Report di Progetto
ISW2 A.A. 2021/2022

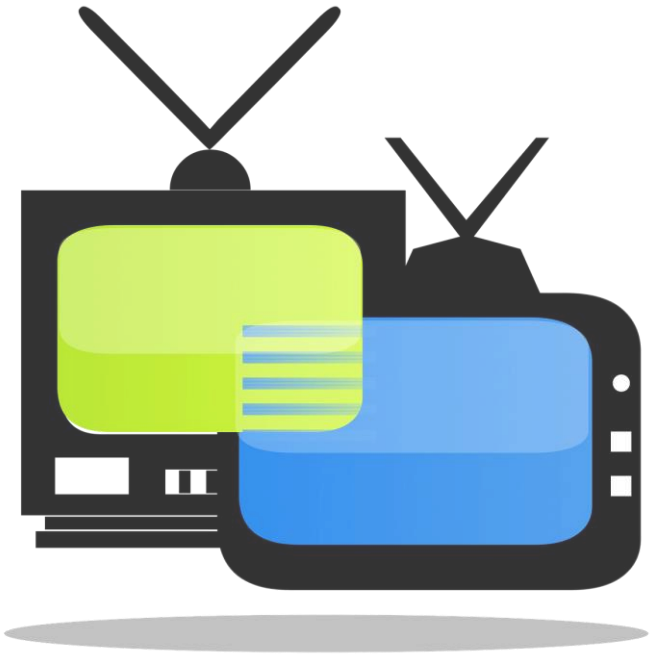
Machine Learning for Software Engineering Deliverable

GIANMARCO BENCIVENNI - 0285847

Contenuto della presentazione

In questa presentazione :

- Gli obiettivi dello studio
- Architettura dell'applicazione
- L'importanza dei dati e delle loro caratteristiche
- Data retrieval – merging Jira and Git
- Scelta e calcolo delle metriche
- Labeling
- Proportion increment
- Addestramento : walk-forward
- Tecniche di data processing : sampling e feature selection
- Evaluation : risultati
- Links e Informazioni



Gli Obiettivi dello studio

- **What**

  Lo studio si pone come obiettivo quello di **addestrare un insieme di classificatori** affinché siano in grado di **identificare**, in un certo istante del ciclo di vita di un certo progetto software, quali siano gli **artefatti software** che **più probabilmente sono difettosi** (*buggy*/*defective*).

  Sono previsti, a seguire, il confronto e la **valutazione delle prestazioni** di tali predittori, prendendo in considerazione i valori assunti dalle metriche di valutazione presentate durante il corso.

  Lo studio prevede l'applicazione pratica dei concetti di *Machine Learning for Software Engineering*, visti a lezione del Prof. Falessi, sui due progetti open-source:

  - Apache Bookkeeper
  - Apache Storm

Gli Obiettivi dello studio

- **Why**
  Un predittore correttamente addestrato può essere uno strumento estremamente utile ai fini della **riduzione dei costi** (in termini di tempo e di denaro!) **nelle attività di software testing e di revisione del codice**, permettendo ai teams di test e sviluppo di disporre di un **approccio mirato**, onde **evitare testing e ricerca estensivi**.

- **How**
  Il conseguimento degli obiettivi prefissati si snoda in alcuni **passaggi** base, che andremo ad affrontare in maggiore dettaglio nel corso della presentazione.
  - **Creazione del Dataset** : raccolta, elaborazione (scelta ed implementazione delle ***features*** di input al modello di machine learning) ed organizzazione dei dati utili all'addestramento e alla valutazione dei predittori.
  - **Pre-processing** : applicazione di tecniche di pre-processing sui dati (***sampling, feature selection***).
  - **Training** : addestramento dei modelli di predizione, secondo la tecnica ***walk forward***.
  - **Evaluation** : valutazione e confronto dei predittori, attraverso ***metriche di valutazione***.

Architettura dell'Applicazione 1/2

- L'applicazione sviluppata per questo studio è basata sul design pattern architetturale **BCE**.
  Si propone una vista ad alto livello sulla responsabilità delle classi principali implementate.
- Sono state sviluppate tre principali classi di **boundary**, ovvero:
  - GitRepositoryManager : questa classe ha la responsabilità di interfacciarsi, sfruttando la libreria **jGit**, alle git repositories dei progetti Apache presi in esame. Qualunque comunicazione con i logs, che sia volta a raccogliere *commit objects* o informazioni ad essi associati ai fini del calcolo delle metriche (features), è gestita tramite chiamate **Rest** da questa classe.
  - JiraTicketManager : questa classe ha la responsabilità di interfacciarsi, tramite **Rest API**, al server di **Jira**, al fine di reperire tutte le informazioni relative agli issues (di tipo **bug**) che sono stati **chiusi**.
  - DatasetBuilder : questa classe di I/O ha la responsabilità di inizializzare e di popolare i file csv contenenti i dataset, solamente dopo che questi siano stati generati nella business logic e mantenuti in forma di `MultiMap` (mappa ordinate in cui la chiave è una tupla di due elementi `(filename,version)`).

Architettura dell'Applicazione 2/2

- Sono state implementate due principali classi **controller**, quali:
  - IssueLifeCycleManager : questa classe è il controllore per la logica di business legata alla costruzione del dataset, a partire dal retrieval dei dati grezzi, fino al raffinamento e all'elaborazione delle metriche di input, al labeling e alla scrittura delle entries sul file csv.
  - ClassifierModel : questa classe è il controllore responsabile per la seconda componente dell'applicazione, quella relativa all'addestramento dei predittori (Ibk, Random Forest, Naive Bayes, Cost Sensitive) e alla loro valutazione.
- Le classi di **entity** sono:
  - IssueObject : rappresenta il singolo **issue**, che è definito principalmente attraverso gli attributi ID, IV, OV, AVs, FV, commits[].
  - CommitObject : rappresenta la singola **commit**, che è definita attraverso gli attributi ID, referenceIssue, files[].
  - FileObject : rappresenta il singolo **file**, che è definito attraverso gli attributi **filepath**, **version**, (**metrics**), **buggyness**. Il file object è l'entità che rappresenta una entry del dataset, una volta popolato in tutti i suoi attributi.

L’importanza dei dati e delle loro caratteristiche 1/2

L’**affidabilità di un predittore** dipende in larga parte dalla **qualità del dataset**.
In questo studio, si è scelto di raccogliere i dati necessari all’addestramento del predittore secondo un **approccio within-project**, prendendo quindi in considerazione le classi appartenenti al solo progetto software per cui si vogliono effettuare le predizioni sulla difettosità delle classi.
Questa scelta tiene in considerazione i seguenti **punti cardine**:

I. «**Size matters**» : Entrambi i progetti software considerati presentano un cospicuo numero di classi ed un volume di dati molto elevato, il che permette di generare dei dataset di dimensioni ragguardevoli (senza dover ricorrere all’approccio *across-project*). Ai fini dell’addestramento di un predittore, il dataset rappresenta un campione rappresentativo di una popolazione ben più ampia, il che implica che **tanto più è ampio (e vario!!!)** il dataset, **tanto più accurato sarà il predittore**.

II. **Maturazione dei dati** : Entrambi i progetti (open-source) esistono e vengono modificati da diversi anni a questa parte. Ad oggi, il numero di release è sufficientemente elevato da poter assumere un **buon grado di maturità** per entrambi i progetti, e questo è un requisito fondamentale per poter generare un dataset che contenga il **meno rumore** (**noise/garbage**) possibile. Vediamo meglio di cosa si tratta nella prossima slide.

L’importanza dei dati e delle loro caratteristiche 2/2

III. **«Garbage-in… Garbage-out!»** :
Un bug è un qualcosa che viene iniettato, in un certo istante, durante il ciclo di vita del progetto software. Il fatto che un bug esista in una classe, non significa che questo sia stato rilevato! Possono trascorrere intere release prima di sperimentare un fallimento e di marcare la classe come «*Buggy*».
Per questa ragione, **un dataset troppo giovane** (immaturo) è un dataset che **è molto probabilmente ricco di garbage**. Addestrare un predittore con dati rumorosi produrrebbe ovviamente dei **risultati altrettanto inaccurati**.

IV. **Fenomeno dello Snoring** :
Una snoring class è una classe in cui è stato iniettato un bug, per la quale tuttavia non sono state sperimentate failures. La presenza di **snoring classes** porta automaticamente alla produzione di **rumore nel dataset**, e perciò si vuole evitare questo problema.

Per **ridurre al minimo il fenomeno dello snoring**, si è scelto di **generare le etichette** (label) per i due dataset **considerando tutte le revisioni**, **ma** di **utilizzare per l’addestramento** solamente la porzione di dataset relativa alla **prima metà delle release**, poiché i **dati** sono **più maturi** e meno affetti da snoring.

Data retrieval – Merging Jira and Git 1/2

Due componenti fondamentali per la fase di costruzione del dataset sono:

- **Jira** : è l'**Issue Tracking System** utilizzato per entrambi i progetti Apache.
- Jira permette di **risalire a tutti gli issues (di tipo bug) chiusi**, indicando:
  - La Opening e la Fixed Versions (**OV, FV**) per i tickets associati ai Bugs.
  - Un **ID univoco associato all'issue/ticket**.
  - Le Affected Versions (**AV**), ovvero le versioni del progetto software affette dal Bug associato al ticket.
- Il primo step effettuato al lancio dell'applicazione è il *retrieval degli issues* da Jira, tramite **API Rest**.
- Il **codice sviluppato** per questa operazione **è derivato dagli esempi forniti a laboratorio** dal Prof. Falessi. Le informazioni relative ad ogni issue vengono mantenute in un IssueObject, e l'insieme complessivo degli issues è diviso e mantenuto dal controllore in due array separati, **discriminando**:
  - **Issues aventi Affected Versions dichiarate in Jira**
  - **Issues non aventi Affected Versions dichiarate in Jira**

Data retrieval – Merging Jira and Git 2/2

- **Git** : è il **Version Control System** utilizzato per entrambi i progetti Apache.
  Interfacciandosi a git tramite la libreria jGit è stato possibile implementare l'**algoritmo** così articolato:
  - Per ogni issue, **ricercare le commit che contengono l'issue ID** all'interno del **commit-message**, popolando così l'array di commits che è mantenuto come attributo per lo stato dell'IssueObject in questione.
  - Criticità : Nel caso in cui la versione del software in cui ricade la commit dovesse essere maggiore rispetto alla FV dichiarata per l'issue in Jira**, il valore della FV** dell'IssueObject in questione **viene aggiornato alla versione dichiarata nella commit**. Questa operazione è una scelta progettuale volta a **risolvere le inconsistenze** tra le informazioni dichiarate in Jira e in Git : il *versioning* in git è un processo automatizzato, mentre in Jira avviene in modo manuale. Questo passaggio fa sì che la **FV** di un issue **corrisponda alla versione dell'ultima commit che contiene l'issue ID** nel messaggio, che è quella **relativa al bug fix**.
  - per ogni commit, acquisire i files toccati, filtrando in modo da ottenere esclusivamente files in **formato «.java»**.
  - Per ogni file, istanziare un'entità **FileObject** definita da (filepath,version) e **calcolarne le metriche** (che specificheremo a breve)

Scelta e calcolo delle Metriche 1/2

Delle **metriche** viste a lezione, sono state scelte e calcolate le seguenti:

- Number of Revisions : numero di revisioni associate al file. Si è scelto in questo caso di registrarne il valore cumulativo, quindi non azzerando la metrica da una release alla successiva.
- Age : età del file, che in questo caso è stata misurata in settimane, a partire dalla creazione.
- Churn : somma, calcolata su tutte le revisioni, di LOCadded – LOCdeleted.
- Number of Authors : numero di autori che hanno lavorato sul file. Si è scelto in questo caso di registrarne il valore cumulativo, quindi non azzerando la metrica da una release alla successiva.
- LOC : valore medio, calcolato su tutte le revisioni di ogni release, delle linee di codice presenti nel file.
- LOC touched : somma, calcolata su tutte le revisioni, di LOCadded + LOCdeleted + LOCmodified.
- Average LOC added : valore medio delle linee di codice aggiunte al file, per release.
- Max LOC added : valore massimo delle linee di codice aggiunte al file, in per release.
- Average Change Set Size : valore medio di change set size nelle revisioni.
- Max Change Set Size : valore massimo del change set size nelle revisioni.

Scelta e calcolo delle Metriche 2/2

In aggiunta alle **metriche** viste in classe, ho deciso di **sperimentare un paio** metriche **aggiuntive** che ho pensato potessero essere delle caratteristiche **correlate alla buggyness di una classe**.

- Number of Imports : il numero di import nell'implementazione di una classe è legato al numero di moduli esterni che verranno utilizzati nel codice. Intuitivamente, quanto maggiori saranno i moduli da importare, tanto maggiore la complessità della classe, nonché la probabilità di iniettare difetti.
- Number of Comments : questa metrica ha secondo me un peso ancora maggiore rispetto alla precedente, e può portare un contributo importante alla classificazione soprattutto quando è accoppiata ad entries con un alto numero di LOC. L'assenza di commenti in una classe, infatti, la rende di difficile manutenibilità, e difficilmente estensibile senza introdurre difetti! Questo è vero soprattutto nelle classi con un alto livello di complessità (i.e. alto valore di LOC).

L'implementazione del calcolo delle metriche è consultabile nel codice sorgente della classe `GitRepositoryManager`, che si interfaccia direttamente ai logs delle commit dei due progetti.
Si mostrano a seguire alcuni esempi di codice in snippets, relativi proprio al calcolo delle metriche.

Code View


```java
public List<FileObject> computeMetricsAndAppendFile(CommitObject commitObject, List<FileObject>files, DiffEntry diff,

    int linesAdded = 0;
    int linesDeleted = 0;
    int linesReplaced = 0;
    RevCommit commit = commitObject.getCommit();

    if ( diff.getNewPath().endsWith( FILE_EXTENSION ) ){

        String  filepath = diff.getNewPath();
        String  fileText = getTextfromCommittedFile( commit, filepath );
        int     fileAge = getFileAgeInWeeks( commit, filepath );

        for ( Edit edit : df.toFileHeader( diff ).toEditList() ) {

            if ( edit.getBeginA() < edit.getEndA() && edit.getBeginB() < edit.getEndB() ){
                linesReplaced += edit.getEndB() - edit.getBeginB();
            }
            if ( edit.getBeginA() < edit.getEndA() && edit.getBeginB() == edit.getEndB() ){
                linesDeleted += edit.getEndA() - edit.getBeginA();
            }
            if ( edit.getBeginA() == edit.getEndA() && edit.getBeginB() < edit.getEndB() ){
                linesAdded += edit.getEndB() - edit.getBeginB();
            }

        }
        int     version = commitObject.getVersion();
        int     loc = getLoc( fileText );
        int     numImports = getNumImports(fileText);
        int     numComments = getNumComments(fileText);
        files.add( new FileObject(  filepath, version, fileAge, loc, linesAdded, linesDeleted, linesReplaced,
                                    changeSetSize, commitObject.getAuthorName(), numImports, numComments ) );
    }
```

Gli snippet sono tratti dal codice sorgente della classe `GitRepositoryManager.java`.
Lo **snippet di sinistra** mostra un estratto del metodo `computeMetricsAndAppendFile`.
Le metriche per un dato file vengono calcolate sul posto, nel momento in cui si ricercano i cambiamenti apportati dalla commit che si sta analizzando. Tramite un'istanza di **`DiffEntry`** è possibile risalire ai cambiamenti effettuati su di un file rispetto alla parent commit.
Nello snippet è possibile osservare parte della computazione delle varie metriche.
Ho voluto inserire il dettaglio relativo alla metrica «numero di commits», nello **snippet sottostante**, poiché la reputo un'idea interessante.

```java
public int getNumComments( String fileText ){
    int numComments = 0;
    try ( BufferedReader reader = new BufferedReader(new StringReader(fileText)) ) {
        for ( String line = reader.readLine(); line != null; line = reader.readLine()) {
            if (    line.contains("//") ||
                    line.contains("/*") ||
                    line.contains("*/") ||
                    !( line.endsWith("}") || line.endsWith("{") || line.endsWith(";") || line.equals("\n") || line.endsWith(")"))     ) {
                        numComments ++;
            }
        }
    } catch ( IOException e ) {
        e.printStackTrace();
    }
    return numComments;
}
```

Labeling : IV, OV, AV, FV

Il **prossimo step** è quello di **etichettare** i FileObjects (che saranno le **entry del dataset**) come **Buggy** o **Not Buggy**.

- **Ricapitolando**, nella struttura dell'applicazione :
  - ogni Issue mantiene una coda di Commit ad esso relative;
  - ogni Commit mantiene una coda di Files in essa modificati, ed una versione del progetto software (associata alla commit date);
  - Ogni File è rappresentato da un nome, una versione, delle metriche, e un valore relativo alla **buggyness**
- Come visto più volte a lezione, **una classe, in una certa versione, deve essere classificata come buggy se la sua versione appartiene alle Affected Versions del ticket** di riferimento. In generale, le Affected Versions partono con la Injected Version e terminano con la versione precedente alla Fixed Version, che è la versione nel corso della quale il bug è stato fixato.
- Il problema è che, come anticipato, abbiamo una parte dei tickets per cui conosciamo IV,OV,AV e FV, ma ne abbiamo anche una di cui conosciamo solamente OV e FV.
- Per questa ragione, è necessario applicare un **metodo al fine di stimare le Injected Version** degli issues per cui Jira non ne riporta il valore. Come noto, esistono differenti metodi, ma quello che è stato utilizzato è il metodo **Proportion**, nella sua versione Incremental.

Proportion - Increment

- La tecnica **Proportion** consente in generale di risalire alle Affected Versions (stimando il valore della Injected Version) laddove in Jira non siano disponibili.
- L'**assunzione** (dimostrata in diversi articoli scientifici che il Prof. Falessi ci ha proposto durante il corso) è che **i difetti** abbiano un **ciclo di vita stabile in termini della proporzione del numero di AV prima del fixing**.
  1. **P** = (FV – IV) / (FV - OV)
  2. **Predicted IV** = FV – (FV – OV) · **P**
- La declinazione **Increment** di Proportion prevede, per ogni issue privo di IV/AV, di considerare il sottoinsieme degli issues (aventi IV/AV) che sono passati a «fixed» prima di esso, e calcolare il valore di **P** come la media delle proporzioni di questi ultimi.
  A seguire, si stima il valore della IV dell'issue tramite la formula (**2**).
- Nella slide a seguire, si propone **l'implementazione di Proportion Increment** sviluppata per questo studio.

Code View

Gli snippets sono tratti dal codice sorgente della classe `IssueLifeCycleManager.java`
Nel corpo del metodo `setAffectedAndInjectedVersionP` (snippet sottostante) viene effettuata la stima di IV e conseguentemente AV per ogni ticket che ne è sprovvisto, tramite la tecnica **Proportion Increment**.

```java
public int computeProportionIncremental( List<IssueObject> filteredIssues ){
    ArrayList<Double> proportions = new ArrayList<>();
    for ( IssueObject issue : filteredIssues ){
        if ( issue.getOv() != issue.getFv() ) {
            double fv = issue.getFv();
            double ov = issue.getOv();
            double iv = issue.getIv();
            double p = ( fv - iv )/( fv - ov );
            proportions.add( p );
        }
    }
    return (int) average( proportions );
}
```

```java
public void setAffectedAndInjectedVersionsP(){
    for ( IssueObject targetIssue : issuesWithoutAffectedVersions ){
        int fv = targetIssue.getFv();
        int ov = targetIssue.getOv();
        List<IssueObject> filteredIssues = issuesWithAffectedVersions.stream()
            .filter(issue -> issue.getFv() <= fv)
            .collect(Collectors.toList());
        int p = computeProportionIncremental(new ArrayList<>(filteredIssues));
        if ( fv == ov ) {
            // The following formula is an approximation of the correct one (the one in the else block).
            targetIssue.setIv( ( fv - ( 1 * p ) ) );
            continue;
        } else{
            targetIssue.setIv( ( fv - ( ( fv - ov ) * p ) ) );
        }
        int minAVValue = targetIssue.getIv();
        int maxAVValue = ( targetIssue.getFv() - 1 );
        ArrayList<Integer> avs = new ArrayList<>( IntStream.rangeClosed(minAVValue, maxAVValue).boxed().collect(Collectors.toList()) );
        targetIssue.setAvs( avs );
    }
}
```

Ad ogni modo, come è possibile notare nella linea evidenziata in rosso, ho dovuto effettuare un'**approssimazione** nella stima di IV. Infatti, nei casi in cui il ticket per cui si vuole effettuare la previsione ha **FV=OV**, la formula per stimare il valore di IV «collassa» in: IV = FV.
Pertanto, in questi casi (comunque una minoranza), ho deciso di sostituire, con un certo margine di errore, il termine (FV-OV) con 1.

Addestramento – La Tecnica WalkForward

- Un ingrediente fondamentale di cui non abbiamo parlato è… il **tempo**!
- Il dataset costruito, non a caso, ha un **ordinamento per versione**.
  Si tratta infatti di una **time-series**, in cui l'ordinamento dei data points è una componente necessaria ai fini del corretto addestramento del predittore.
- Infatti, così come un predittore addestrato per un certo progetto software non avrà risultati ottimali per un progetto di altra natura, un predittore addestrato su dati relativi a un tempo futuro (seppur di uno stesso progetto) non avrà buone capacità di previsione su dati relativi al suo passato!
- La **tecnica di addestramento walk-forward** è designata a risolvere il problema dell'addestramento su serie temporali. La logica di addestramento è la seguente :
  - Suddivisione del dataset in K batch ordinati (in questo caso, si suddivide per versioni)
  - Addestramento iterativo (in K-1 passaggi) prendendo alla j-esima iterazione i primi j batch come training set, ed il j+1-esimo batch come test set.
  - La valutazione finale del predittore si ottiene mediando sui risultati delle metriche di ogni iterazione.
- L'implementazione della tecnica si trova nel codice della classe `ClassifierModel.java`, nel metodo `modifiedWalkForwardTrainingAndTest`

Tecniche di data processing
Sampling e Feature Selection 1/2

- Il modulo dell'applicazione che entra in gioco a valle della creazione del dataset, e che implementa la logica relativa a pre-processing dei dati, addestramento e valutazione dei predittori, fa uso intensivo dell'API di **Weka**, una libreria di Machine Learning per applicazioni Java.
- L'utilizzo di Weka mi ha permesso di implementare le tecniche di **sampling** (under-sampling, over-sampling, smote) e di **feature selection** (euristica *best-first*), così come la **tecnica di classificazione sensibile al costo**.
- **Sampling** : un problema ricorrente nell'addestramento e nella valutazione di un classificatore risiede nel fatto che spesso le classi nel dataset sono sbilanciate in modo pronunciato. Ciò si traduce nel fatto che il modello non avrà sufficienti esempi della classe minoritaria, e una volta addestrato non possiederà buone capacità di generalizzazione. Per questo studio ho trovato necessario applicare il sampling, proprio perché il numero di data points classificati come *buggy* sono comunque in forte minoranza in entrambi i progetti rispetto ai *not buggy*.
  - **Under-sampling** (API SpreadSubSample): diminuzione dei punti della classe maggioritaria
  - **Over-sampling** (API Resampling): aumento (per duplicazione) dei punti della classe minoritaria
  - **Smote** (API SMOTE): aumento dei punti della classe minoritaria attraverso la creazione di data points sintetici, a partire da quelli esistenti.

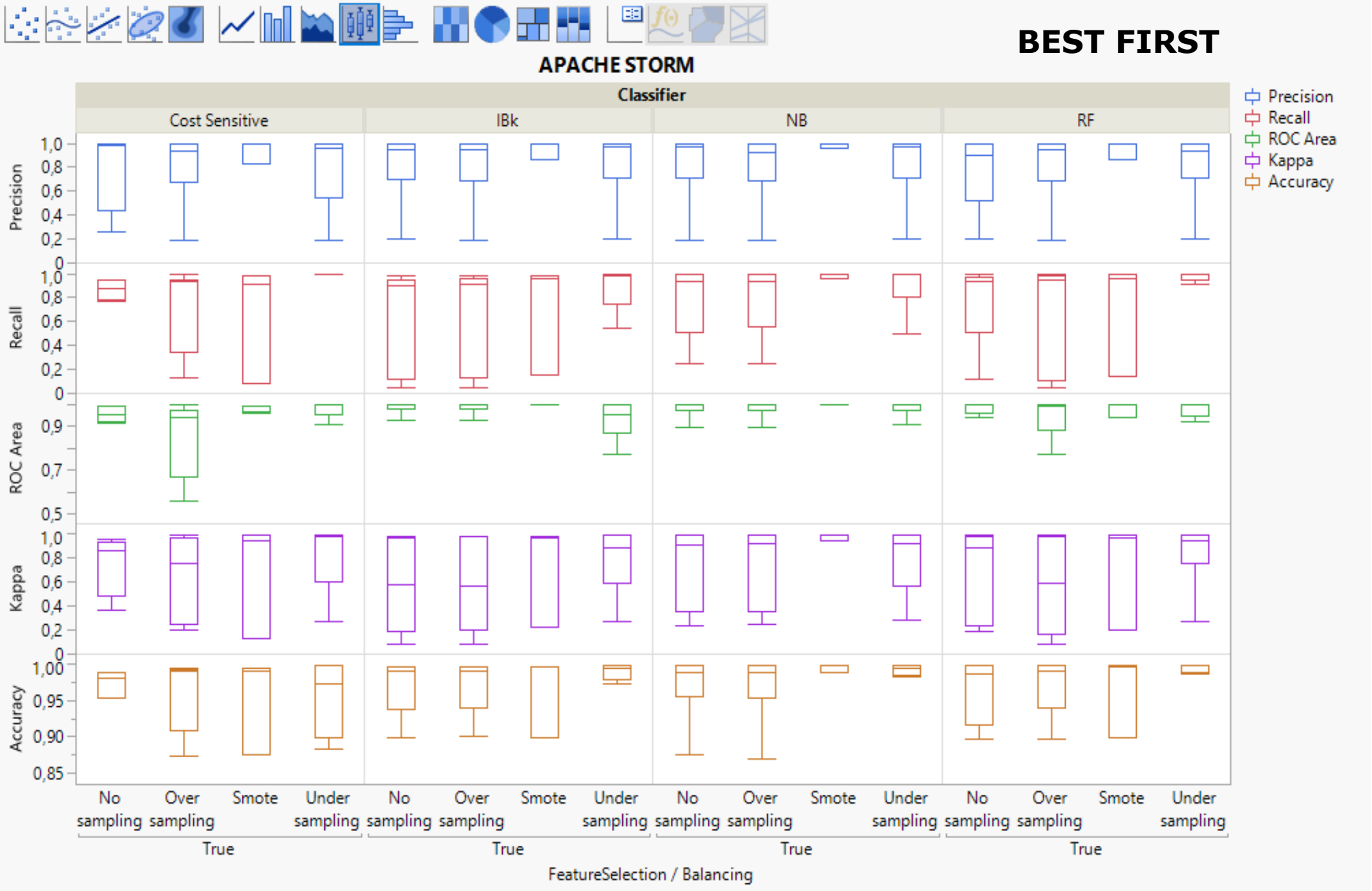
Tecniche di data processing
Sampling e Feature Selection 2/2

- **Feature Selection** : la tecnica della feature selection ha lo scopo di diminuire la complessità del modello (tagliando il numero di features di input si ha ovviamente un modello più semplice) e, teoricamente, di migliorare le performance del training rispetto al caso con l'intero set di features.
  L'approccio specifico utilizzato per lo studio è denominato best-first, e si basa su un algoritmo greedy.
  - **Best-First** : Si inizia creando una suite di N modelli, ognuno dei quali utilizza solo una delle N caratteristiche del nostro set di dati come input. Viene selezionata la caratteristica che produce il modello con le migliori prestazioni. Nell'iterazione successiva, viene creata un'altra suite di N-1 modelli con due caratteristiche in ingresso: quella selezionata nell'iterazione precedente e un'altra delle N-1 caratteristiche rimanenti. E così via.
- **Cost Sensitive Analysis** : la libreria weka permette anche l'utilizzo di un **Cost Sensitive Classifier** (**J48**).
  Data la semplicità d'implementazione, ho voluto aggiungere questo classificatore ai tre consigliati nelle specifiche, poiché ho trovato francamente molto interessante, soprattutto per questo tipo di studio, l'idea di un'analisi orientata ai costi.
  - In pratica, settando una **matrice di costo** in modo da apportare una specifica **penalità** all'occorrenza di falsi negativi (FN), durante l'addestramento del modello si avrà che ogni **errore di tipo FN** viene pesato tanto quanto indicato dall'utente!
  - Come diretta conseguenza il modello, una volta addestrato, dovrà superare un certo valore di soglia prima di poter classificare un data-point come negativo, perché è come se fosse conscio del fatto che è una scelta sensibile/pericolosa.

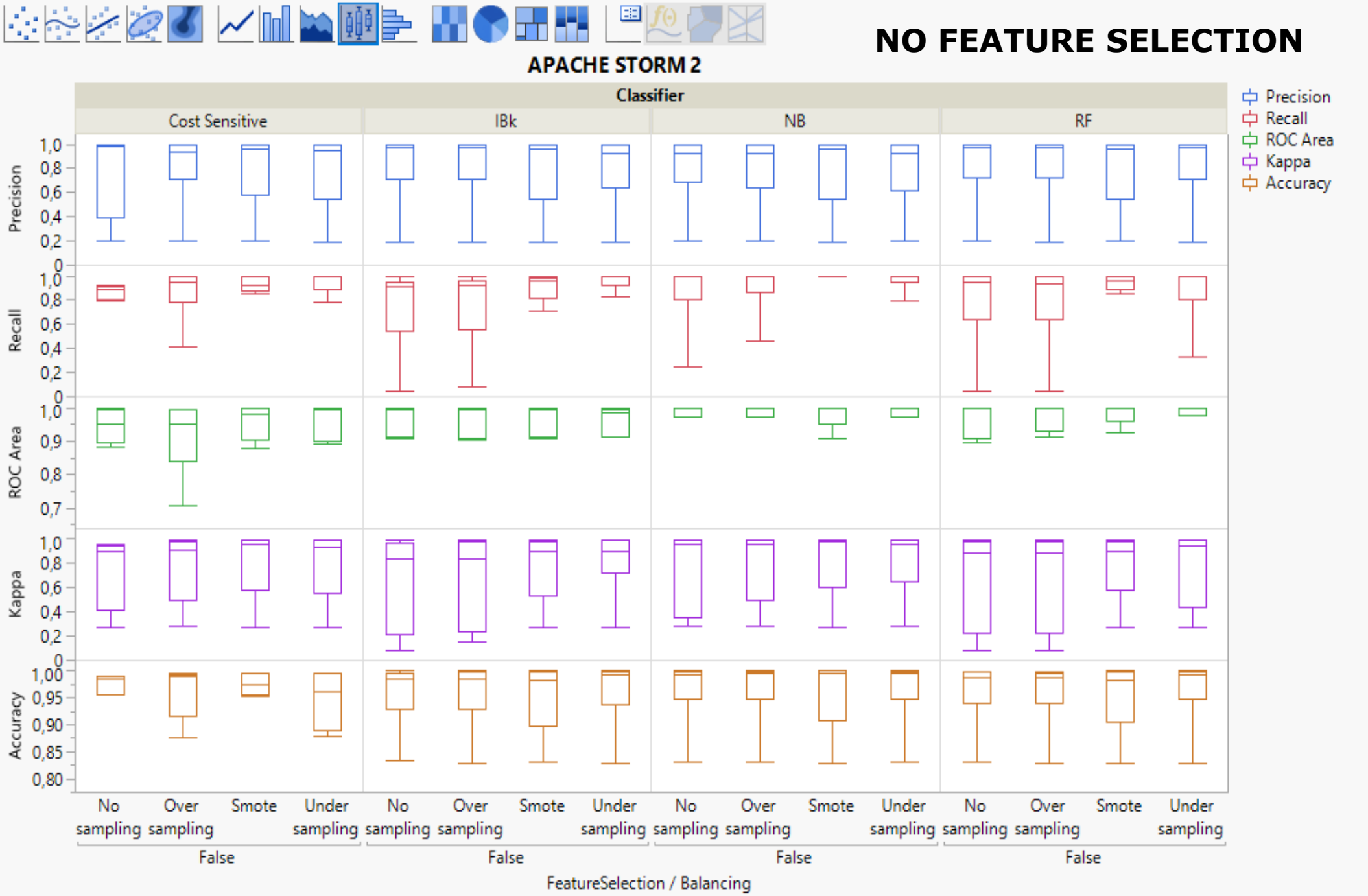
Valutazione dei modelli Ibk, RF, NB, J48

- I modelli di Machine Learning (classificatori) addestrati per lo studio sono:
  - **IBK**, **Random Forest**, **Naive Bayes**, Cost Sensitive Classifier (**J48**).
- La valutazione dei predittori tiene conto di alcune metriche che abbiamo affrontato a lezione, quali:
  - **Precision** : $\frac{TP}{TP+FP}$ (**quanti dei punti che ho classificato come positivi lo erano davvero?**)
  - **Recall** : $\frac{TP}{TP+FN}$ (**quanti positivi sono riuscito a «beccare»?**)
    - **Precision e Recall devono essere osservate insieme**! Se ho alta Precision e bassa Recall, significa che i punti classificati come positivi erano giusti… ma vuol dire che ho tanti falsi negativi! Viceversa, se la Recall è alta e la Precision bassa, significa che magari i positivi li ho beccati tutti… ma ho tanti falsi positivi!
  - **AUC** : è l'area sotto la curva ROC. Indica la probabilità che una istanza di positivo presa randomicamente sia classificata meglio di un'istanza di negativo presa randomicamente. In generale la vogliamo alta (>0.5), dato è molto più grave un falso negativo rispetto ad un falso positivo!
  - **Kappa** : indica quanto il classificatore si discosta da un classificatore Dummy (perfettamente randomico)
  - **Accuracy** : 1- error rate, indica la percentuale di punti classificati correttamente, indipendentemente dal fatto che questi siano classificati come positivi o negativi.

BEST FIRST


APACHE STORM
Classifier
Cost Sensitive
IBk
NB
RF
Precision
Recall
ROC Area
Kappa
Accuracy
No sampling
Over sampling
Smote
Under sampling
True
FeatureSelection / Balancing

NO FEATURE SELECTION



Risultati - Storm

- Analizziamo i risultati ottenuti per il **progetto Storm**, facendo **focus sui punti chiave**.

i. Problemi con Smote : il sampling di tipo smote ha portato in generale molti problemi, poiché molto spesso la classe minoritaria è risultata insufficiente per applicare smote correttamente, e i valori delle metriche sono stati settati a NaN, e perciò li ho dovuti scartare nella valutazione finale.
I box plot relativi a smote risultano infatti con variabilità molto «piatta» e sono in qualche modo «falsati».

ii. Feature Selection : la tecnica di feature selection, salvo pochi casi, si è rivelata **controproducente**, aumentando in particolare la **variabilità dei risultati sulla Recall**. La mia interpretazione di questo fatto è che, probabilmente, le features nativamente utilizzate apportavano migliori risultati in fase di addestramento se utilizzate assieme. Si può dedurre che le features avessero una **bassa inter-correlazione** (tra loro), ma una **buona correlazione con l'output** del predittore.

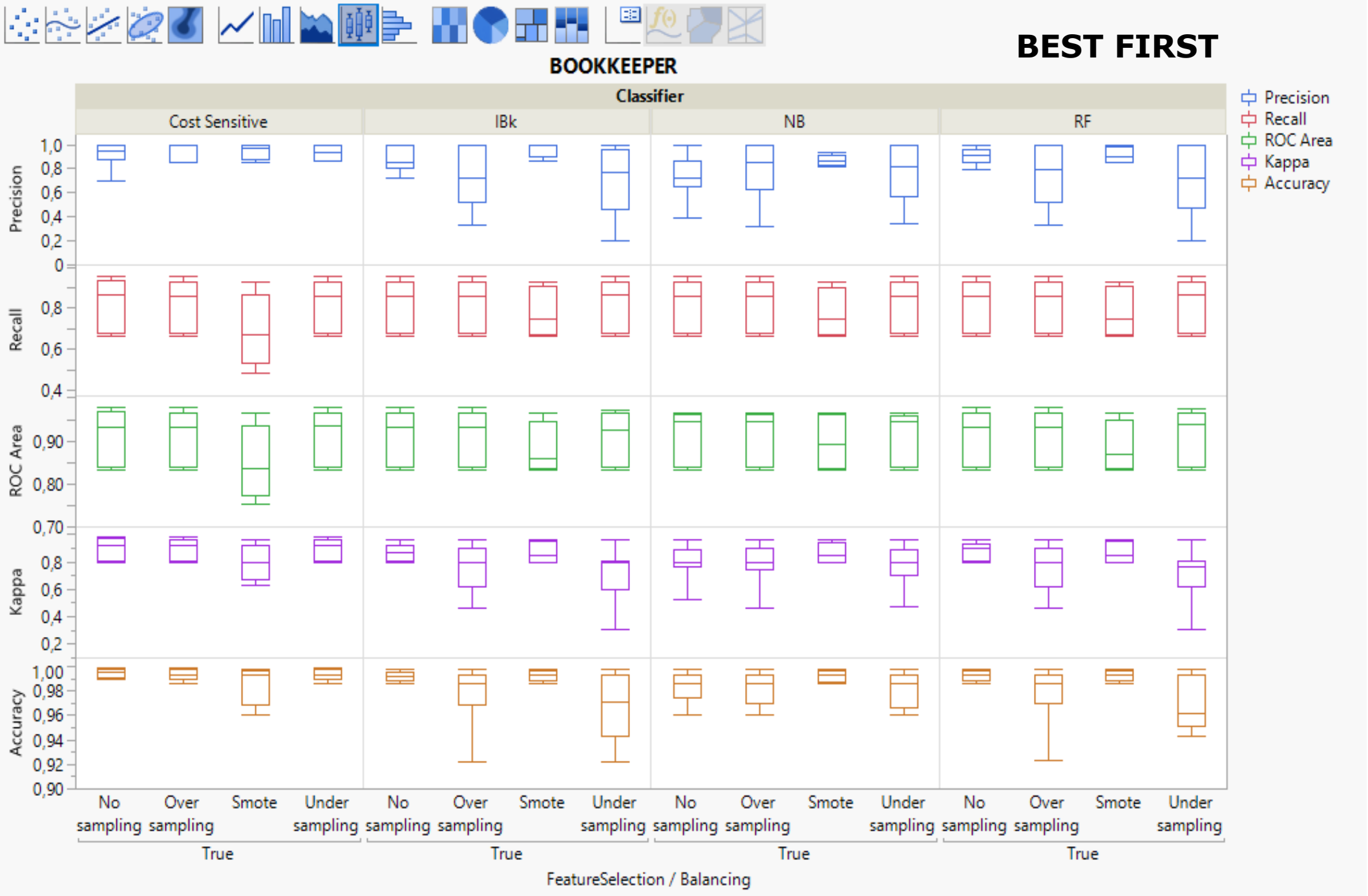
iii. No Silver Bullet : dai risultati ottenuti si evince chiaramente che **non esiste il proiettile d'argento**.
Alcuni classificatori, in determinate configurazioni, si comportano meglio di altri per alcune caratteristiche, e peggio per altre.

iv. Precision e Recall : a livello di Precision e Recall, la configurazione migliore è stata ottenuta dal **Cost Sensitive Classifier in assenza di feature selection**, e in particolare con **over-sampling**. Mi aspettavo un risultato di questo tipo, sebbene sia importante sottolineare che sto valutando la bontà del risultato **non solamente sui valori medi** assunti, bensì **anche** sul fatto che è importante avere una **bassa varianza sui risultati**, ai fini della validazione del modello.
È interessante notare anche che la **classificazione orientata ai costi produce alti valori di Recall**, ma inevitabilmente **impatta negativamente sulla Precision**, perché il modello è portato a classificare più positivi che negativi.
Volendo guardare alla Precision da sola, infatti, i risultati migliori sono stati ottenuti da Random Forest/NoFS.

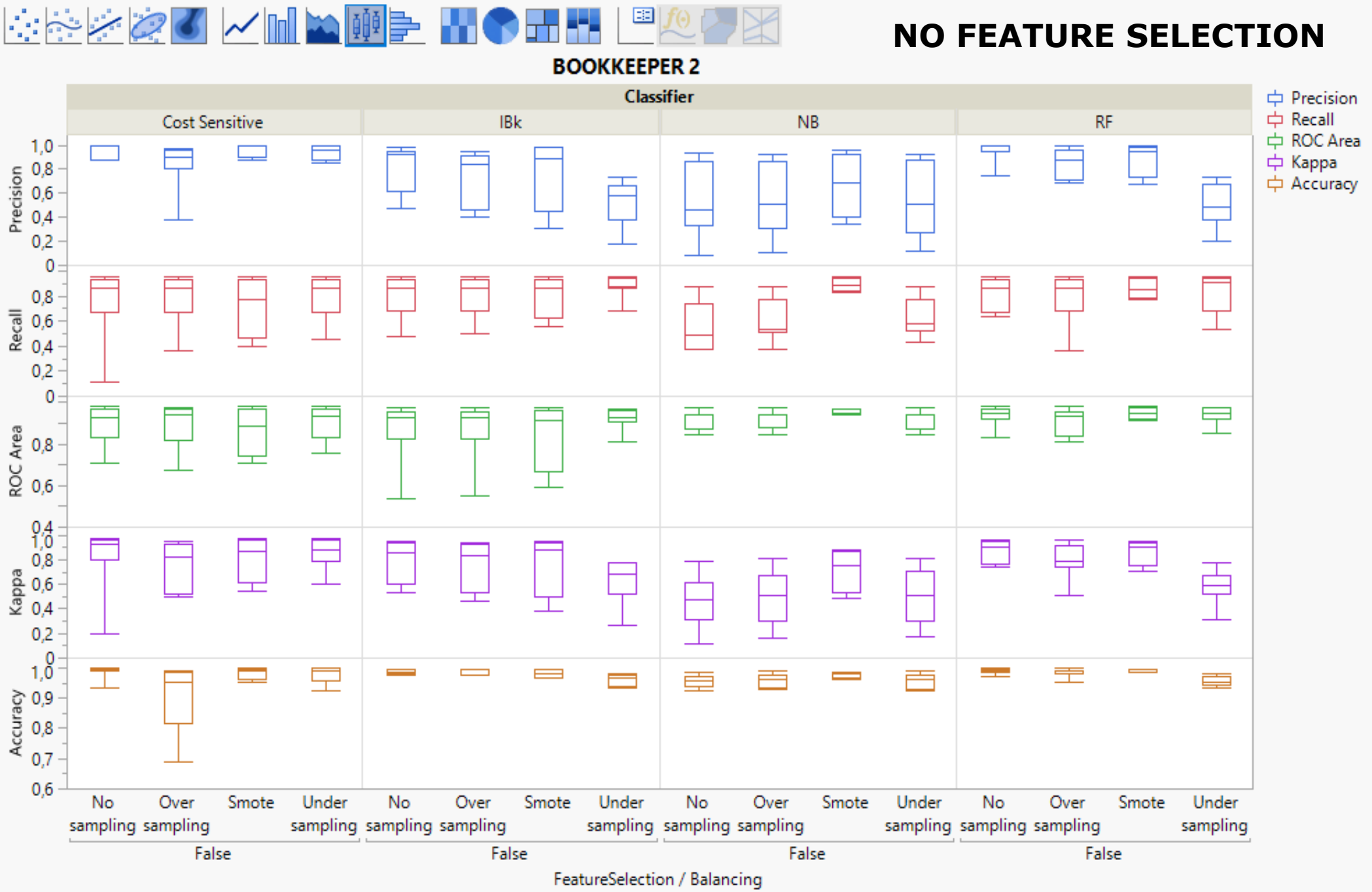
v. AUC : i risultati migliori a livello di AUC sono quelli ottenuti per **Naive Bayes/NoFS** e **Random Forest/NoFS**, in entrambi casi in modo analogo con l'applicazione dell'under-sampling o dell'over-sampling.

vi. Kappa : è interessante notare che, per la metrica di valutazione kappa, i risultati migliori sono stati ottenuti dal classificatore **Naive Bayes/NoFS**, pur essendo questo tra i peggiori nel contesto di Precision e di Recall.

BEST FIRST


BOOKKEEPER
Classifier
Cost Sensitive
IBk
NB
RF
Precision
Recall
ROC Area
Kappa
Accuracy
1,0
0,8
0,6
0,4
0,2
0
0,90
0,80
0,70
1,00
0,98
0,96
0,94
0,92
0,90
No sampling
Over sampling
Smote
Under sampling
True
FeatureSelection / Balancing

NO FEATURE SELECTION



Risultati - Bookkeeper

- Analizziamo i risultati ottenuti per il **progetto Bookkeeper**, facendo **focus sui punti chiave**.

i. Problemi con Smote : in questo caso i problemi con la tecnica smote sono stati minori, poiché solamente poche volte la classe minoritaria è risultata insufficiente per applicare la tecnica. Ad ogni modo, è stato comunque necessario eliminare qualche riga dal file di *evaluation* a causa dei valori NaN, come nel caso di Storm, e la variabilità sui valori con smote è comunque bassa. Ad ogni modo, **nel dataset di Bookkeeper la classe buggy era molto più rappresentata** rispetto a quanto non lo fosse in Storm.

i. Feature Selection : la tecnica di feature selection ha in generale avuto un impatto **di poca rilevanza**, per quanto riguarda le metriche Precision, Recall e AUC. Un netto miglioramento dato dall'utilizzo della feature selection è tuttavia osservabile sui valori medi e sulla variabilità della metrica Kappa, e questo vale per tutti i predittori.

i. Precision e Recall : anche in questo caso, a livello di Precision e Recall, la configurazione migliore è stata ottenuta dal **Cost Sensitive Classifier** a cui **non è stata applicata feature selection e nemmeno il sampling**.
Si può spiegare il risultato col fatto che, data la presenza di una migliore rappresentanza della classe buggy, in questo caso il sampling non è stato particolarmente necessario.

i. AUC e Kappa : i valori migliori per AUC e Kappa sono stati ottenuti con il predittore **Naive Bayes**, con feature selection **best first** e **no sampling**.

Info e Links

Ambiente di sviluppo e **frameworks** utilizzati :

- **VSCode** come IDE per lo sviluppo dell'applicazione
- **Maven** come framework per la gestione delle dipendenze e per la build del progetto
- **Java 11** come linguaggio di programmazione (principale)
- **Excel** e **JMP (GUI)** per lavorare dati e grafici

Istruzioni per **l'utilizzo del codice** :

- Scaricare il codice sorgente dal link alla github repository **https://github.com/CLOCKWORK95/ML_4_SE**
- Posizionarsi da terminale nella directory : `ML_4_SE/my-app`
- lanciare il comando bash : `sh buildAndLaunch.sh` e seguire le indicazioni da CLI

Link a **Sonarcloud** per la repository del progetto **ML_4_SE** :

- **https://sonarcloud.io/summary/overall?id=CLOCKWORK95_ML_4_SE**