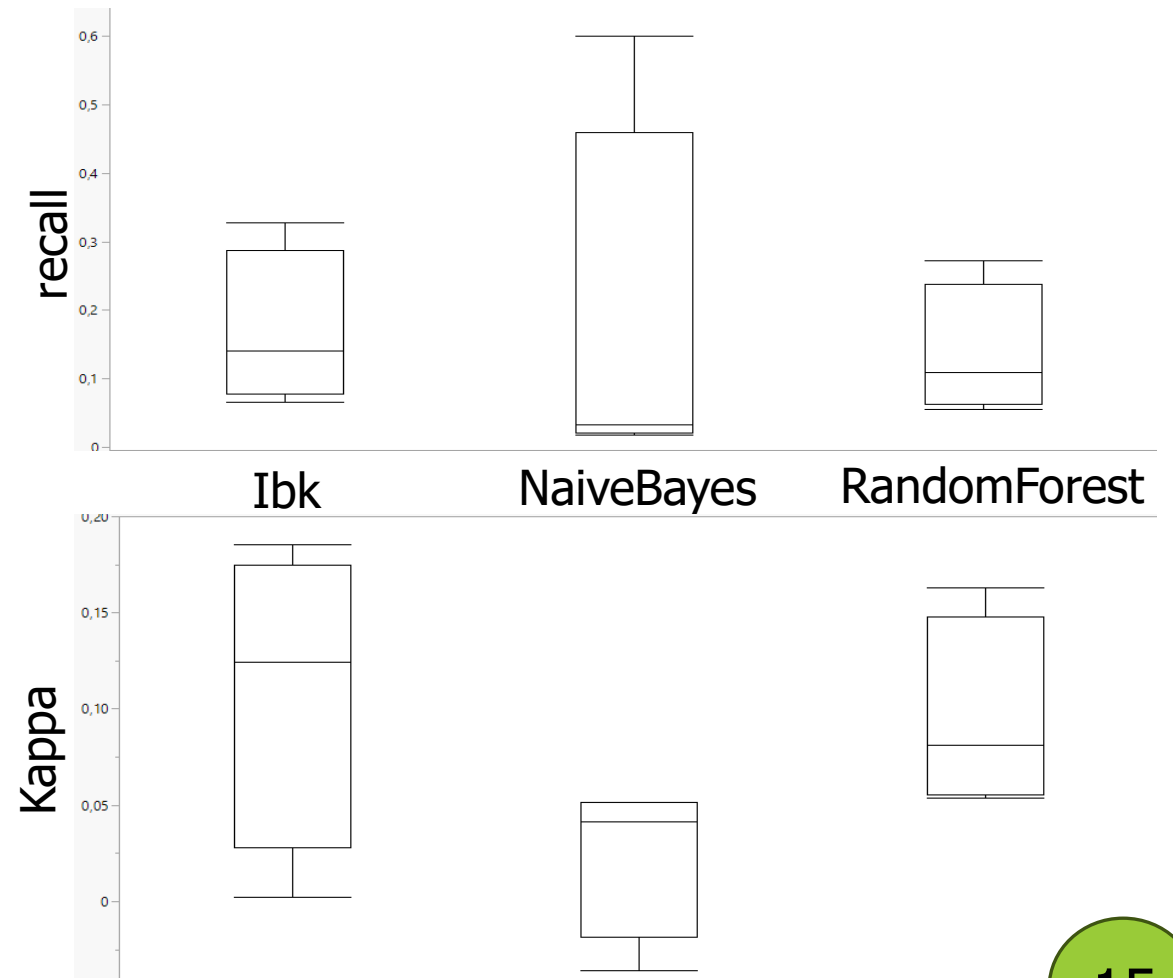
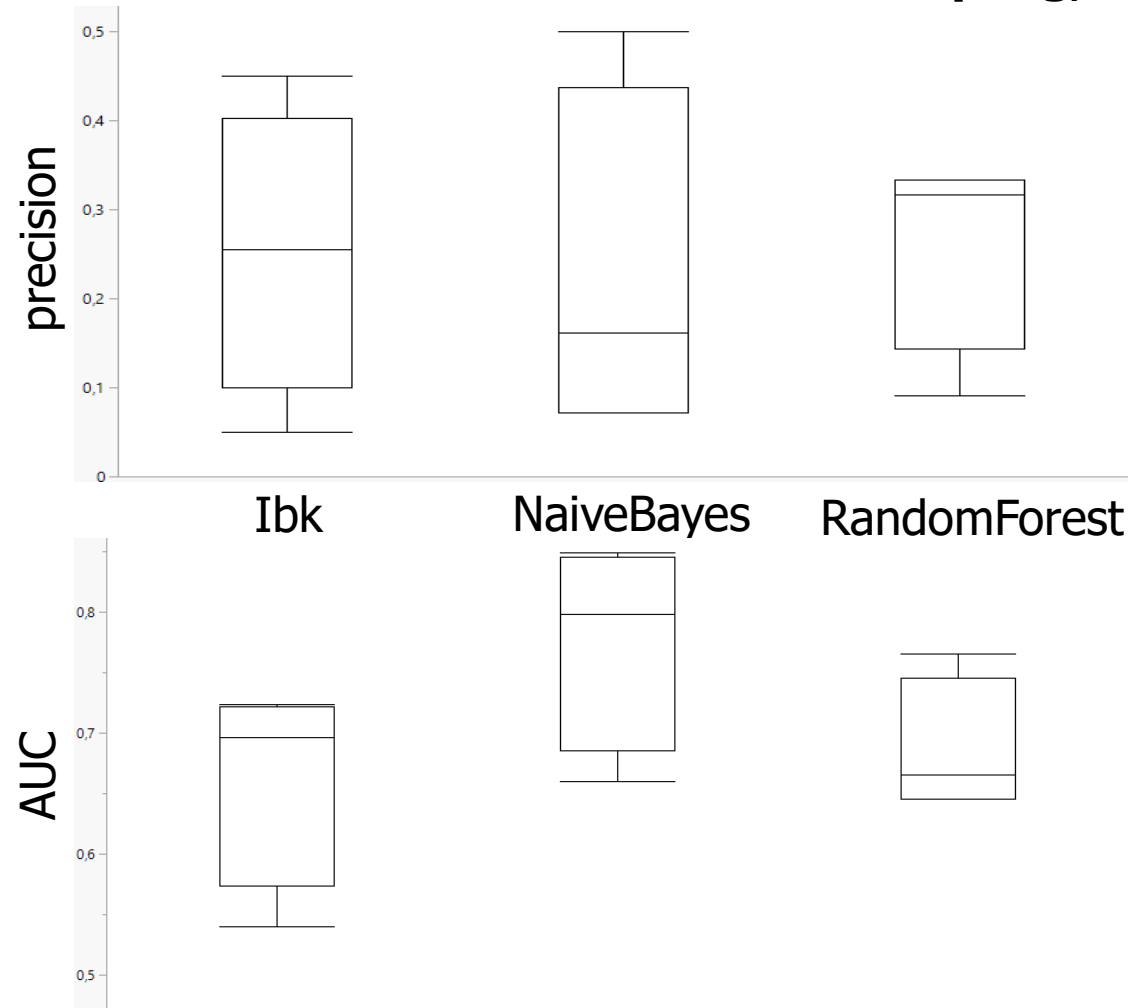
- Vengono confrontate le performance predittive di tre classificatori: **Naive Bayes, Ibk, Random Forest.**
- Sono applicate diverse strategie nel tentativo di migliorare le valutazioni:
  - 1. Feature selection**, in particolare approccio filter con ricerca forward
  - 2. Sampling**, nei diversi casi di **undersampling, oversampling** e **SMOTE**
  - 3. Cost sensitive classifier.**
- I confronti vengono realizzati sulla base delle seguenti metriche: **Precision, Recall, AUC, Kappa.**

Nei riquadri come questo verranno indicati i comportamenti migliori per ogni metrica

# Analisi dei risultati: BookKeeper(1/5)

Random forest → precision  
Ibk → recall (simile a random forest), Kappa  
NaiveBayes → AUC

- Valutazione dei classificatori **senza sampling, feature selection o cost sensitive**:

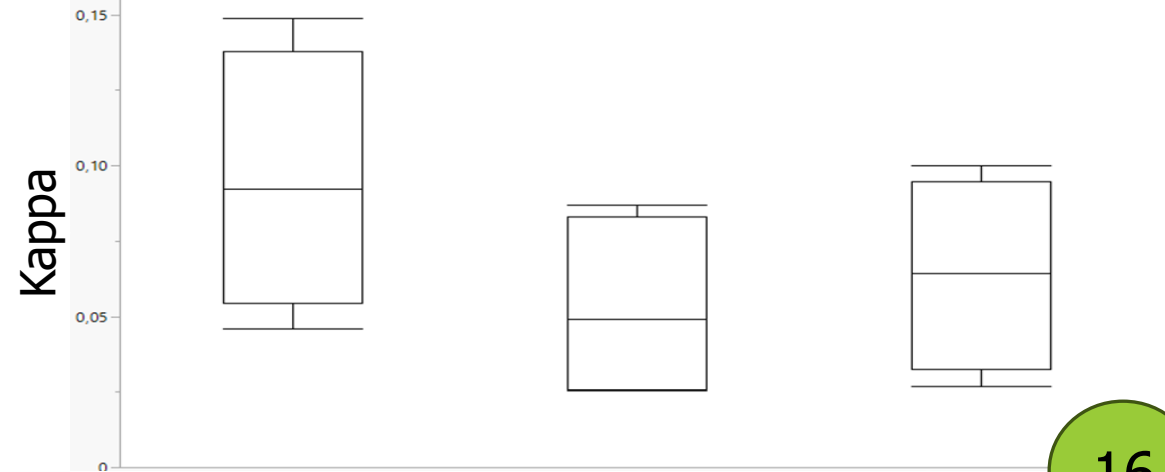
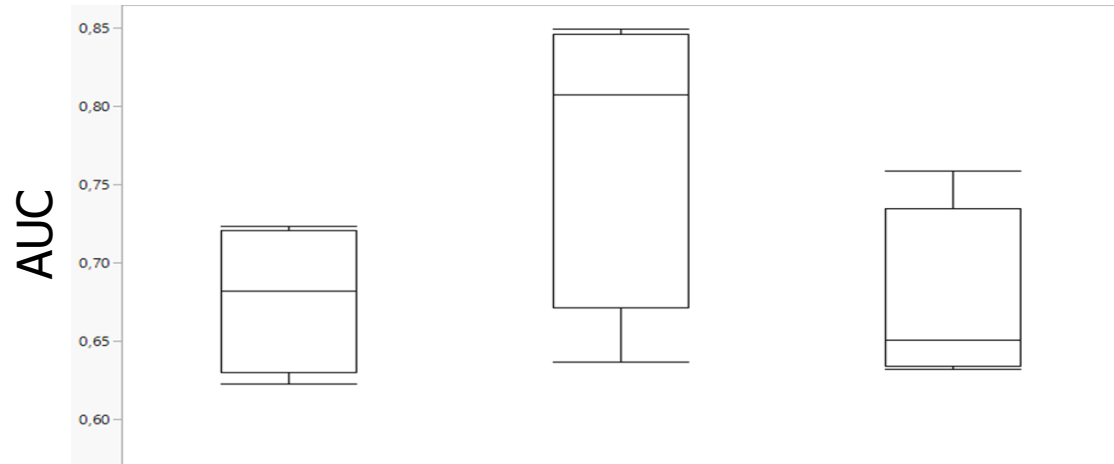
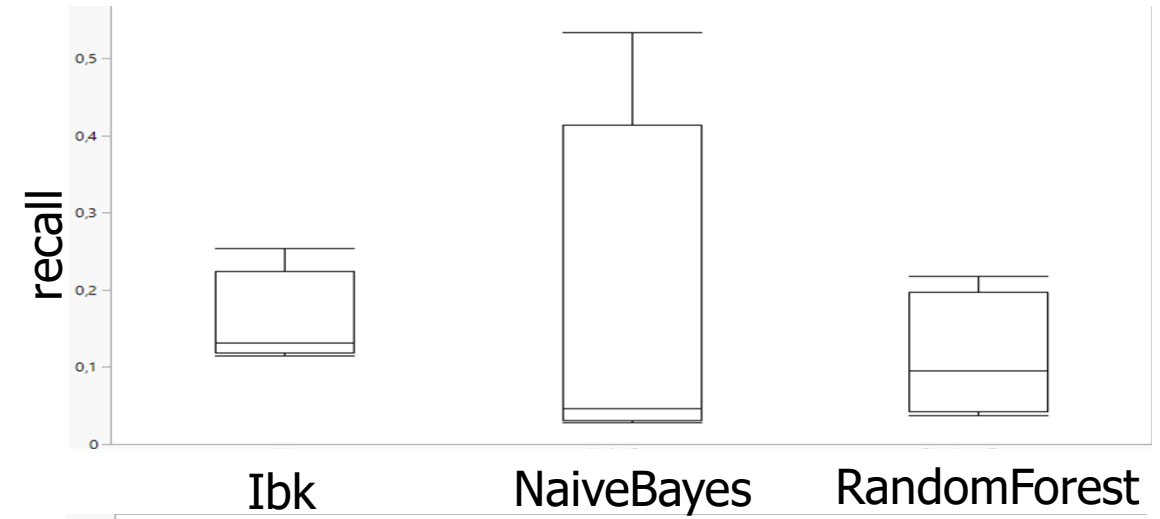
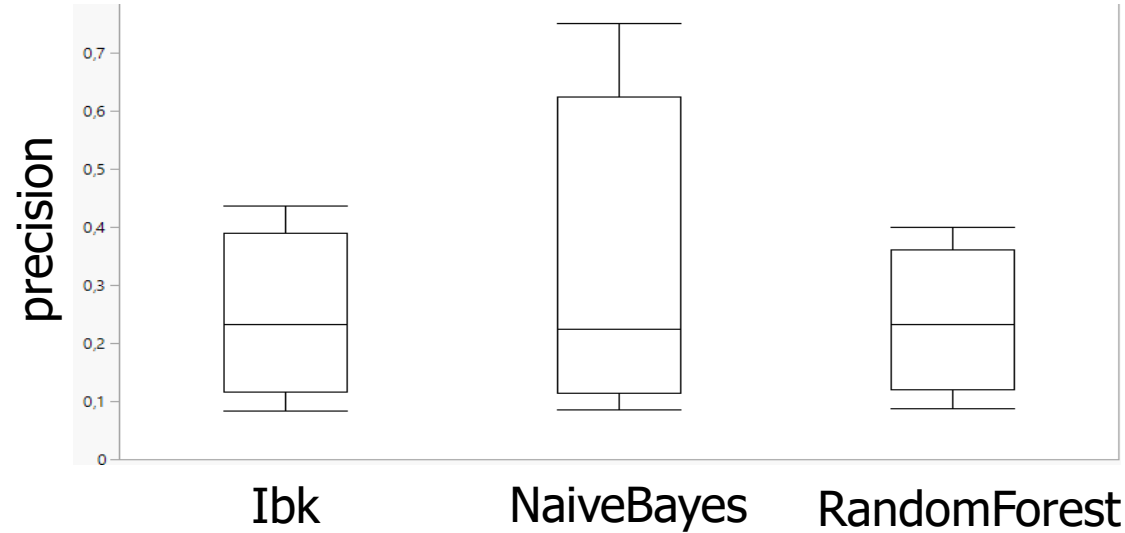




# Analisi dei risultati: BookKeeper(2/5)

NaiveBayes → precision, AUC  
Ibk → recall, Kappa

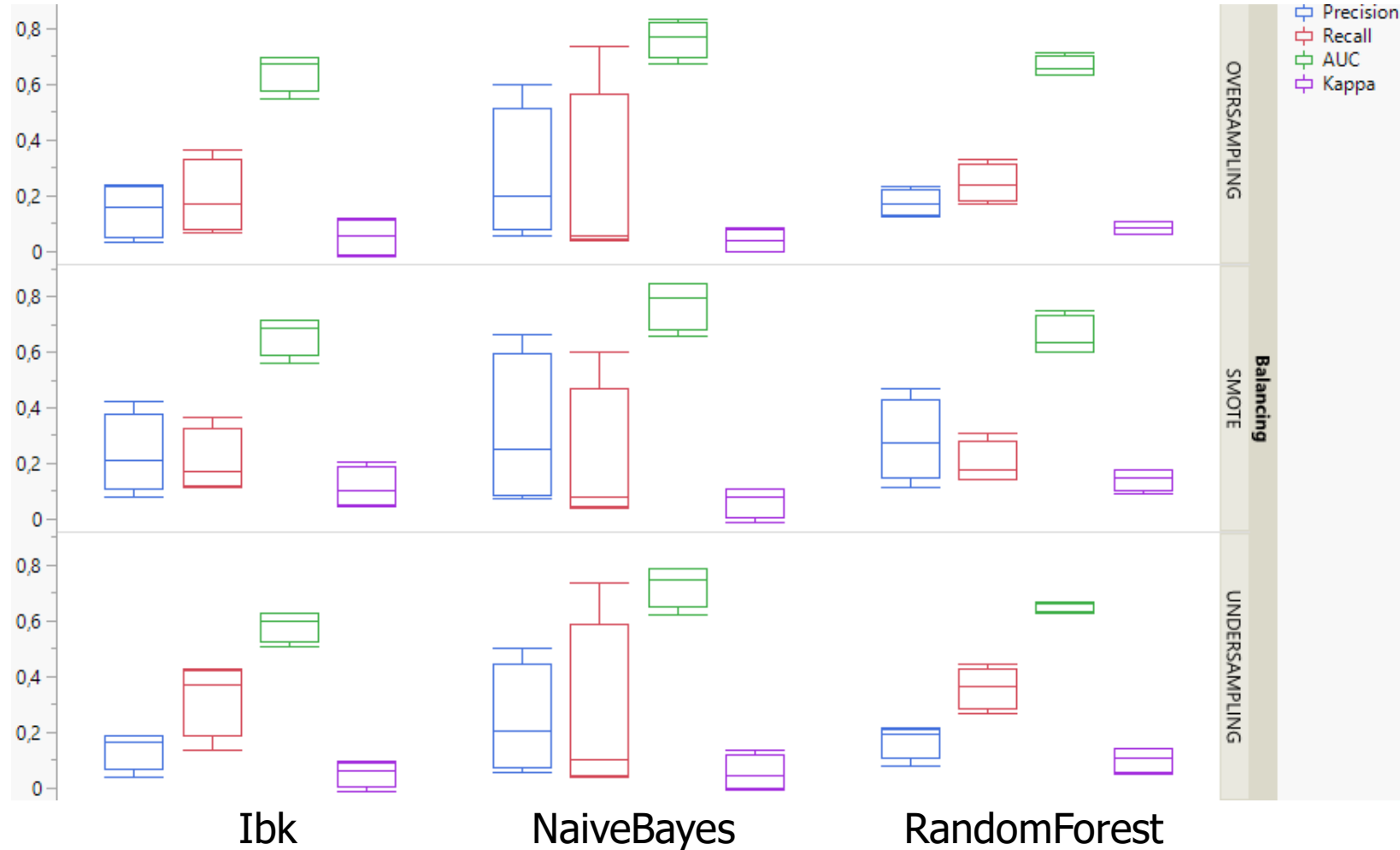
- Valutazione dei classificatori con **feature selection**, nello specifico approccio **filter** e ricerca **forward**



# Analisi dei risultati: BookKeeper(3/5)

Considero SMOTE (migliore):  
NaiveBayes → precision, Recall, AUC  
Kappa → tutti simili

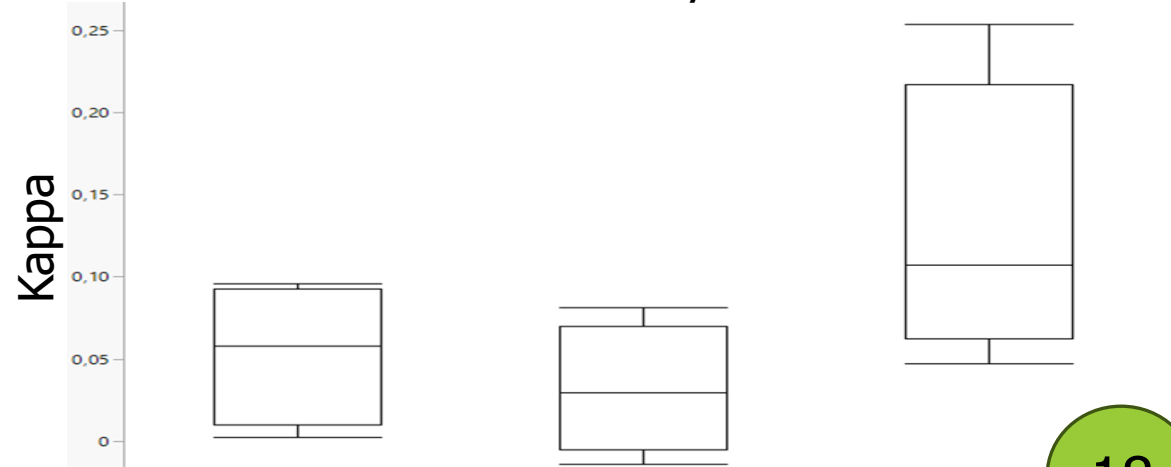
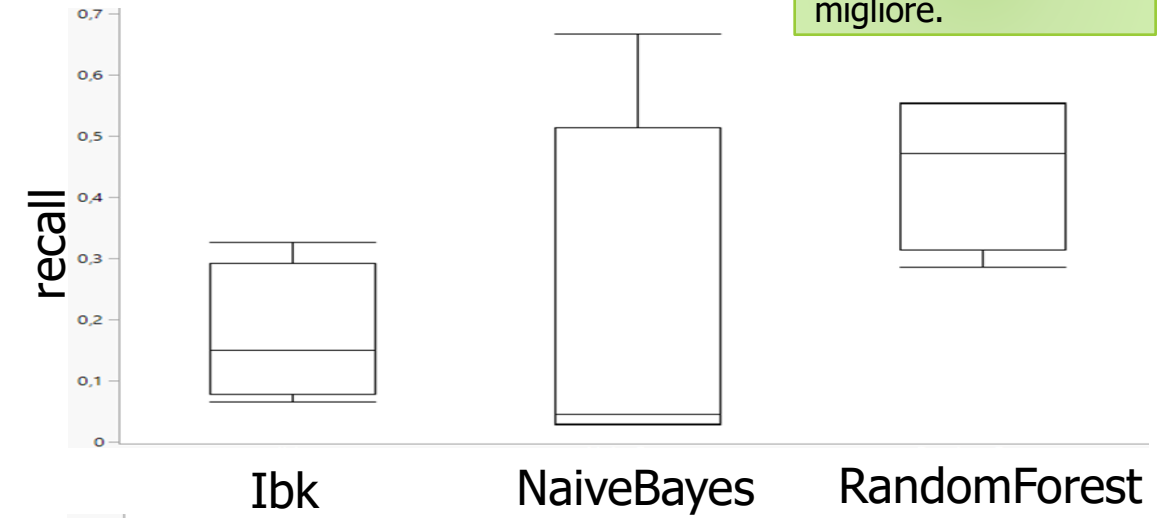
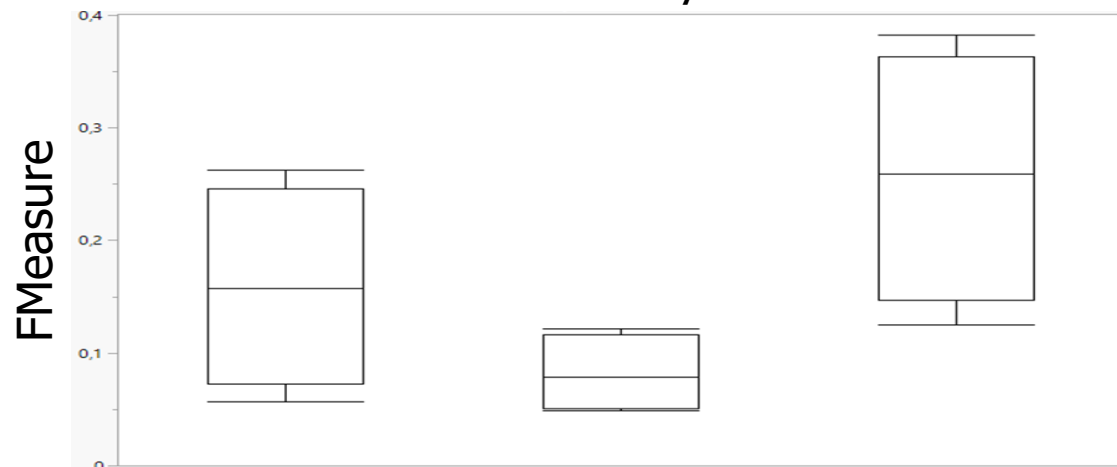
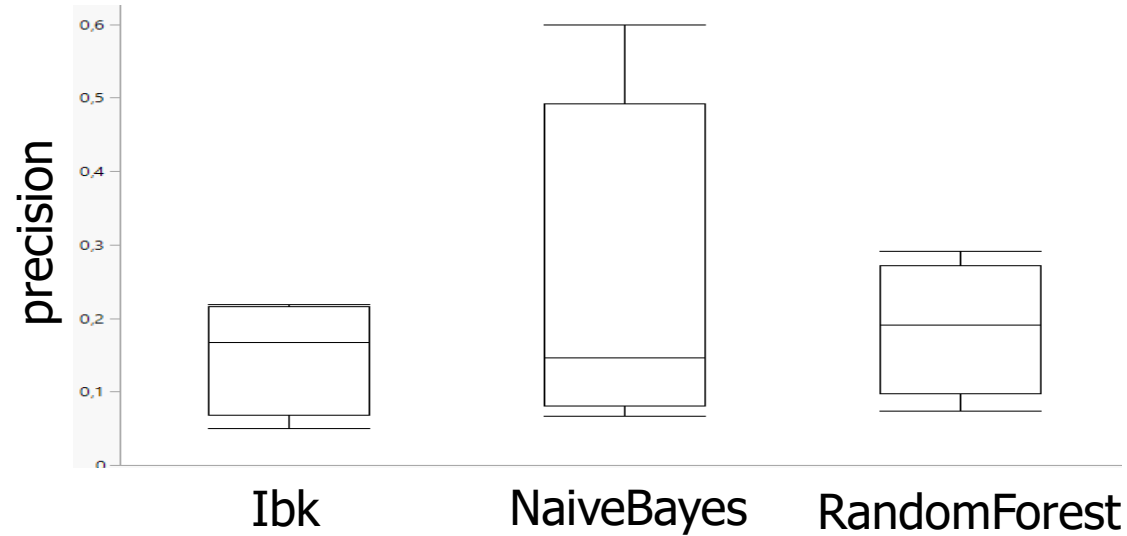
- Valutazione dei classificatori con tre sampling alternativi: **oversampling**, **undersampling** oppure **SMOTE**



# Analisi dei risultati: BookKeeper(4/5)

L'AUC è indipendente dalla threshold e quindi non è una metrica qui significativa. Perciò è stata sostituita con la **FMeasure**.

- Valutazione dei classificatori con **cost sensitive classifier**



**Random Forest** risulta nel complesso migliore.

# Analisi dei risultati: BookKeeper(5/5)

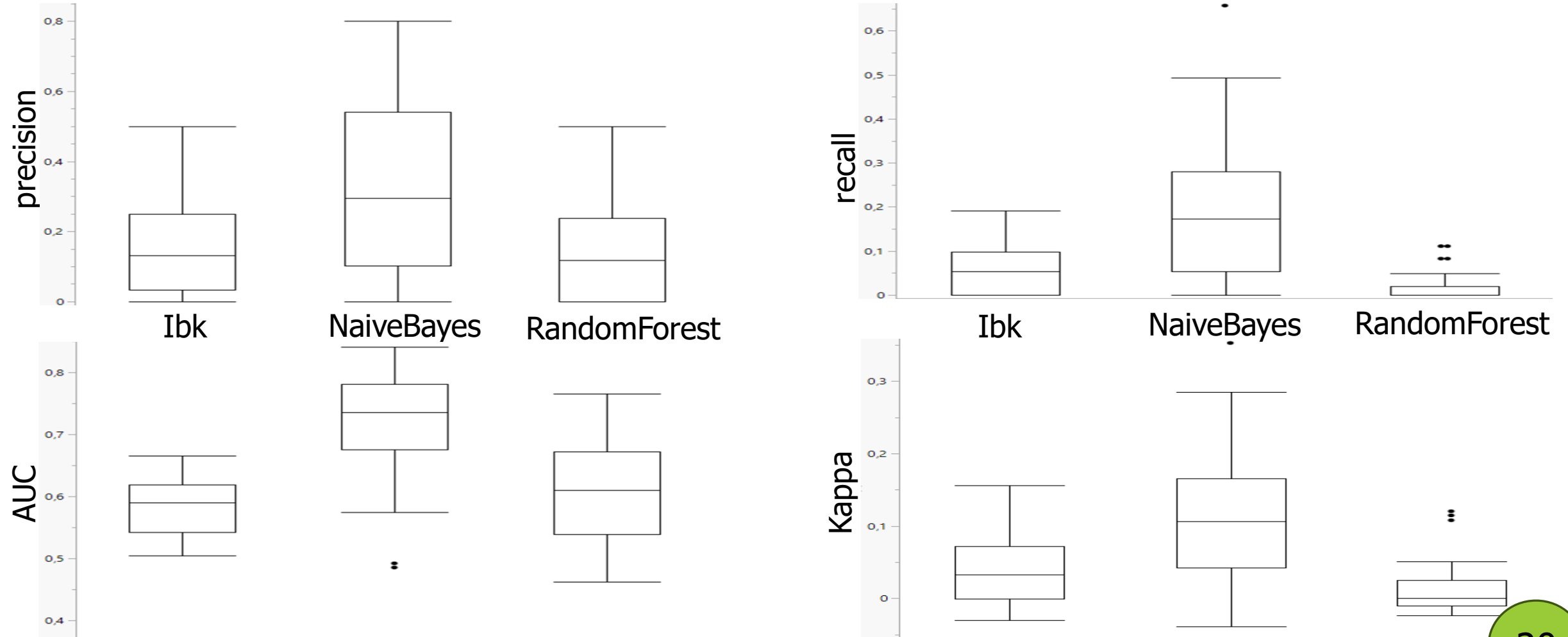
---

- **Senza alcuna strategia applicata**, si nota come **Ibk** non predice molto bene quando ci troviamo nelle prime iterazioni del walkforward, ma si comporta bene quando consideriamo più release nel training set (si notino i valori massimi nei grafici). **Random forest** risulta invece avere le distribuzioni più concentrate intorno ad un ristretto numero di valori, perciò risulta il migliore in termini di consistenza.
- Applicando la **feature selection** si nota come **Ibk** mostri buoni risultati, con distribuzioni molto strette. Invece **NaiveBayes** ha ottime valutazioni di predizione solo in alcune iterazioni (picchi sui valori massimi), ma è inconsistente. Però rispetto al caso base, la recall e la kappa di **Ibk** peggiorano (le altre metriche sono invariate) perciò **feature selection non risulta utile per IbK**. Invece **risulta utile per NaiveBayes** in quanto precision e kappa migliorano (le altre metriche sono invariate).
- Tra le strategie di **balancing** la migliore risulta essere **SMOTE**, infatti l'aumento delle istanze della classe minoritaria (buggy=true), tramite sintesi di nuove istanze 'artificiali' a partire da quelle originali, nel training set contribuisce ad una predizione che favorisce i positive (e quindi i TP), portando ad un giovamento su tutte le metriche. Confrontando i classificatori con **SMOTE** applicato **NaiveBayes** presenta valori superiori su tutte le metriche (solo sulla Recall **Ibk** è più consistente, ma NaiveBayes presenta massimo notevolmente superiore). Rispetto al caso senza alcuna tecnica applicata, NaiveBayes con SMOTE mostra un **miglioramento** della **precision** (altre metriche invariate).
- **Random Forest** risulta il classificatore che mostra i giovamenti maggiori dall'applicazione di **cost sensitive classifier**, specialmente nell'ambito della Recall. Infatti aumentare il costo dell'errore sui FN ha indotto il classificatore a favorire i positive, portando a maggiori errori (la precision non è ottimale), ma permettendo di individuare molte più classi buggy (la **recall** migliora di molto rispetto al caso senza cost sensitive). Questa combinazione risulta anche la migliore tra quelle esposte.

# Analisi dei risultati: Syncope(1/5)

**NaiveBayes** risulta migliore su tutte le metriche.

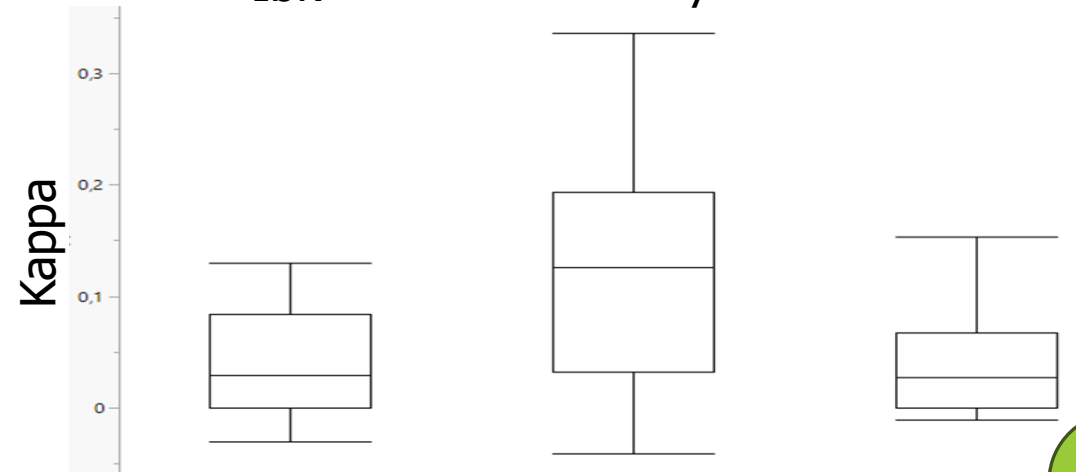
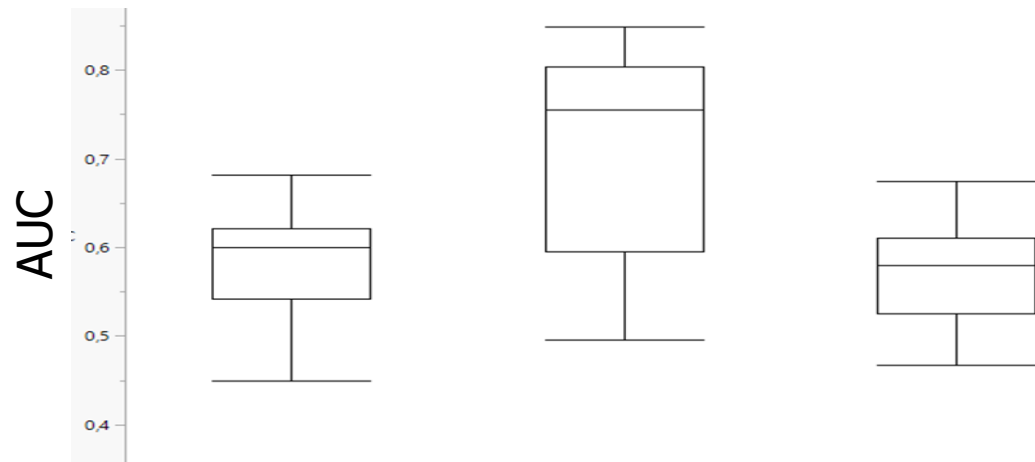
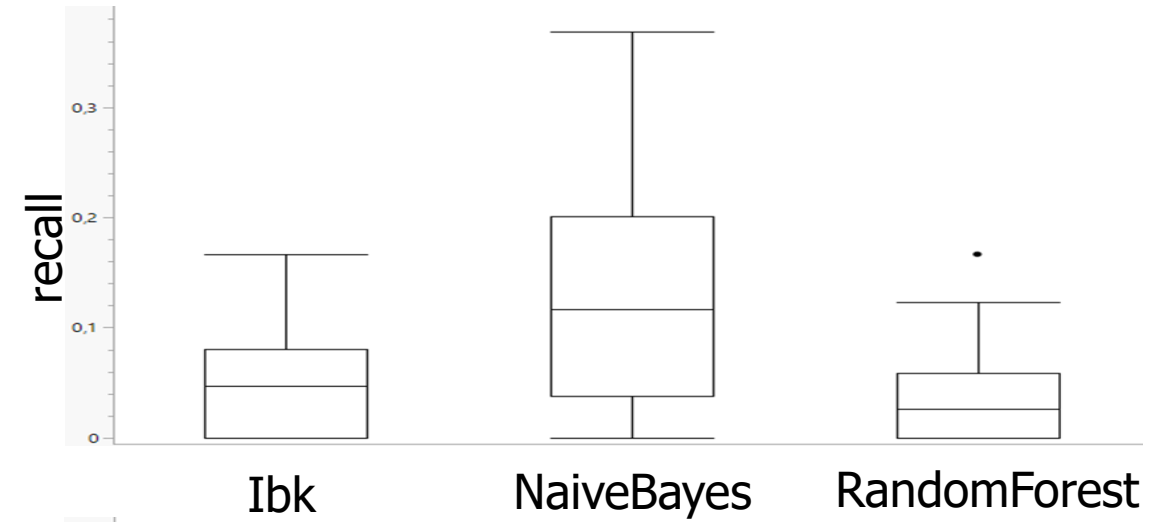
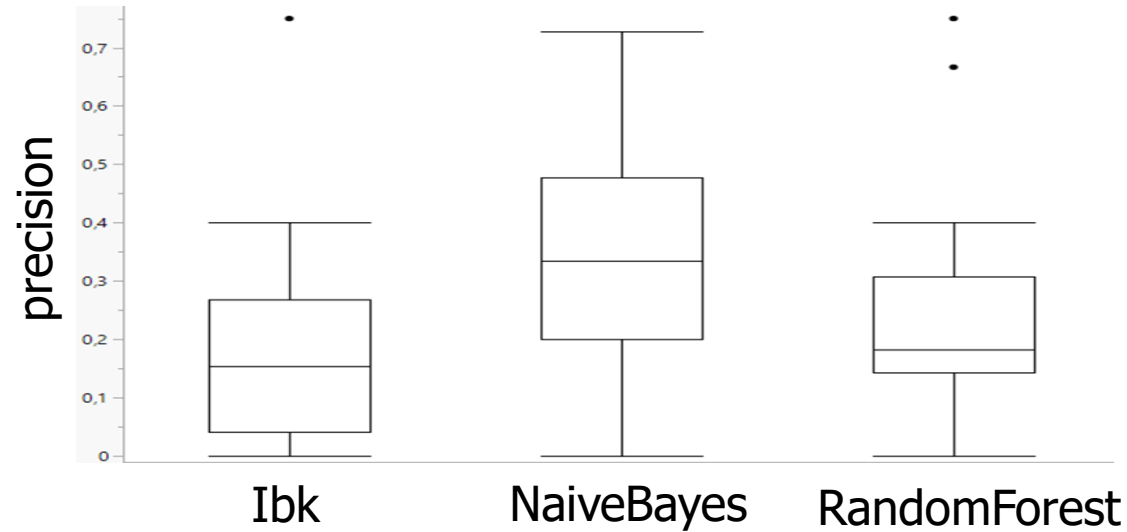
- Valutazione dei classificatori **senza sampling, feature selection o cost sensitive**:



# Analisi dei risultati: Syncope(2/5)

**NaiveBayes** risulta migliore su tutte le metriche.

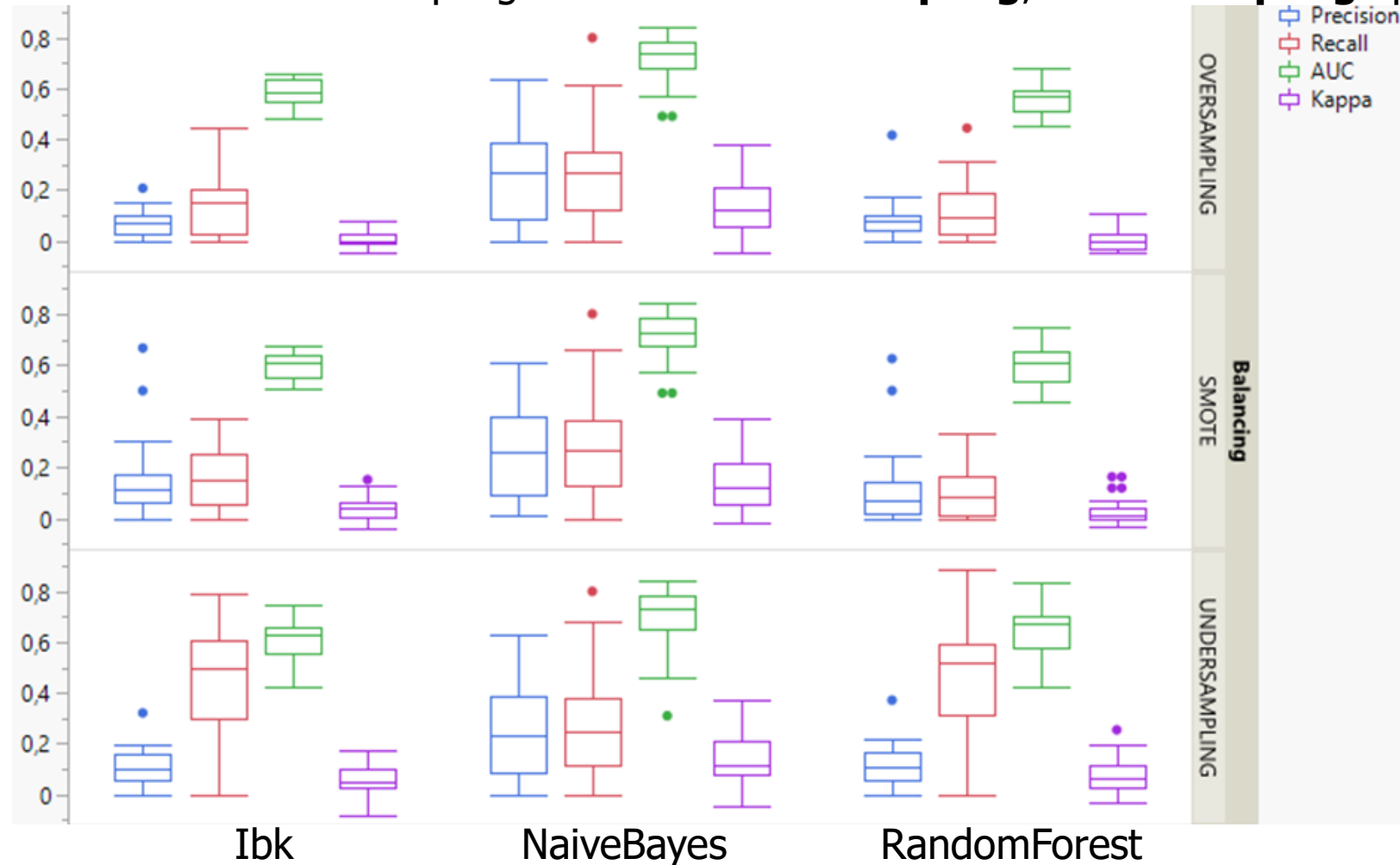
- Valutazione dei classificatori con **feature selection**, nello specifico approccio **filter** e ricerca **forward**



# Analisi dei risultati: Syncope(3/5)

Considero UNDERSAMPLING (migliore):  
NaiveBayes → precision, AUC, Kappa  
RandomForest → recall (simile a IbK)

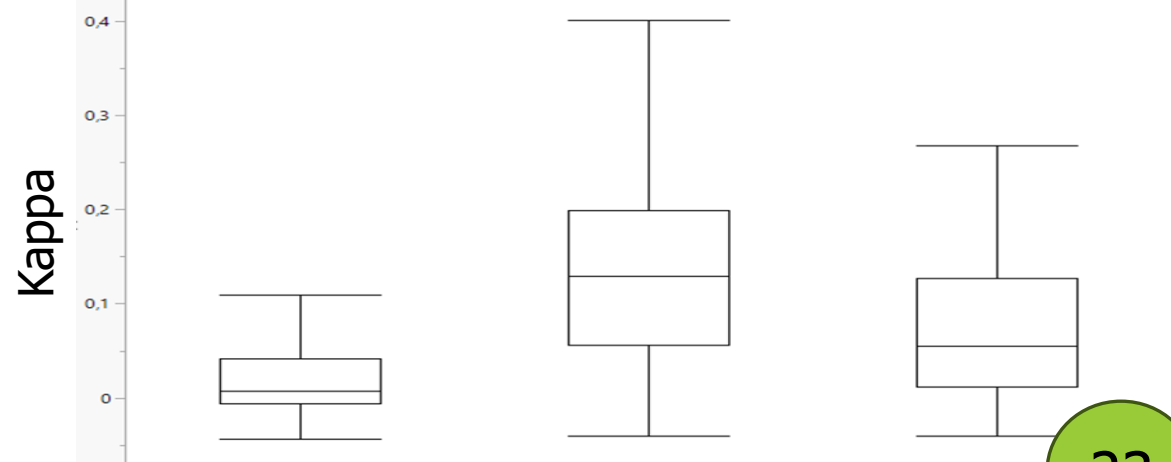
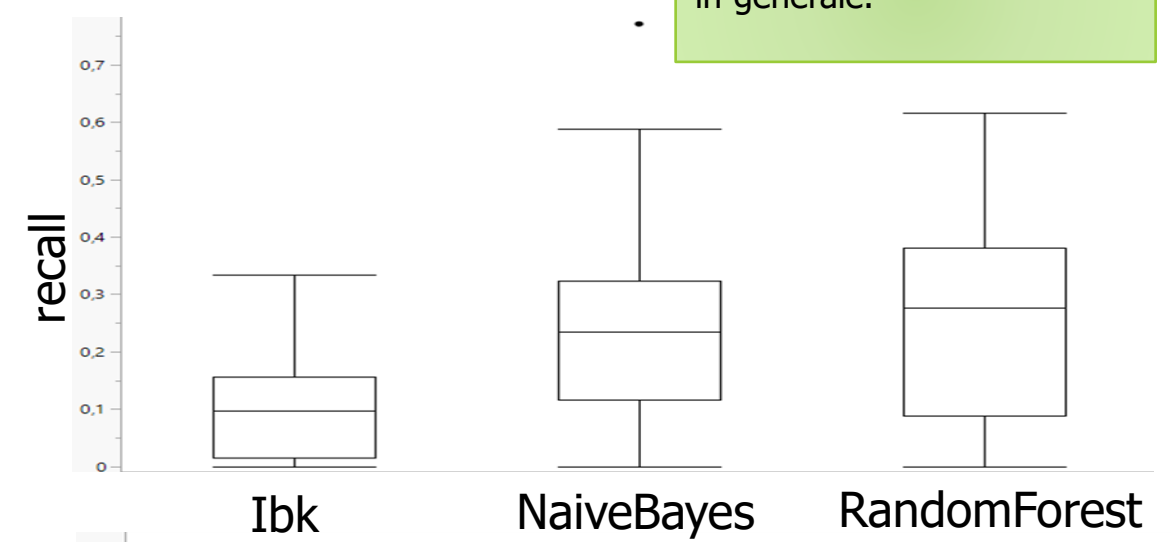
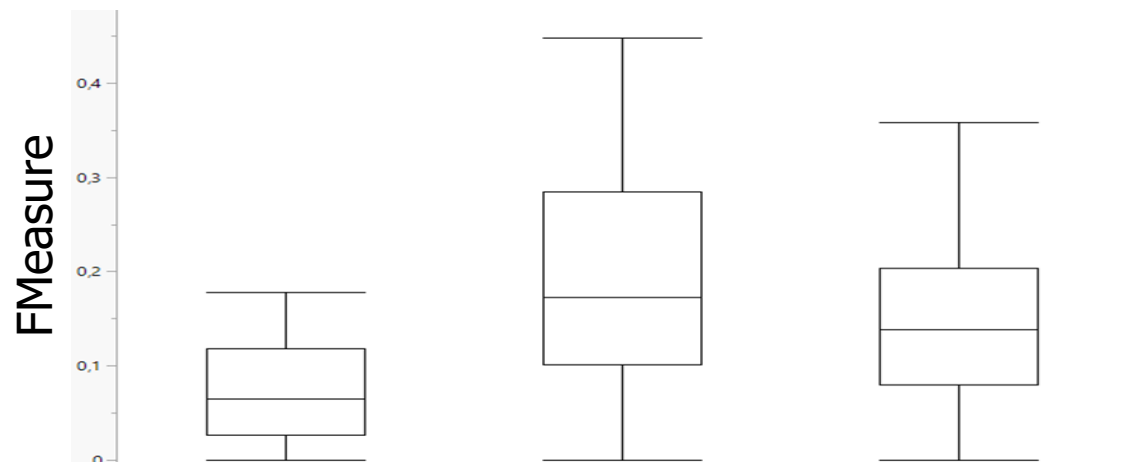
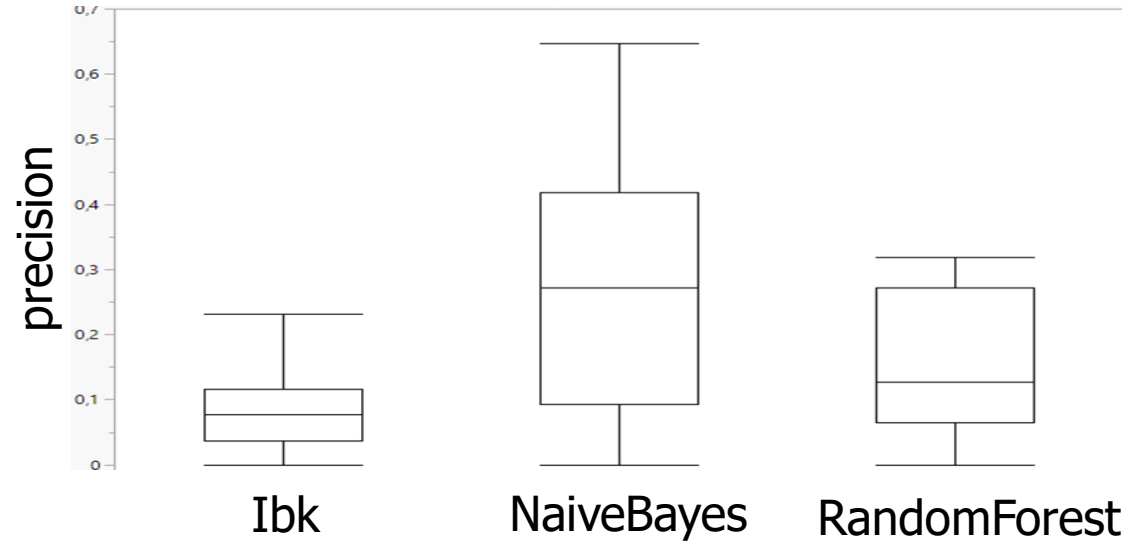
- Valutazione dei classificatori con tre sampling alternativi: **oversampling**, **undersampling** oppure **SMOTE**



# Analisi dei risultati: Syncope(4/5)

L'AUC è indipendente dalla threshold e quindi non è una metrica qui significativa. Perciò è stata sostituita con la **FMeasure**.

- Valutazione dei classificatori con **cost sensitive classifier**



**NaiveBayes** risulta migliore in generale.



# Analisi dei risultati: Syncope(5/5)

---

- Senza alcuna tecnica applicata, **NaiveBayes** presenta delle performance predittive migliori rispetto agli altri classificatori.
- Applicando **feature selection**, il **migliore** tra i classificatori risulta sempre **NaiveBayes**, che però **non trae giovamento** da questa selezione degli attributi, infatti precision e recall peggiorano (le altre metriche sono invariate).
- Tra le diverse strategie di **balancing**, l'**undersampling** risulta la scelta migliore. Confrontando i classificatori con questo tipo di sampling applicato, **NaiveBayes** mostra in generale delle performance migliori, inoltre si evidenzia un **guadagno su recall e kappa** (le altre sono invariate) rispetto al caso senza balancing.
- Applicando **cost sensitive classifier**, **NaiveBayes** continua a risultare migliore. **Recall e Kappa migliorano** rispetto al caso base, mentre la precision peggiora (stesso discorso svolto per bookKeeper, aumentano i positivi, sbaglia di più, ma individua più classi buggy). Inoltre con cost sensitive si ottengono dei valori complessivamente più alti nelle varie metriche rispetto a undersampling.

# Considerazioni finali

---

- Nell'ambito di **BookKeeper**, l'utilizzo di **Random forest con cost sensitive classifier** risulta la scelta migliore. In **Syncope** invece **NaiveBayes con cost sensitive classifier** risulta la combinazione migliore. In linea generale cost sensitive è quindi la tecnica che maggiormente migliora le prestazioni dei classificatori per l'individuazione della buggyness nelle coppie classe/release. Ciò è dovuto ad una gestione dei costi degli errori che sfavorisce i negativi (buggy = false), che hanno un costo d'errore maggiore, e favorisce i positivi, portando i classificatori ad individuare più classi buggy.
- Tra le strategie di sampling, che hanno anche esse condotto a dei buoni risultati, la tecnica **SMOTE** è risultata utile in **BookKeeper**, l'**undersampling** in **Syncope**. Questo comportamento trova le sue ragioni nella diversa size dei due progetti software. BookKeeper infatti conta poche release (solo 6), quindi un numero inferiore di istanze nel dataset, perciò una sintesi di nuove istanze della classe minoritaria nel training set come quella attuata da SMOTE favorisce la classificazione per questo progetto. L'opposto si verifica in Syncope, dove la dimensione ingente del dataset originario (sono 32 release) giova di una riduzione della presenza della classe maggioritaria come quella attuata dall'undersampling.
- Infine si nota come i risultati ottenuti su Syncope applicando diverse tecniche presentano un maggiore 'equilibrio' (NaiveBayes risulta sempre il classificatore migliore sulle diverse metriche). L'elevato numero di coppie classe/release in Syncope rende i suoi risultati più attendibili.

# Riferimenti

---

1. *Leveraging the Defects Life Cycle to Label Affected Versions and Defective Classes.* Bailey Vandehei, Vaniel Alencar Da Costa, Davide Falessi. In ACM Transactions on Software Engineering and Methodology, Volume 30, Issue 2.
2. *The Impact of Dormant Defects on Defect Prediction: A Study of 19 Apache Projects.* Davide Falessi, Aalok Ahluwalia, Massimiliano Di Penta. ACM Transactions on Software Engineering and Methodology, Volume 31, Issue 1.
3. *Preserving Order of Data When Validating Defect Prediction Models.* Davide Falessi, Likhita Narayana , Jennifer Fong Thai , Burak Turhan.