MACHINE LEARNING FOR SOFTWARE ENGINEERING

Filippo Muscherà - 0338276

AGENDA

1. Introduzione
2. Progettazione
3. Variabili
4. Risultati
5. Assunzioni & Minacce alla validità
6. Link

1. INTRODUZIONE

Problema

- Nell'ingegneria del software, uno dei punti cruciali è **individuare e correggere** eventuali **bug** del software, prima che questo venga rilasciato.
  A questo scopo si effettua l'attività di testing.

- Il testing è un'attività dispendiosa, sia dal punto di vista economico che temporale.
  Non si può testare tutto in modo esaustivo.

- Bisogna **individuare quali classi testare**, e quali tralasciare.
  Come si può fare questo senza correre il rischio che il testing perda di efficacia?

1. INTRODUZIONE

Idea

- L’idea è quella di usare il **Machine Learning** per **predire** quali classi sono affette da **bug**.

- Sulla base di questa predizione si indirizzeranno gli sforzi per sviluppare casi di test, e individuare effettivamente i difetti del codice.

- Per **addestrare il modello** di Machine Learning si può fare uso di informazioni:
  - Prese da altri progetti
  - Prese da release passate del progetto stesso.

- Quindi si useranno dati del passato per addestrare e validare il modello, e quando si troverà il più efficace, lo si userà per predire i bug nella release attuale del progetto che si sta sviluppando.

1. INTRODUZIONE

Metodologia

- Abbiamo preso due progetti open-source della Apache Software Foundation: BookKeeper e OpenJPA.
- Di questi progetti sono state individuate le classi buggy per ogni release.
- Sono poi stati considerati quattro classificatori: Naive Bayes, Ibk, Random Forest e Multilayer Perceptron.
- Si è cercato di stabilire quale di questi si comportasse meglio nel predire la presenza di difetti nelle classi dei progetti.
- Si sono impiegate varie tecniche di utilizzo dei classificatori, anche combinate tra loro. Sono state considerate:
  Feature Selection, **Sampling** e **Cost Sensitive Learning.**

2. PROGETTAZIONE

Individuazione dei bug

- Prima di poter procedere alla costruzione del dataset per addestrare il modello di Machine Learning, bisognerà individuare quali classi sono buggy, e in quali release.
  Idea: **ogni bug ha un ciclo di vita**:

Injected Version: è la versione in cui il bug viene introdotto in una classe.

Opening Version: è la versione in cui il bug viene individuato a seguito di una failure.

Fix Version: è la versione in cui il bug viene risolto.

- Una classe quindi sarà difettosa dalla Injected Version alla Fix version. La Fix version non sarà però compresa tra quelle affette dal bug, dato che sarà la prima che conterrà il fix per quest'ultimo.

2. PROGETTAZIONE

Proportion[1]

- La **OV** e la **FV** sono facilmente individuabili dal **ticket di Jira**, poiché ci sarà sempre una data di apertura e chiusura del ticket.

- La **IV** invece, la maggior parte delle volte **non è indicata** nel ticket, ma noi ne abbiamo bisogno per calcolare il ciclo di vita del bug.

- Idea: ci sarà una **proporzionalità** tra il tempo che passa tra FV e OV e tra FV e IV. In pratica, se un bug è stato difficile da individuare, ci sarà voluto, in proporzione, più tempo per risolverlo. E viceversa.

- Questa tecnica prende il nome di **Proportion**, dove:

$$p = \frac{FV - IV}{FV - OV} \longrightarrow IV = FV - (FV - OV) \cdot p$$

- Quindi, sfruttando i ticket che hanno anche la IV, ci si può calcolare **p**, e utilizzarla per **stimare** la **IV** dei ticket che non la possiedono.

2. PROGETTAZIONE

Tecniche di Proportion

- Per la **stima** della **IV** sono state adottate due tecniche diverse, in base alla situazione che si presentava:

Se si hanno a disposizione **almeno 5 ticket** con la IV si calcola **''Proportion Increment''**: si calcola p usando tutti i ticket con la IV presenti nel progetto fino a quel momento.

PRO: Usa informazioni raccolte dal progetto stesso, che avranno la **correlazione più alta** possibile con gli altri difetti del progetto.

CONTRO: Inizialmente potrei non avere abbastanza ticket.

Se si hanno **meno di 5 ticket**, si usa **''Proportion Cold Start''**: si calcola p per altri progetti (nel nostro caso altri progetti Apache) e si prende la **mediana**.

PRO: È sempre possibile da calcolare.

CONTRO: Potrebbe avere una correlazione minore con il progetto attuale, dato che si prendono informazioni da altri progetti.

Nota: quando parliamo di ticket ci riferiamo sempre a ticket **validi**.

2. PROGETTAZIONE

Dataset e Snoring[2]

- Ora sappiamo come individuare le classi con difetti, e sappiamo come associare a ogni difetto le release afflitte dal bug. Si deve ora costruire il dataset per addestrare i classificatori.

- Nota: per l'addestramento dei modelli di ML, si scarta tutta la seconda metà del dataset, per ridurre il fenomeno dello **snoring**.

  Per snoring si intende quel fenomeno che si verifica quando una **classe** che si ritiene non buggy, in realtà contiene uno o più **bug non ancora identificati (bug dormienti)**.

- Ogni classificatore ha bisogno di alcune **metriche** su cui basarsi. Durante il training cercherà di mettere in relazione il valore delle varie metriche con la buggyness della classe. Userà poi le relazioni che ha individuato per fare le sue predizioni.

2. PROGETTAZIONE

Metriche

Nome	Descrizione
LOC	Numero di righe di codice
LOCTouched	Somma di righe aggiunte e rimosse in una release
NR	Numero di commit che hanno toccato la classe
nAuth	Numero di autori della classe
locAdded	Righe di codice aggiunte alla classe in una release
maxLocAdded	Numero massimo di righe aggiunte alla classe in una release
avgLocAdded	Media delle righe di codice aggiunte alla classe dai commit di una release
Churn	Di quante LOC il codice è cambiato rispetto alla release precedente
maxChurn	Churn massimo all'interno della release
avgChurn	Churn medio all'interno della release
handledExceptions*	Numero di eccezioni di cui si effettua il catch all'interno della classe
nFix	Numero di bug fixati che la classe ha avuto dall'inizio del progetto
cyclComplexity*	Numero di cammini indipendenti presenti all'interno del codice della classe

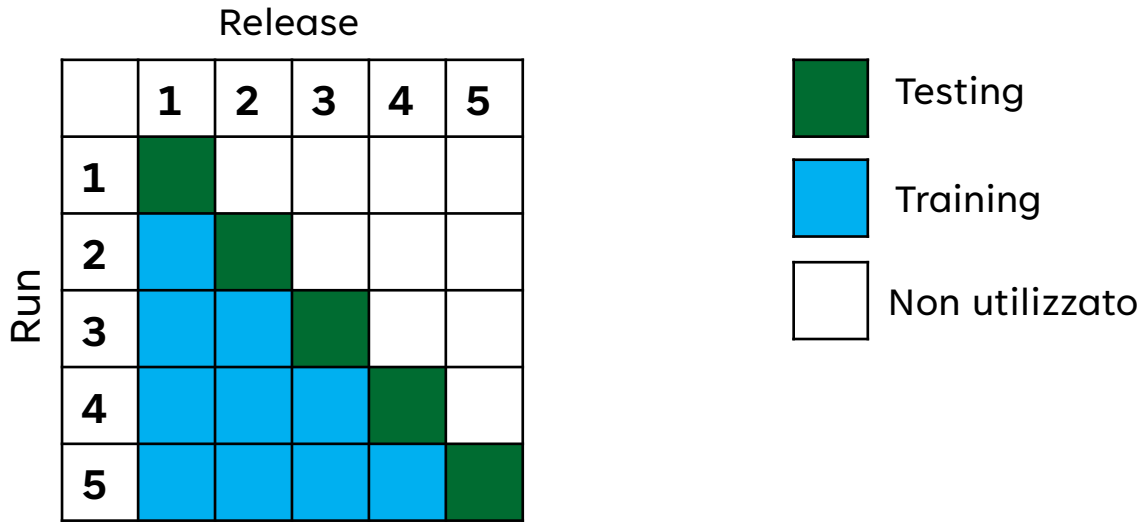
*: Metriche personalizzate

2. PROGETTAZIONE

Walk Forward

- Per definire quale classificatore ha le migliori performance nel predire la buggyness delle classi, abbiamo bisogno di **valutare i vari classificatori**.

- Si è usato la tecnica del **Walk-Forward**:



- Usiamo walk-forward perché è una tecnica **time-series**: tiene conto **dell'ordine temporale dei dati**. Per esempio per predire la release 3, userò i dati delle release 1 e 2. Se usassi i dati della release 4, starei usando informazioni di cui non potrei disporre, perché future rispetto a ciò che sto predicendo.

2. PROGETTAZIONE

Training & Testing Set

- Proprio perché stiamo usando tecniche time-series, bisogna costruire il training e il testing set in maniera diversa:

Training set:

- Deve simulare la situazione in cui si troverà a operare il classificatore, in modo **realistico**.
- **Non** può usare **informazioni** (ticket) **future**.
- **Non** avrà informazioni del tutto **precise** (snoring).

Testing set:

- Il suo scopo è valutare le prestazioni del classificatore. Fungendo da **oracolo**, deve essere il più preciso possibile.
- Può usare informazioni **future**, sfruttando **tutti i dati a disposizione**.
- Sarà **preciso**, non realistico.

3. VARIABILI

Classificatori e Variabili

- Sono stati utilizzati i seguenti **classificatori**, in combinazione con le **tecniche** elencate qui sotto, per ogni iterazione del Walk-Forward:

Classificatori:

- Naive Bayes
- Ibk
- Random Forest
- Multilayer Perceptron

È stato utilizzato un classificatore per ognuna delle famiglie di modelli disponibili su Weka (bayes, lazy, tree, function).

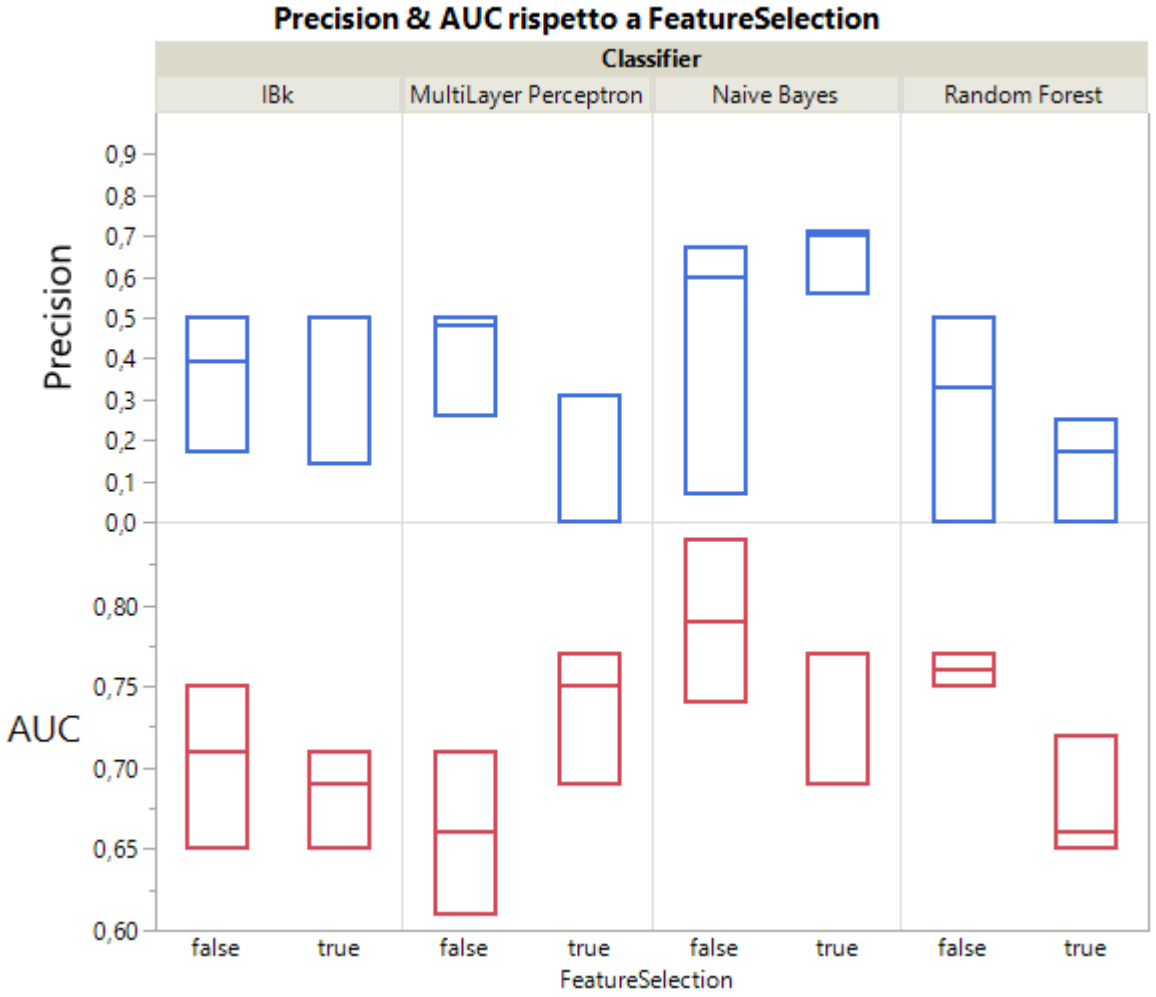
Tecniche:

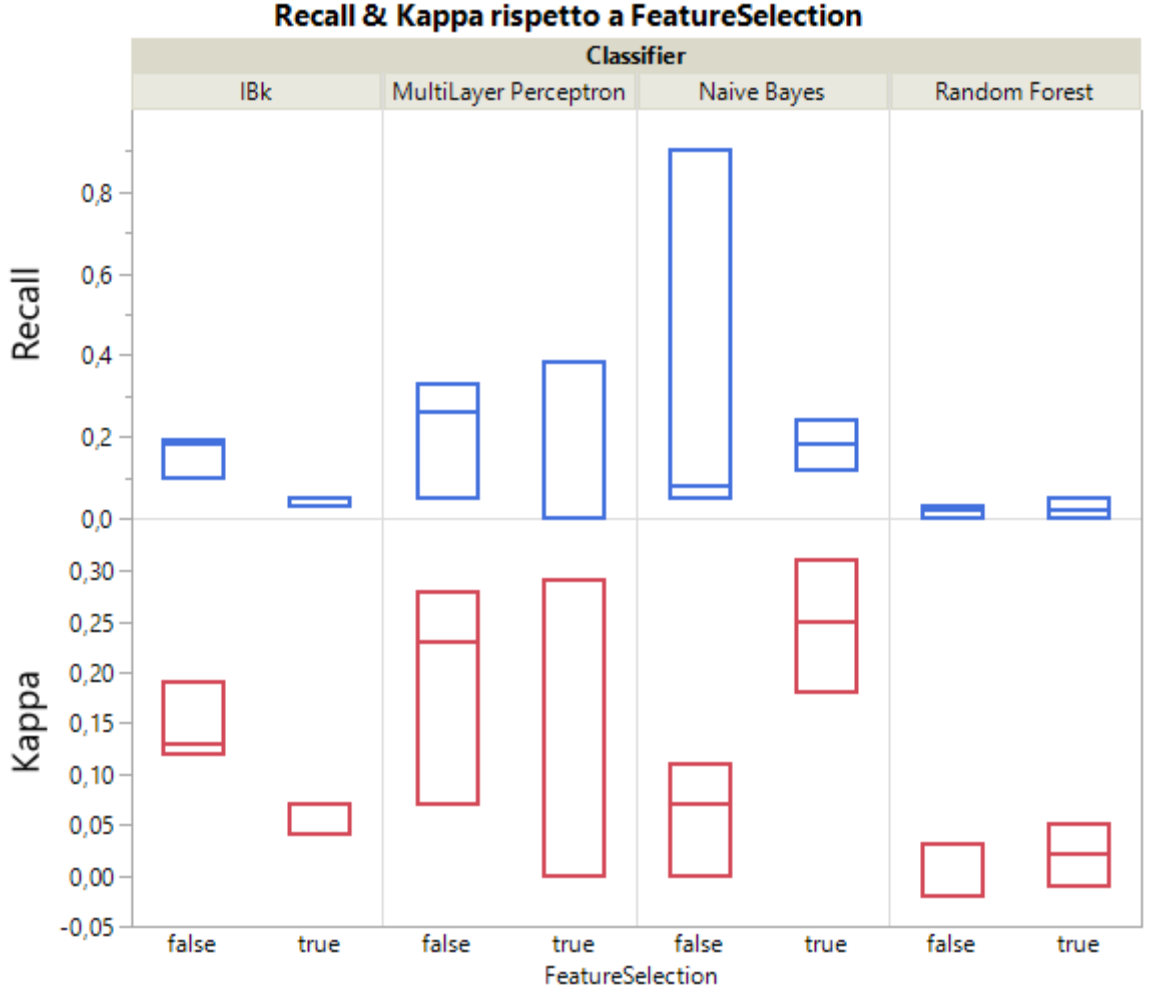
- Feature Selection (bestFirst)
- Undersampling
- Oversampling
- SMOTE
- Cost Sensitive Learning

Non sono stati utilizzati contemporaneamente Sampling/SMOTE e Cost Sensitive, perché sono due metodi diversi di fare **balancing** del dataset.

4. RISULTATI

BookKeeper: No FS vs FS

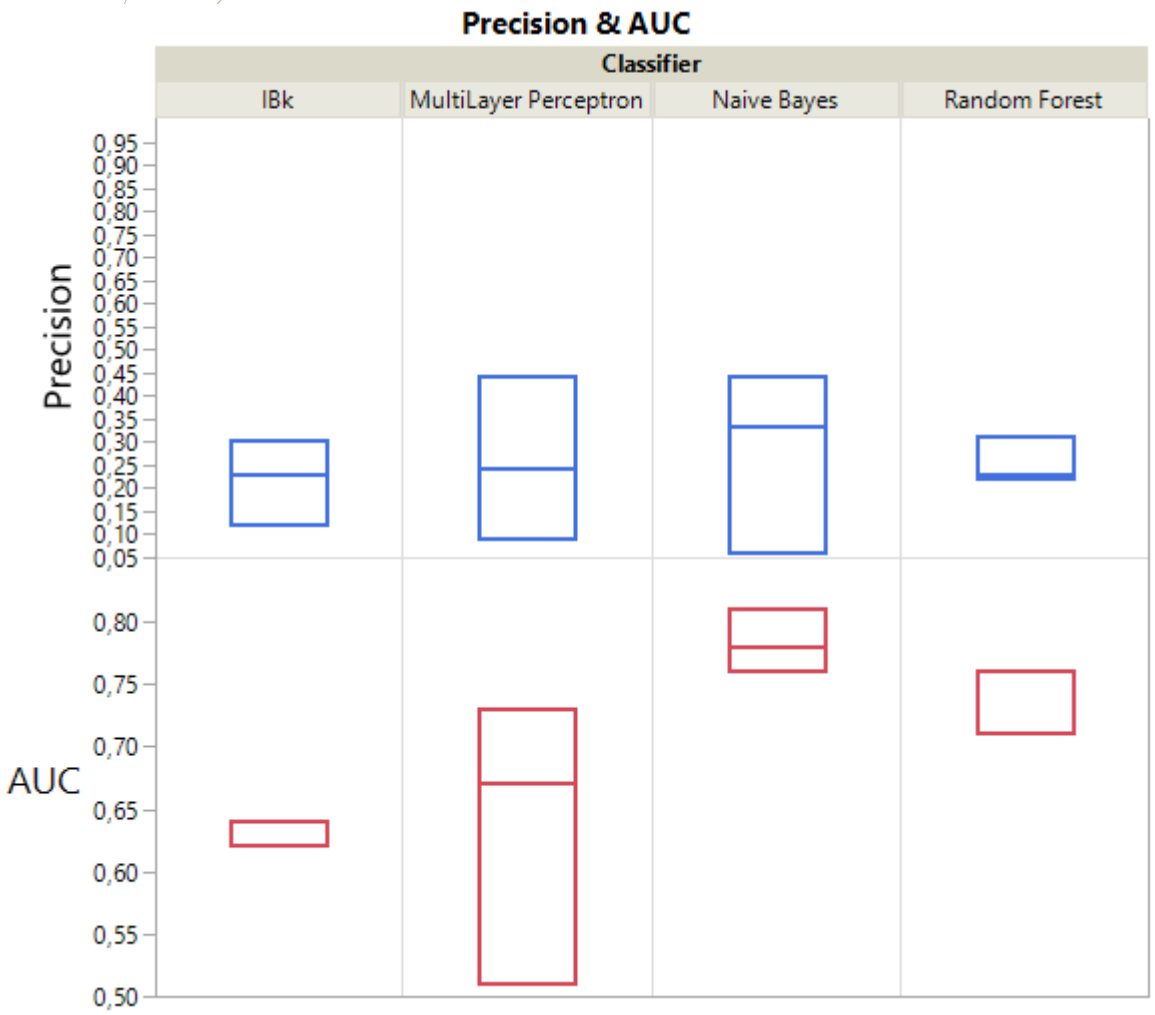


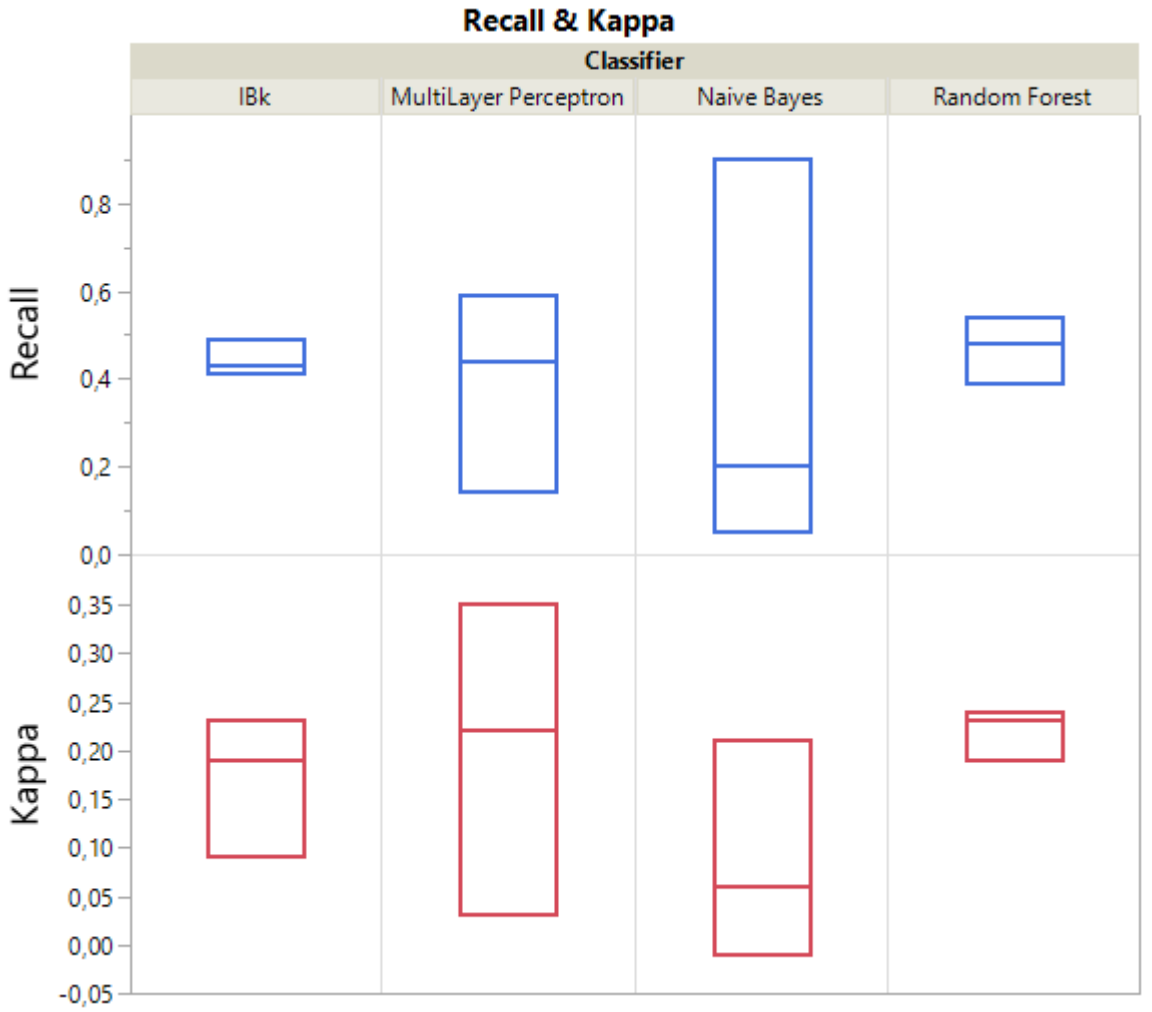


- In questa situazione **Naive Bayes** raggiunge risultati migliori, ma è anche meno stabile: ha recall che si avvicinano sia a 1 che a 0. La FS sembra però stabilizzare questo comportamento.

4. RISULTATI

BookKeeper: Undersampling

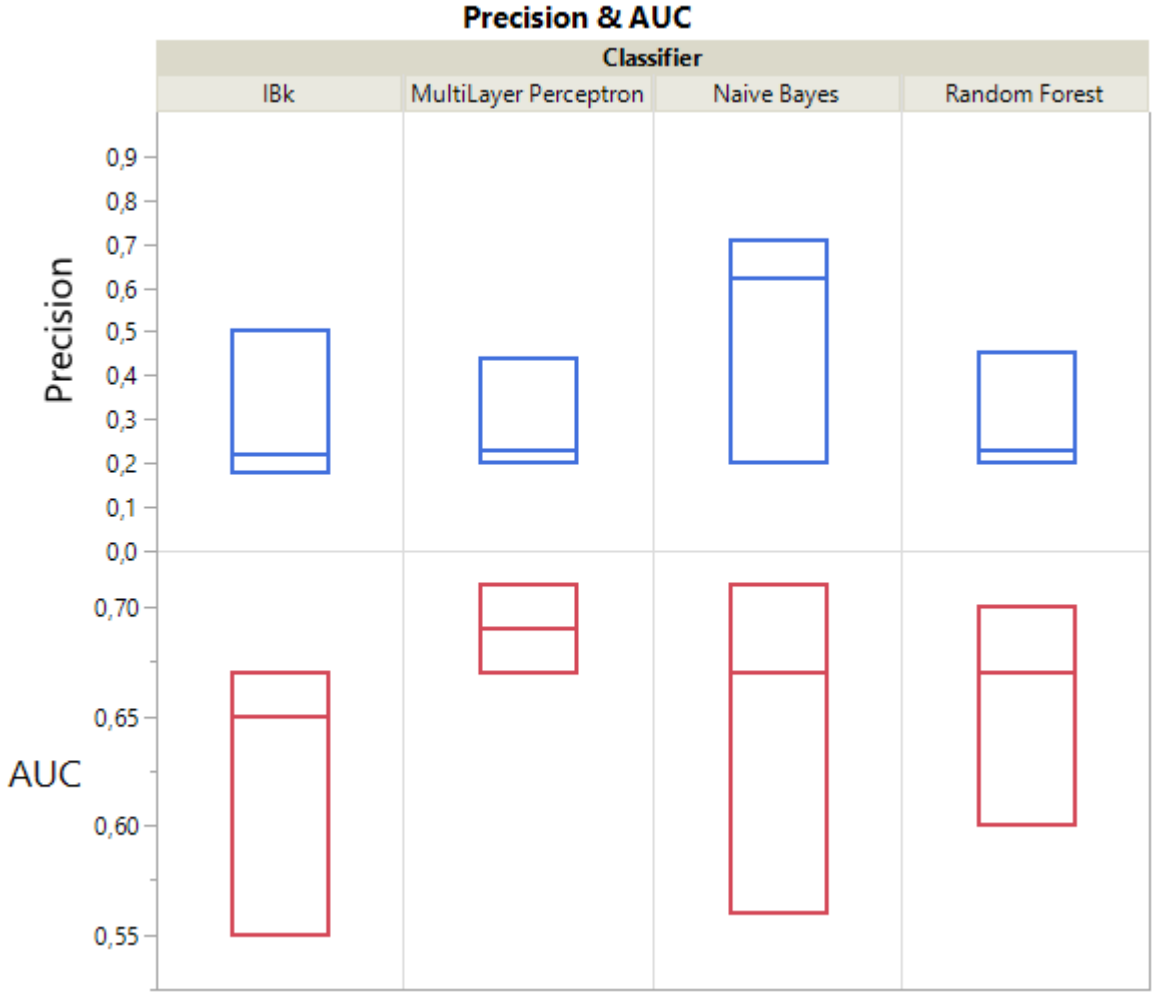


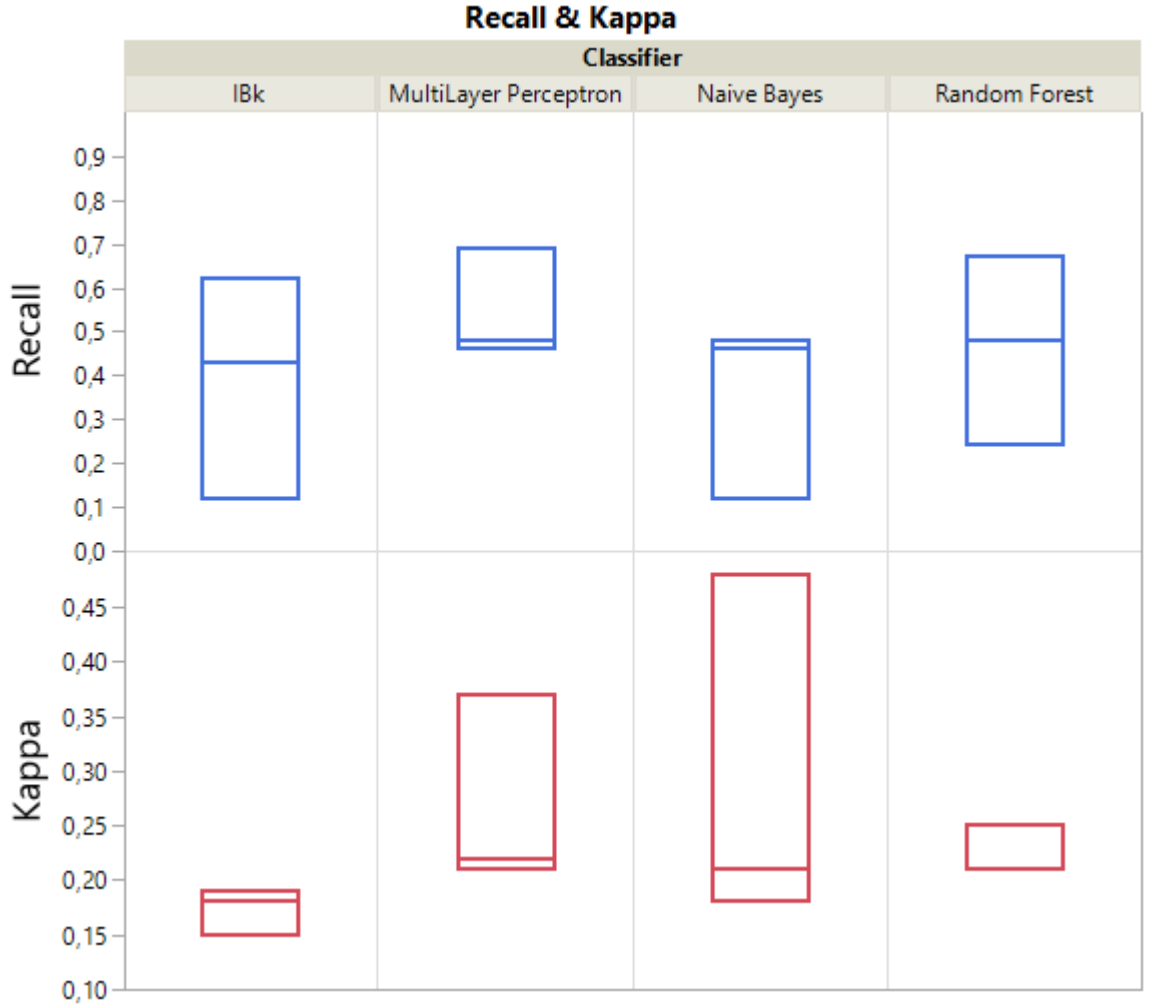


- Con **l'undersampling** la **recall migliora** sensibilmente per **tutti i classificatori**. **Ibk** e **Random Forest** sembrano i classificatori **migliori** e più stabili. Sembra chiaro che per ottenere risultati migliori serva del **balancing**.

4. RISULTATI

BookKeeper: Cost Sensitive

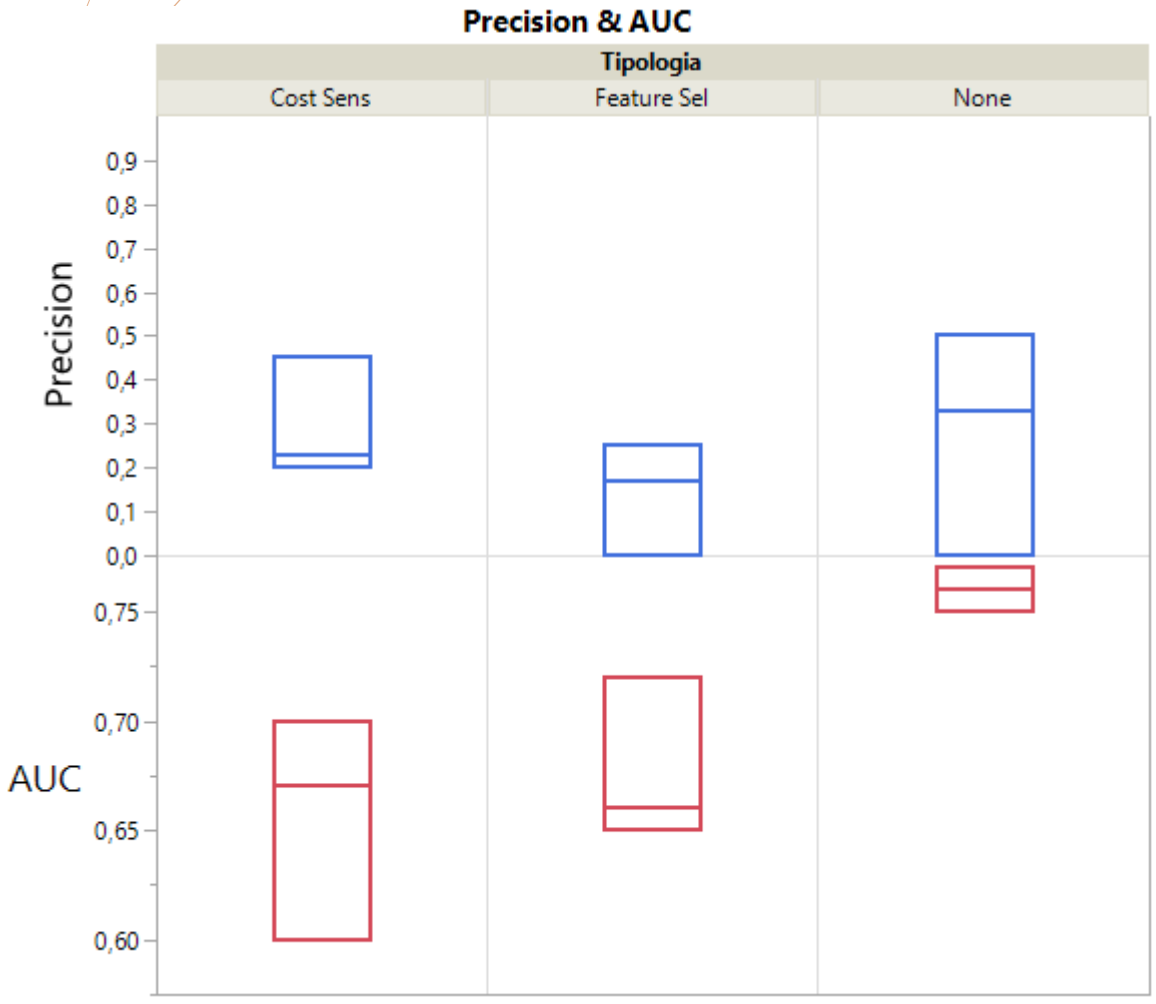


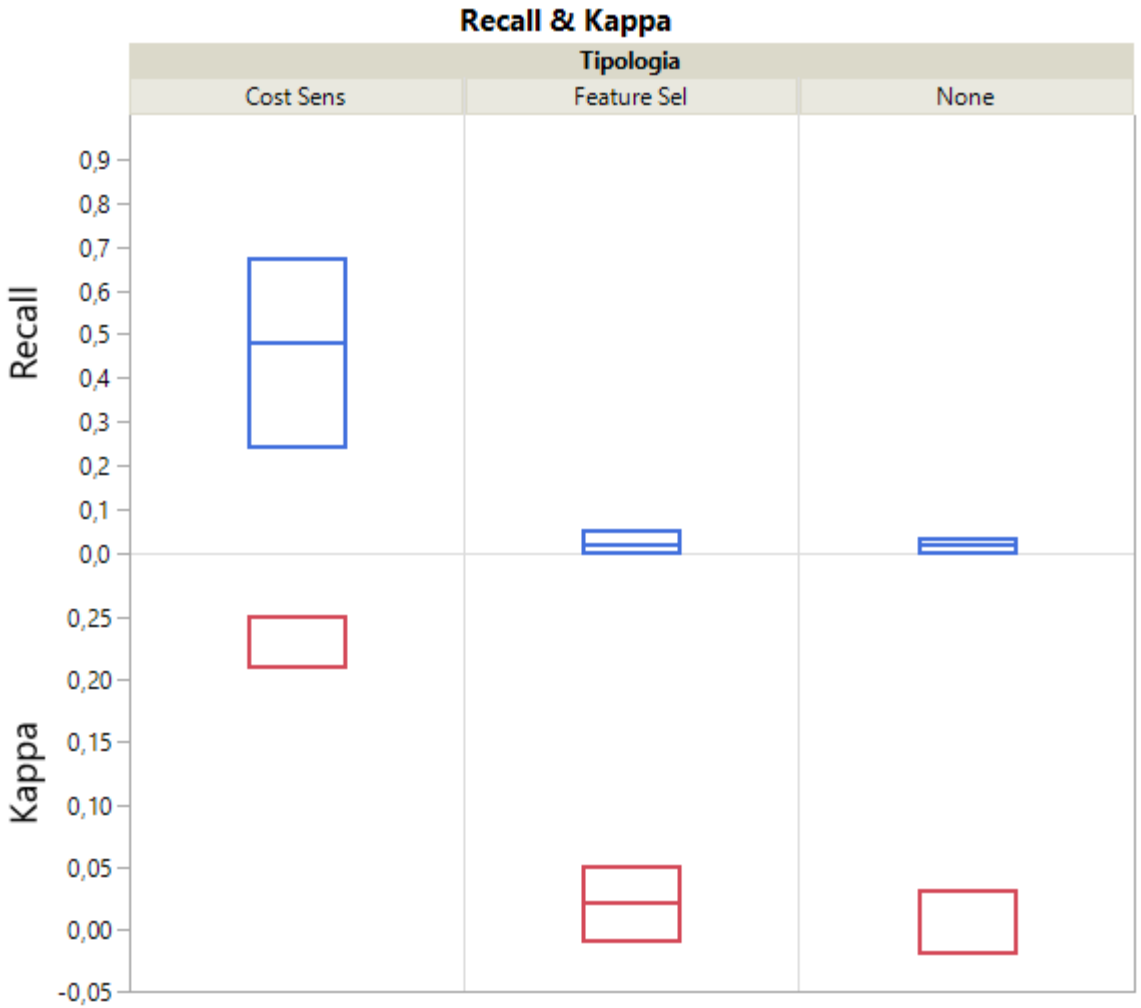


- Con il **Cost Sensitive** il **Multilayer Perceptron** sembra il **migliore**. Anche Random Forest sembra dare buoni risultati. Nel complesso finora **Random Forest è sembrato il classificatore migliore** (quando si usa il balancing). Analizziamolo meglio.

4. RISULTATI

BookKeeper: Random Forest con Cost Sensitive, FS, None

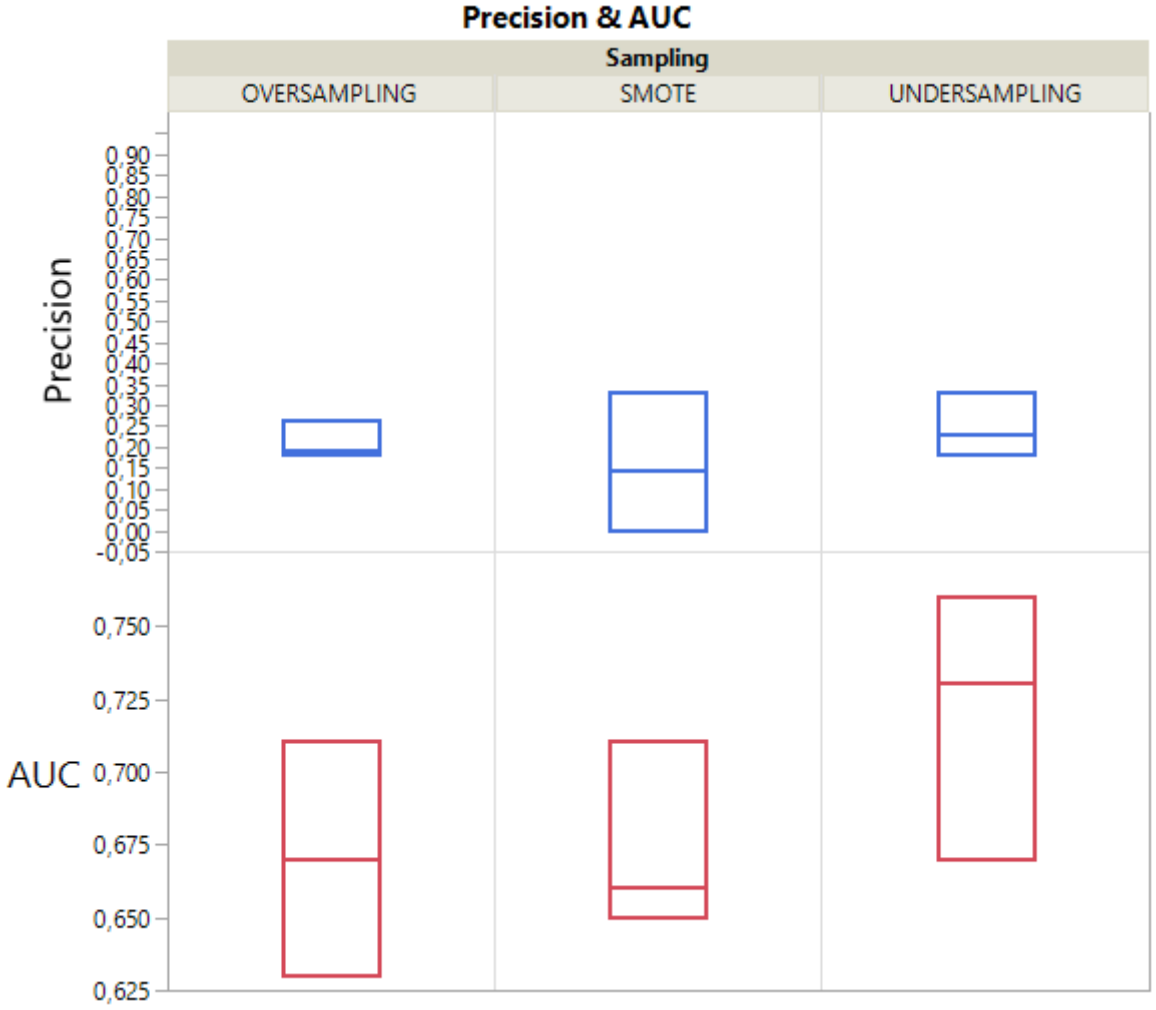


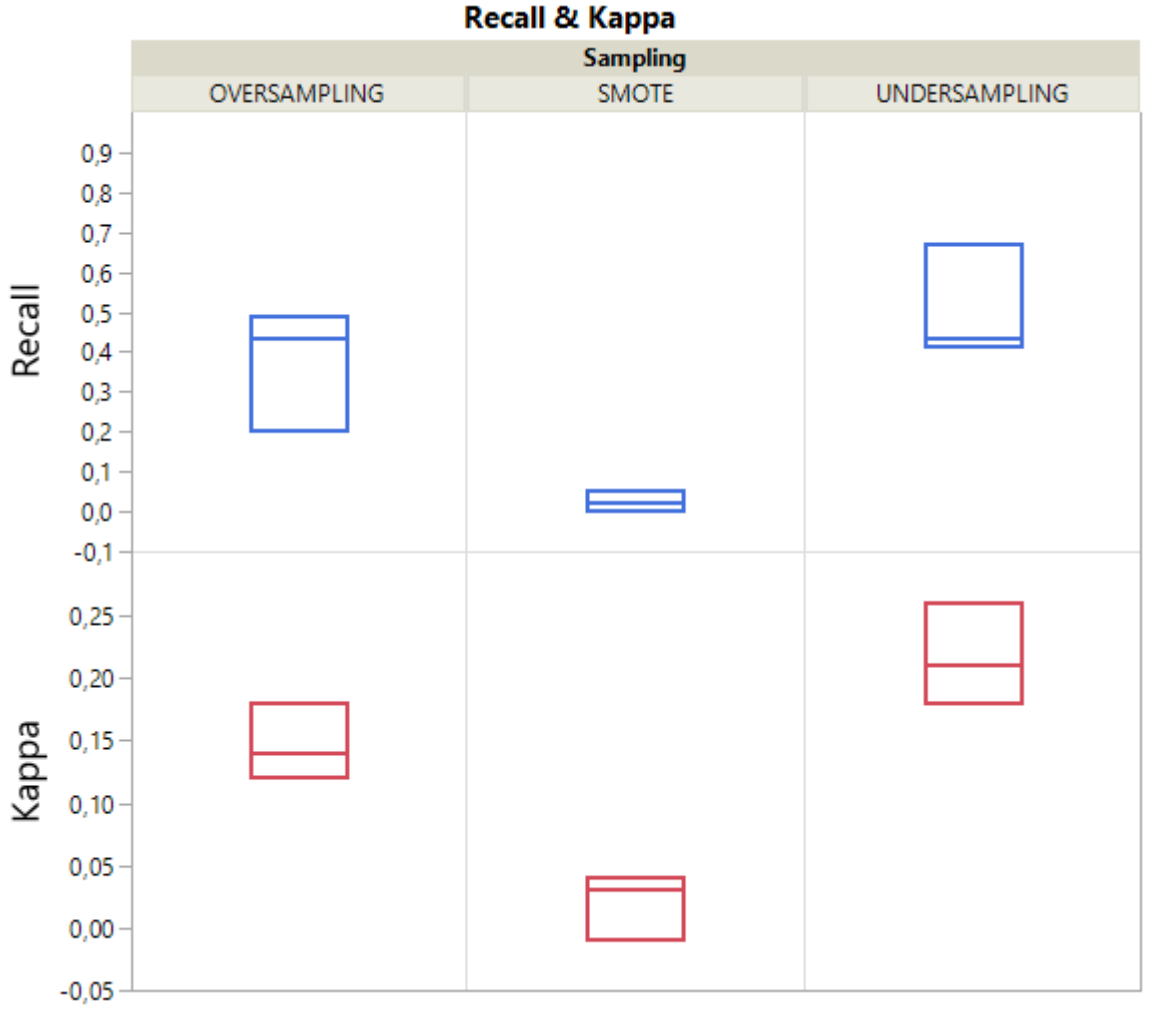


- **Random Forest** si comporta molto bene con **Cost Sensitive**. Raggiunge un **buon valore** di **recall** e la precision ha un degrado quasi nullo. Anche il valore kappa è molto buono con CS.

4. RISULTATI

BookKeeper: Random Forest con Balancing (e Feature Selection)

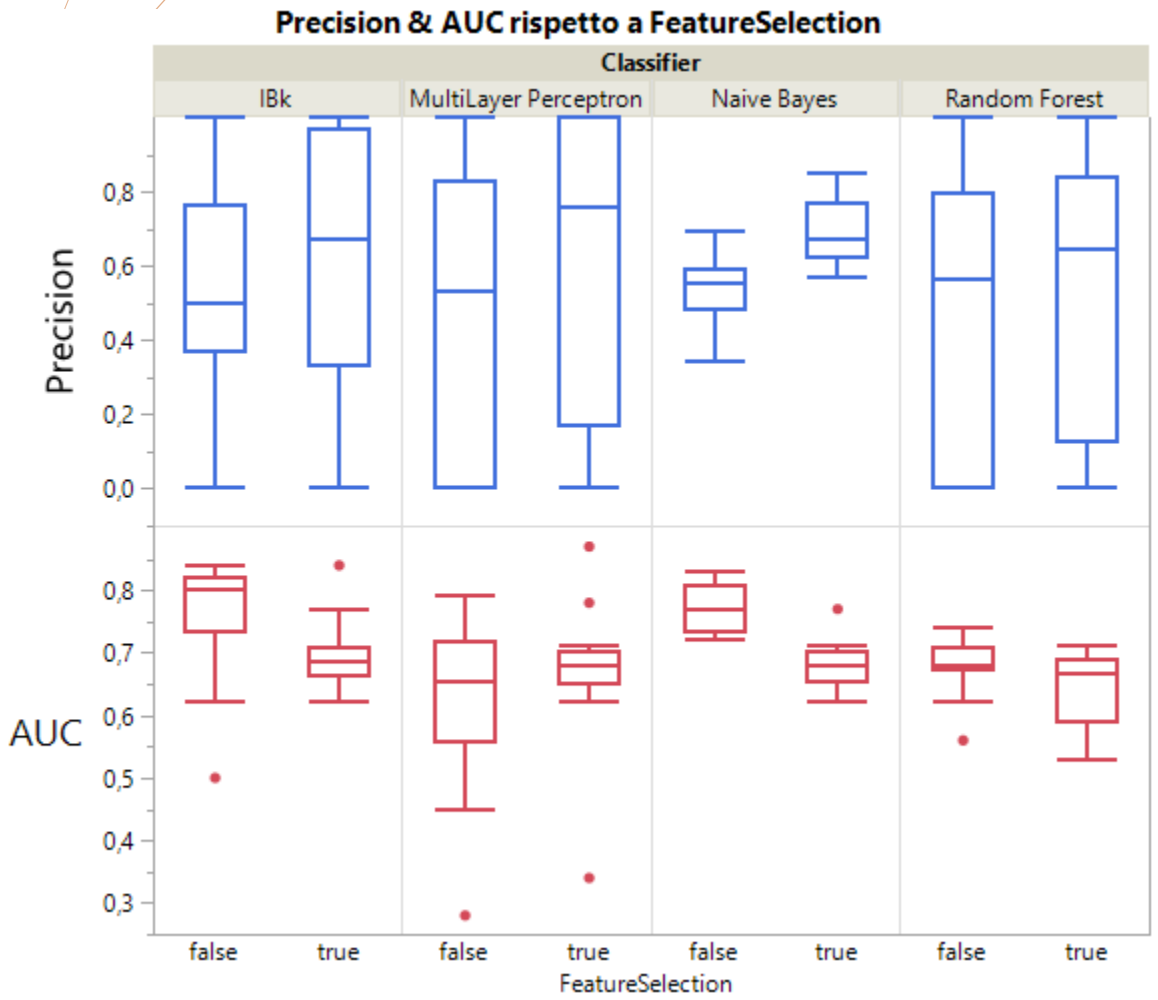


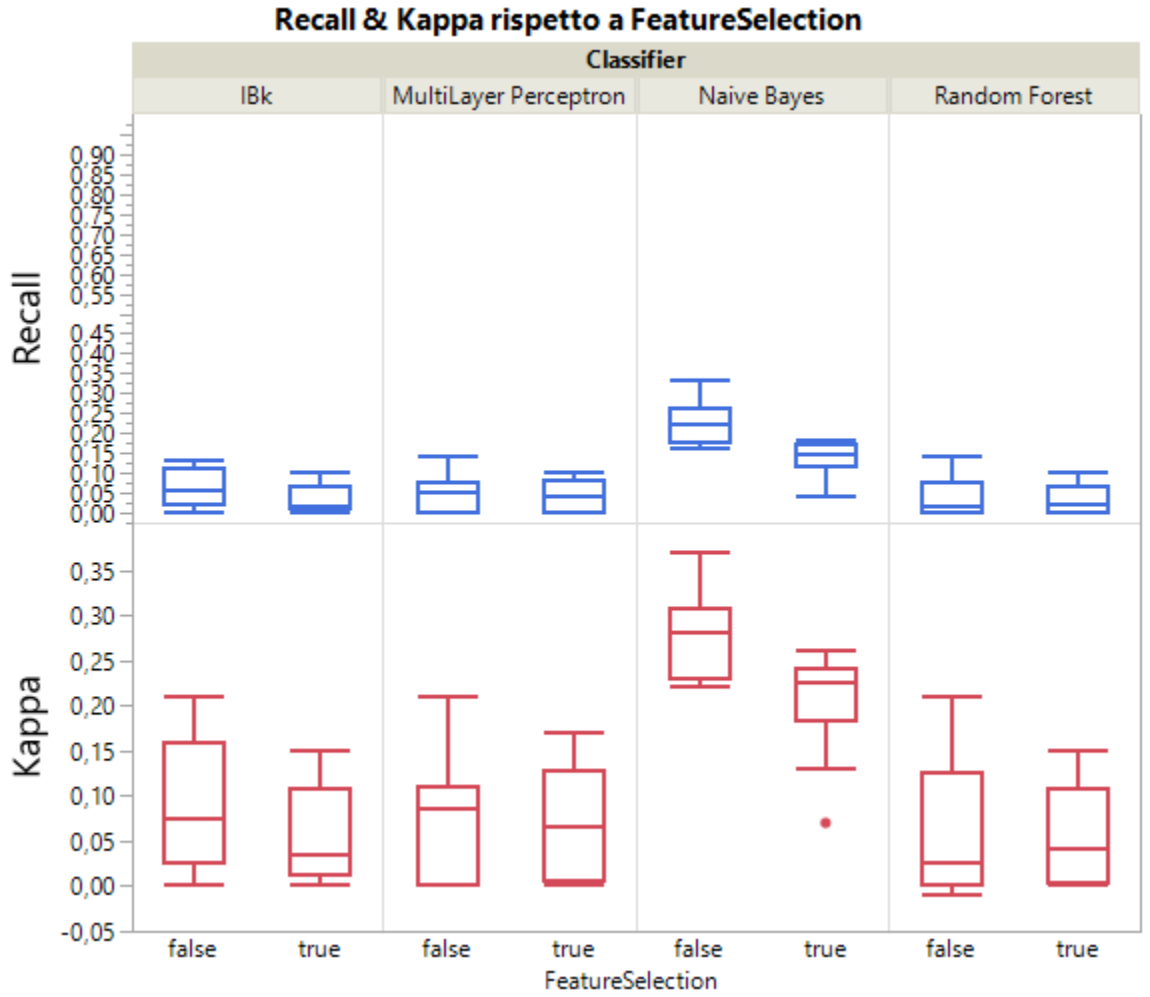


- In conclusione, **Random Forest con Undersampling e Feature Selection sembra essere la combinazione migliore**. L'impiego di cost sensitive resta una valida alternativa.

4. RISULTATI

OpenJPA: No balancing. No FS vs FS

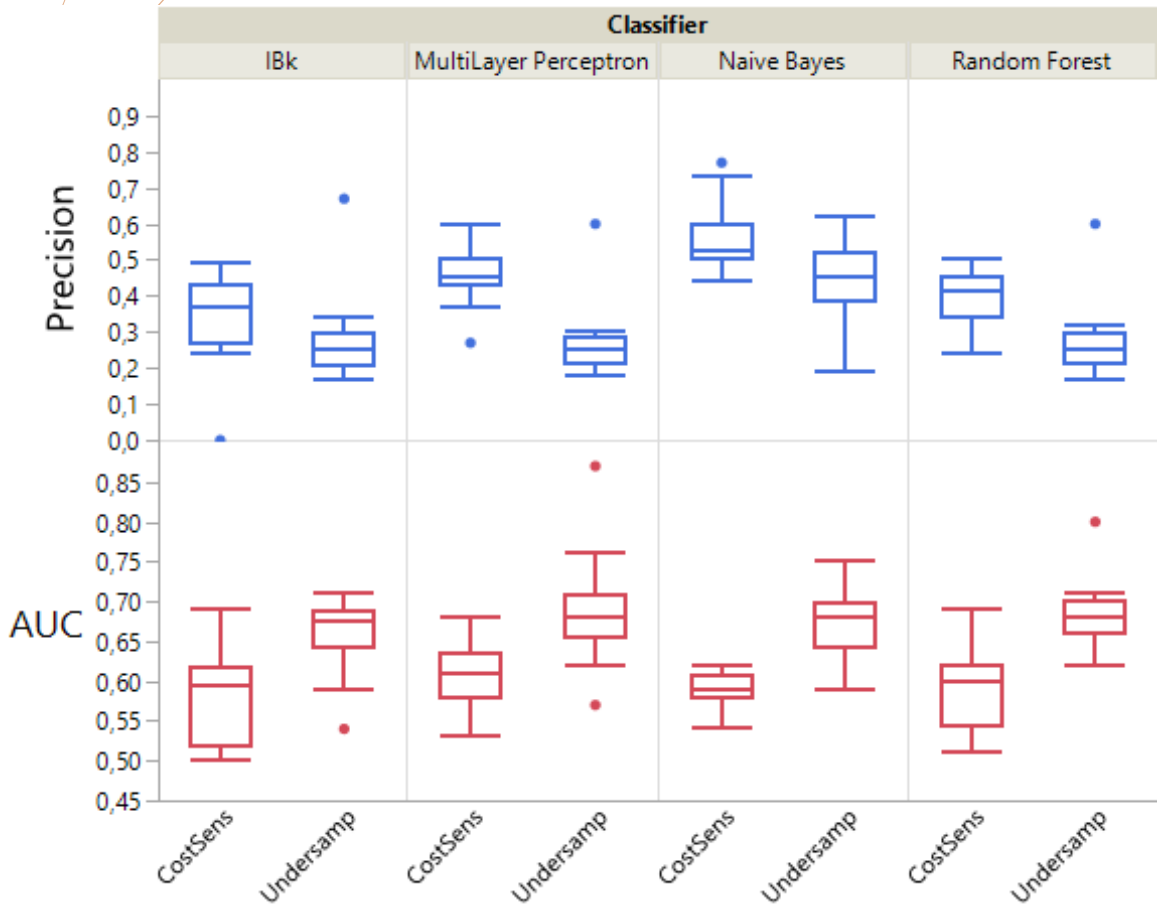


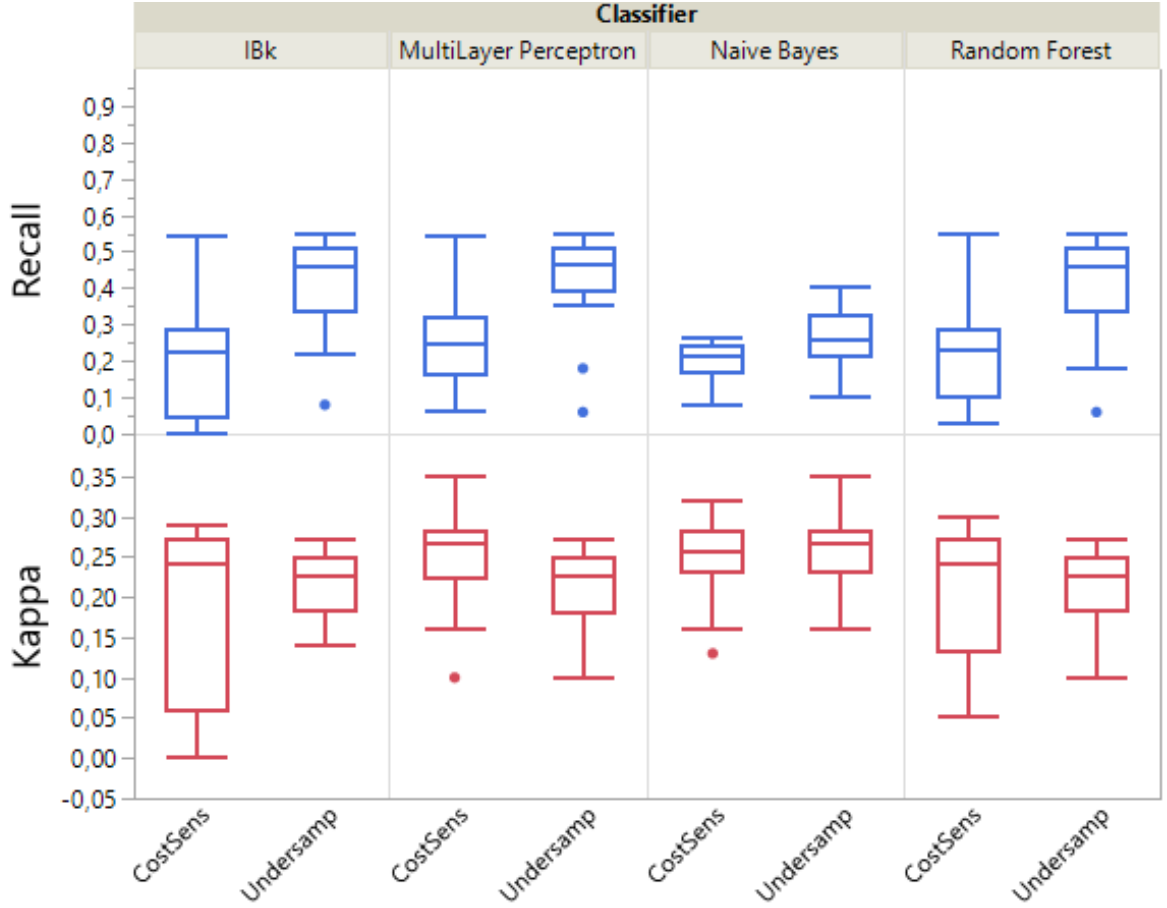


- Sul dataset di OpenJPA la feature selection sembra avere un impatto minore rispetto a quello di BookKeeper. Come ci si aspetterebbe, le variazioni della recall sono minime. Anche qui **Naive Bayes**, senza balancing, sembra essere il migliore.

4. RISULTATI

OpenJPA: Feature Selection. Undersampling vs Cost Sensitive



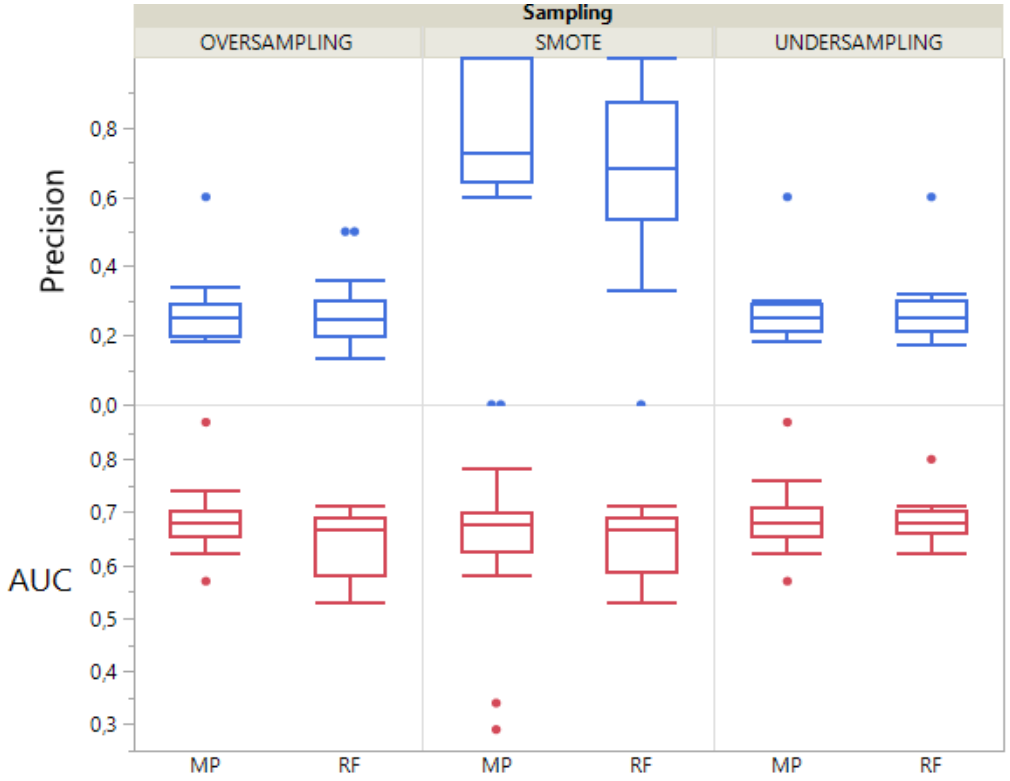


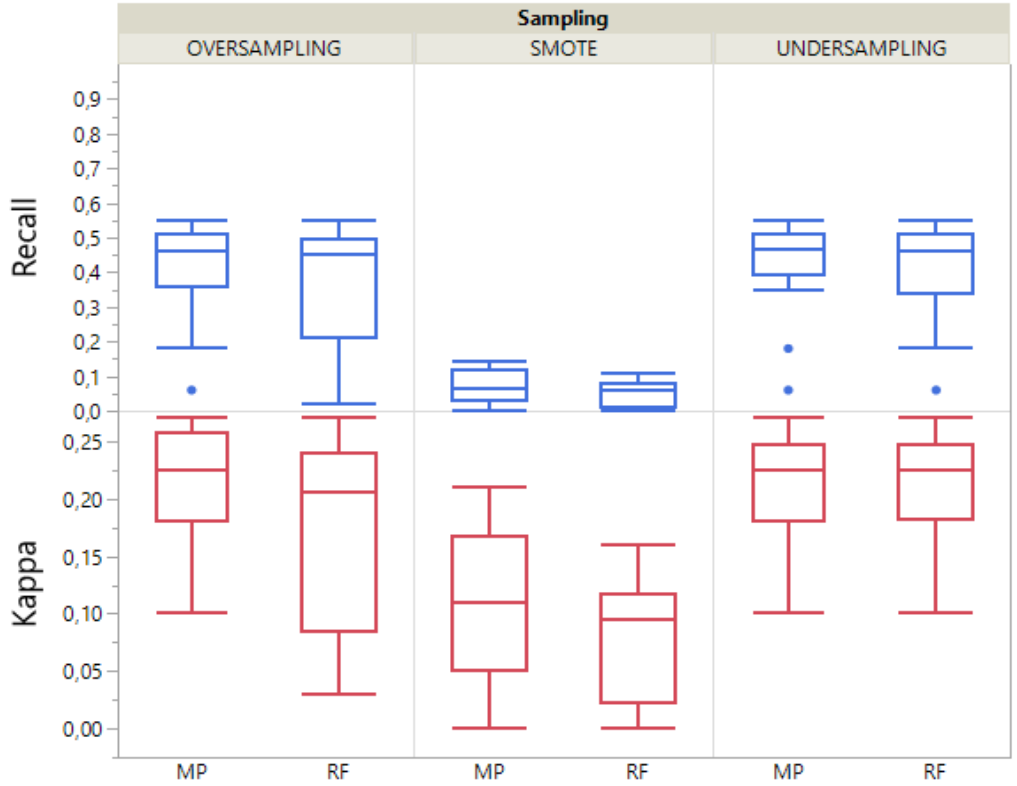
- Il balancing **migliora nettamente la recall**. In media **l'undersampling** sembra essere più **efficace** del cost sensitive, soprattutto per recall e AUC. Il **Multilayer Perceptron** (MP) sembra essere il migliore. Anche **Random Forest** (RF) e **IBk** sembrano dare **buoni risultati**, molto simili tra loro. Naive Bayes è il meno efficace.
- Analizziamo meglio quindi MP e anche RF, dato che non si discosta molto da MP, e con BookKeeper si è dimostrato il classificatore migliore.

4. RISULTATI

OpenJPA: Sampling (con FS). Random Forest vs Multilayer Perceptron

- Abbiamo già osservato che l'utilizzo di questo dataset senza nessuna tecnica o con solo feature selection non produce risultati soddisfacenti. Analizziamo quindi le **varie tecniche di balancing**.





- SMOTE non migliora le prestazioni della recall. Under e Over sampling sembrano comportarsi meglio. In particolare **Undersampling** sembra essere quello che **produce i risultati migliori**. MP e RF si comportano in modo molto simile, ma il **MP sembra il più stabile**: le recall minime sono meno basse rispetto a RF.

4. RISULTATI

BookKeeper & OpenJPA: Conclusioni

BookKeeper:

- Per questo dataset il classificatore **migliore** si è rivelato essere **Random Forest**.
- La tecnica che ha portato ai risultati migliori è stata l'**Undersampling**. Cost Sensitive rimane comunque una valida alternativa.

OpenJPA:

- Per il dataset di OpenJPA i classificatori **Multilayer Perceptron** e Random Forest hanno prestazioni vicine, ma MP sembra il **migliore**.
- La tecnica migliore da usare si è dimostrata essere l'**Undersampling**.

Conclusioni generali:

- **Random Forest** e **Multilayer Perceptron** sembrano essere ottimi classificatori. RF sembra avere buoni risultati su entrambi i dataset.
  La **tecnica migliore** è stata **l'undersampling**.
  Il cost sensitive resta una valida alternativa, magari con un CFN più alto potrebbe essere più efficace.
- **Naive Bayes** è il modello che si comporta (leggermente) **meglio** quando il **dataset** è **molto sbilanciato** (ovvero quando non si usa sampling/cost sensitive).
  Random Forest e IBk spesso hanno prestazioni quasi identiche.
- Come previsto, la **feature selection** non ha grande impatto sull'accuratezza, ma aiuta a migliorare le **performance** del training dei modelli.

4. RISULTATI

Metriche: Conclusioni

```
Ranked attributes:
 0.376    1 LOC
 0.353   12 CYCL_COMPLEX
 0.331   11 NFIX
 0.256    2 NR
 0.256    3 N_AUTH
 0.222    9 AVG_CHURN
 0.189    8 MAX_CHURN
 0.181   10 HND_EXCEPT
 0.168    7 CHURN
 0.129    6 AVG_LOC_ADDED
 0.12     4 LOC_ADDED
 0.103    5 MAX_LOC_ADDED
```

1: Iterazione 18 OpenJPA

```
Ranked attributes:
 0.40578   11 NFIX
 0.27796    2 NR
 0.25328    3 N_AUTH
 0.08205   10 HND_EXCEPT
 0.05739    1 LOC
 0.05159    7 CHURN
 0.04652   12 CYCL_COMPLEX
 0.02819    9 AVG_CHURN
 0.0236     6 AVG_LOC_ADDED
 0.01487    5 MAX_LOC_ADDED
 0.00576    8 MAX_CHURN
 0.00517    4 LOC_ADDED
```

2: iterazione 3 di BookKeeper

I risultati prodotti dai classificatori sono fortemente influenzati da quali metriche si utilizzano. La gran parte delle metriche scelte sono già analizzate nel paper [1]. Analizziamo quindi le due aggiuntive: **handledExceptions** e **cyclomaticComplexity**.

Chiaramente la **correlazione** tra la **buggyness** della classe e ogni **metrica varia** leggermente cambiando progetto e cambiando iterazione. Vediamo due esempi:

- Nella fig. 1 vediamo come la complessità ciclomatica sia la seconda metrica con la correlazione più alta rispetto alla buggyness. Questo conferma l'ipotesi fatta quando si è aggiunta la metrica nel progetto: classi con **più percorsi** di esecuzione indipendenti sono più **difficili da testare** → più inclini ad avere **bug**.

- Nella fig. 2 vediamo che il numero di eccezioni di cui si effettua un catch ha una correlazione abbastanza buona con la buggyness. L'assunzione di questa metrica è che in classi dove si devono gestire **più eccezioni**, ci sono più scenari d'errore da dover considerare → la **logica** del codice è più **complicata** → potenziale tendenza ad avere più **bug**.

- Le metriche standard ricalcano i comportamenti descritti dal paper [1].

5. ASSUNZIONI & MINACCE ALLA VALIDITÀ

I risultati mostrati sono stati ottenuti facendo delle **assunzioni**. Al variare di queste assunzioni potrebbero variare anche i risultati.

- In particolare, si sarebbero potuti ottenere risultati diversi con diverse scelte per:
  - I **Ticket validi per proportion**.
    Si è assunto che lo fossero quelli con OV<FV, e in cui le Affected Versions contenessero la OV ma non la FV. Si potrebbero usare vincoli più/meno stringenti.
  - Le tecniche di **proportion**: si sono usate cold start e increment. Usare moving window avrebbe potuto dare risultati diversi.
  - Le **metriche** scelte per la costruzione del dataset.
  - **Progetti diversi** avranno sviluppatori non in comune. Questo può influenzare lo snoring e l'utilizzo dei ticket (AV più o meno precise). Inoltre, progetti diversi producono dataset di dimensioni diverse.
  - Tecniche time-series diverse, per esempio usare **Moving Window** invece di **Walk Forward** potrebbe produrre risultati differenti.

- Un **CFN** diverso per il Cost Sensitive Learning sicuramente può influenzare (sia in meglio che in peggio) i risultati.
- Un **fine tuning** dei parametri dei **classificatori** potrebbe portare a comportamenti migliori di alcuni modelli, che in questo caso sono stati utilizzati in maniera **black-box**.
- Inoltre c'è sempre l'eventualità di avere un **bug** nel codice che produce i dataset.

6. LINK

Progetto & Fonti

- **GitHub:** codice Java per la realizzazione dei dataset e del training dei classificatori.

- **SonarCloud:** Pagina SonarCloud del progetto.

- [1]: Paper Proportion: Leveraging the Defects Life Cycle to Label Affected Versions and Defective Classes, Bailey Vandehei, Daniel Alencar Da Costa, Davide Falessi.

- [2]: Paper Snoring: The Impact of Dormant Defects on Defect Prediction: A Study of 19 Apache Projects, Davide Falessi, Aalok Ahluwalia, Massimiliano Di Penta.