REPORT ISW2

PACI ANDREA

MISURA DELLA STABILITA' DI UN ATTRIBUTO DI PROGETTO

PREDIZIONE DELLA DEFECTIVENESS DI FILE SORGENTE

Matricola: 0286387

Studente: Andrea Paci

Email: andrea.paci1998@gmail.com

Nota Preliminare

Il seguente report si pone l'obbiettivo di descrivere il processso di *analisi* e *studio* effettuato su progetti *open source* di **Apache**, utilizzando strumenti come lo **Statistical Control Process**, **Versioning Control Systems**, **Ticket Tracking** e **Machine Learning** per fornire risultati **quantitativi** e **statisticamente** significativi.

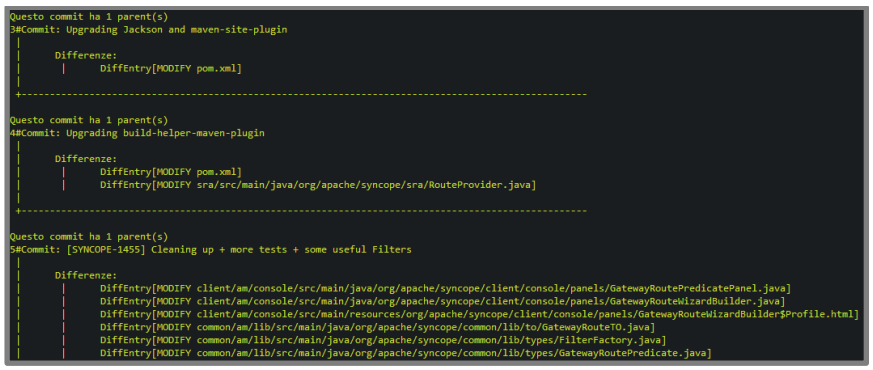
Lo studio è stato diviso secondo **due deliverables** principali, le quali verrano analizzate in maniera separata nei rispettivi paragrafi.

Tutti i risultati che verranno presentati nel report sono frutto di una *automazione* fornita mediante *codice Java*, pertanto sono facilmente ottenibili per un progetto qualsiasi.

Il codice sorgente delle applicazioni Java è presente nella repository: https://github.com/andreapaci

Entrambi le applicazioni sono quindi hostate sulla piattaforma di **GitHub** e sono connessi a **Travis CI** per il build automatico mediante **Ant** e **SonarCloud** per l'analisi del codice riguardante eventuali *code smells, vulnerabilità, eccessi di Cyclomatic Complexity*.

Nella repository sono anche presenti i risultati forniti in maniera *raw* (senza nessun tipo di formattazione/presentazione) liberamente consultabili nella cartella `output` e dei file di *logging* che permettono di analizzare l'ultima esecuzione dell'applicativo (`log.txt`)



*Esempio di una porzione del **LOG***

Tecnologie utlizzate

L'insieme delle teconologie/software utlizzati per lo sviluppo/mantenimento degli applicativi e l'analisi dei dati risultanti dall'esecuzione degli applicativi sono:

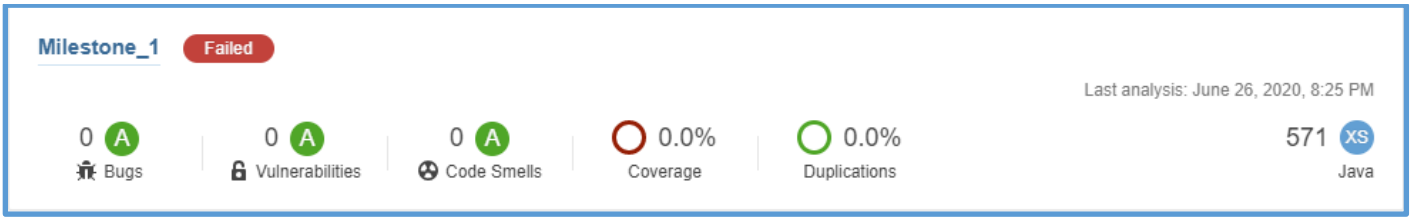
- **Java JDK 1.8** come enviroment di sviluppo
- **Eclipse IDE** come IDE
- **Github** e **TortoiseSVN** come sistema di **Versioning Control** e **Remote Repository**
- **Github** e **Jira** per il raccoglimento di informazioni
- Le librerie **JGit.jar, json.jar** e **weka.jar** rispettivamente per la connessione con Github, parsing di file JSON e per l'*evaluation* di modelli di **ML**
- **Ant** per il build in locale
- **TravisCI** per il building in remoto
- **Sonar Cloud** per l'identificazione di difetti relativi alla qualità software
- **Excel** per manipolazione ed estrapolazione dei dati

1. DELIVERABLE 1

1.1 Descrizione Iniziale

Il primo deliverable ha come scopo quello di **misurare la stabilità** di un **attributo** di **progetto**, in particolare il numero di **bug fixati per ogni mese** dalla data di creazione del progetto fino ad oggi. I progetti presi in analisi sono stati **PARQUET** e **PDFBOX**. Una trattazione riguardante la **stabilità** di un **attributo di progetto** è essenziale nel momento in cui si vuole **monitorare**, **controllare** e **predirre** il comportamento dello stesso durante il **processo di sviluppo**. Molteplici ricerche suggeriscono che un **progetto stabile** permette di migliorare notevoli aspetti dello sviluppo, e quindi fornire un **prodotto** di **qualità maggiore**. L'attività di misurare permette di **prendere decisioni** basate **su evidenze oggettive**, e quindi assicurare una *certa qualità* sia per quanto riguarda il progetto che per quanto riguarda il **processo** che ha come prodotto il progetto. Lo strumento principale utilizzato per misurare la **stabilità** dell'attributo *bug fix/mese* è stato lo **Statistical Control Process**, il quale mette in relazione il numero di bug fix per ogni mese con lo **scarto quadratico medio** (o *deviazione standard*) al fine di evidenziare quali valori ricadono fuori dai valori limite per la *stabilità* del *processo*.

Il link della dashboard su **Sonar cloud è**: https://sonarcloud.io/dashboard?id=andreapaci_Milestone_1



1.2 Analisi dell'applicativo

Il caloclo della metrica di *bugFix/mese* è stata effettuata usando un applicativo Java (SDK 1.8) che automatizza il processo dal *raccoglimento dei dati* dalle varie fonti interessate:

- Viene usato il sistema di **Ticket Tracking** offerto da **JIRA** per il retrive dei **ticket** relativi a **bug fixati/risolti** e le varie **release** con le **rispettive date**. Per fare ciò è stata fatta una interrogazione ai server di JIRA che ritorna un file `.json` dal quale si può fare parsing per ottenere le informazioni volute
- Viene invece usato il sistema di **Repository** offerto da **GitHub** per ottenere informazioni sulla data dell'effettivo fix di un determinato bug, cioè *l'ultimo commit* (temporalmente parlando) che *presenta uno specifico ticket*. Inoltre, come sarà più chiaro più avanti nel report, sono stati collezionati anche tutti i commit e raccolti in base a *mese/anno* del push anch'essi.

L'indirizzi usati per ottenere i file `.json` sono i seguenti:
*- Per il retrive dei **ticket:***
https://issues.apache.org/jira/rest/api/2/search?jql=project%20%3D%20**projName**%20AND%20issuetype%20%3D%20Bug%20AND%20status%20in%20(Resolved%2C%20Closed)%20AND%20resolution%20%3D%20Fixed%20ORDER%20BY%20updated%20DESC&fields=key
*- Per il retrive delle **versioni/release:***
https://issues.apache.org/jira/rest/api/2/project/**projName**

Una volta ottenuti i **ticket** e le varie **versioni** da **JIRA**, questi vengono salvati su due *array* distinti. Successivamente. si utlizzano le informazioni sui ticket precedentemente ottenute per recuperare la *data dell'ultimo* commit relativo ad un determinato ticket utlizzando il sistema di **repository** offerto da Github. Il fatto di incrociare le informazioni presenti sulle due piattaforme, invece che di usare le informazioni presenti solo su JIRA, proviene dalla consapevolezza che la data del *bug fix* riportata come campo per ogni ticket potrebbe non essere coerente con l'effettiva data in cui il bug in questione è stato risolto, la quale corrisponderà più precisamente alla data in cui è stato effettuato l'ultimo commit relativo al ticket preso in esame.

Una volta raccolti e analizzati, i dati ottenuti vengono *formattati* e *salvati* all'interno di file **`.csv`**, uno relativo ai bug fix per mese insieme ad *ulteriori statistiche* che ci permettono di estrapolare informazioni statisticamente significative sui risultati ottenuti, e un altro relativo alle versioni del progetto preso in analisi. Il **`.csv`** relativo ai bug fix è stato construito in modo da velocizzare la creazione del grafico che mostra l'andamento e la stabilità del progetto.

Project name	PDFBOX				
GitHub repository link	https://github.com/apache/pdfbox/				
Non-linked tickets	424				
Non-linked tickets Percentage	19,602404				
MONTH-YEAR	NUMBER BUG F	TOTAL COMMITS	MEAN	MEAN+3STD	MEAN-3STD
ago-08	2	21	12,07639	38,70002	0
set-08	0	11	12,07639	38,70002	0
ott-08	1	4	12,07639	38,70002	0
nov-08	4	14	12,07639	38,70002	0
dic-08	2	7	12,07639	38,70002	0
gen-09	6	16	12,07639	38,70002	0
feb-09	11	23	12,07639	38,70002	0
mar-09	6	14	12,07639	38,70002	0
apr-09	17	37	12,07639	38,70002	0

*Esempio del file **.csv** relativo ai bug fix estrapolato usando **PDFBOX.***

Index	Version ID	Version Name	Date
1	12313046	0.7.0	22/01/2005
2	12313045	0.7.1	10/04/2005
3	12313044	0.7.2	11/09/2005
4	12313043	0.7.3	12/10/2006
5	12313346	0.8.0	23/09/2009
6	12314330	1.0.0	16/02/2010
7	12314761	1.1.0	30/03/2010
8	12314868	1.2.0	29/06/2010
9	12315183	1.2.1	11/07/2010
10	12315175	1.3.1	26/10/2010
11	12315444	1.4.0	20/12/2010
12	12316012	1.5.0	03/03/2011
13	12316242	1.6.0	02/07/2011
14	12316940	1.7.0	29/05/2012

*Esempio del file **.csv** relativo alle versioni estrapolato usando **PDFBOX.***

Le *statistiche* calcolate mediante l'applicativo sono:

- **Media** dei bug fix
- **Deviazione standard** usata per il computo del **lower/upper limit**
- **Percentuale** e **conteggio** di **Ticket** non linkati a nessun commit su Github

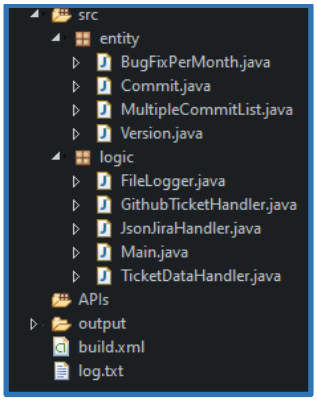
E' stata posta particolare attenzione sulla **validità** dell'output piuttosto che sull'ottimizzazione, questo poiché l'applicativo sviluppato è pensato per essere eseguito *una tantum* e *non iterativamente*, si trarrebbero pochi benifici da un'eventuale ottimizzazione. A questo proposito è stata considerato di implementare molteplici **check** sulla validità dei dati in fase di *runtime* al fine di garantire che i dati acquisiti siano *corretti* e *consistenti* fornendo una aggiuntiva sicurezza sulla *validità* dei dati. Tutto questo ovviamente comporta un **downgrade** minimo alle *performance* dell'applicativo, ma come trattato poco sopra, non è un problema in questo specifico contesto.

*Un esempio di **check aggiuntivo** che è stato implementato nella funzione* `TicketDataHandler.retreiveBugFixPerMonth` *è quello di contare che il numero di **commit** relativo a tutti i ticket sia uguale alla **somma** dei **bug fixati** per **tutti i mesi**. Ovviamente un controllo di questo tipo rallenta l'esecuzione dell'eseguibile (anche se di poco), ma ci da una ulteriore garanzia che i dati raccolti siano consistenti*

Non sono state effettuate particolari operazioni di **ottimizzazione** del codice poiché il tempo di esecuzione varia dai 30 secondi fino ad 1 minuto massimo, di cui la maggior parte del tempo viene speso per il *clone* della *repository* del progetto (step necessario se si vuole accedere al log delle commit), che viene prontamente eliminata alla fine dell'operazione per evitare di memorizzare dati non più neccessari e lasciare il *file system* nello stato in cui si trovava prima dell'esecuzione del dell'applicativo.

L'interfaccia con **GitHub** è stata creata usando l'**API JGit** insieme ad altre librerie di supporto, tutte **embedded** nel sorgente per garantire massima portabilità ed evitare di dover configurare l'ambiente di esecuzione se l'applicativo eseguito su una macchina diversa da quella di sviluppo.

Il codice è stato creato pensando ad **un'alta modularità** al fine di poter **riutilizzare** le componenti, evitare **design** considerati **problematici** (es. Metodi estremamente grandi, Classi *tuttofare*, ecc.), fornire un buon livello di **leggibilità** del codice (da che ne consegue un mantenimento/aggiornamento del codice più semplice) e un **organizzazione** basata su **funzionalità/contesto di operazione** delle classi.



Gerachia delle classi.

Per fornire dei dati sull'esecuzione del nostro *parser* è stato implementato un **logger** come **Singleton** (**GOF**)**,** il quale fa distinzioni su livello di **gravità** del messaggio, che scrive su di un file `log.txt` le operazioni eseguite e le informazioni di contesto durante tutto il tempo di runtime.

Le scelte che sono state fatte relativamente al **raccoglimento, interpretazione** e **esportazione** dei dati sono le seguenti:

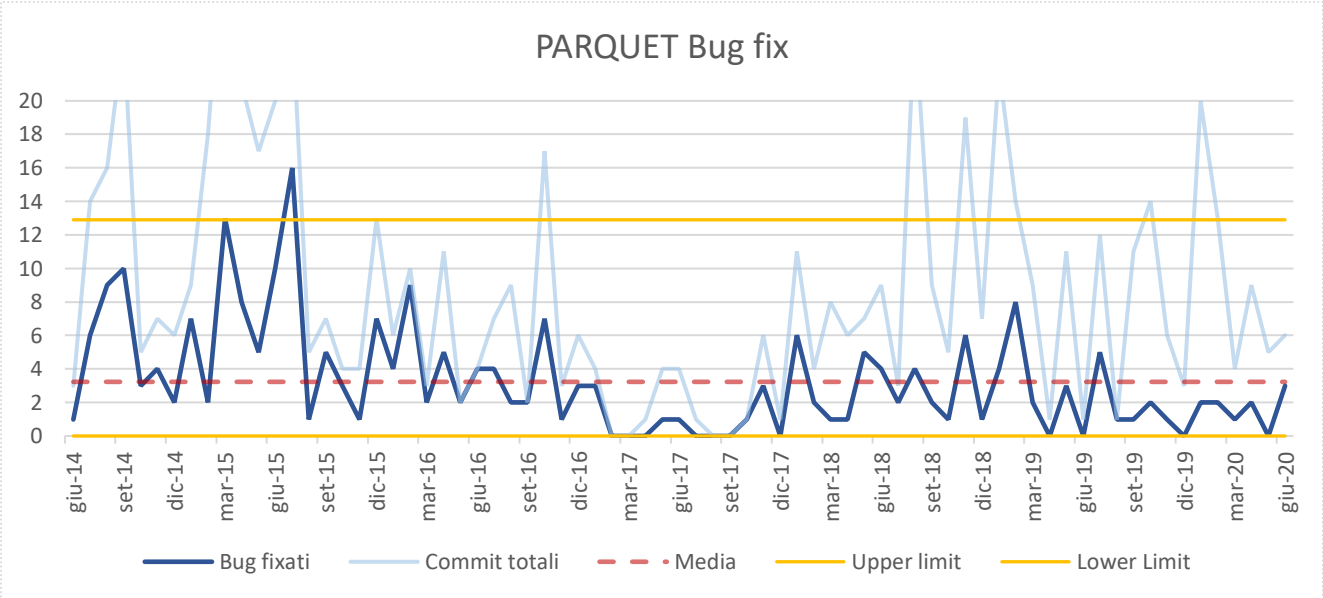
- La data del **fix** di ogni **bug ticket** su JIRA è stata assunta *meno affidabile* di quella su **GitHub**, quindi da ignorare.
- Nel caso vi sia presente un ticket su JIRA ma non su GitHub, il ticket viene **scartato**.
- Tutte le operazioni di analisi devono essere fatte direttamente a **runtime** invece che usando strumenti come fogli di calcolo (vedi *media, deviazione std, ecc.*)
- Se il **lower limit** è sotto lo **zero**, allora viene automaticamente impostato a 0
- Se uno dei punti del grafico non è nell'intervallo [lowerLimit, upperLimit] il progetto (o meglio, il processo) è considerato **instabile**.

1.3 Analisi dei dati e del grafico

Come già accennato nel paragrafo precedente, oltre al **conteggio** dei bug fixati per ogni mese, vengono anche calcolati **media** e **lower/upper limit**. Il lower/upper limit sono calcolati come segue:

```
lower/upper limit = mean -/+ 3 * standard_deviation
```

I lower/upper limit servono per **classificare** la **stabilità** un **processo di sviluppo**, infatti l'eventualità di valori che ricadono fuori dall'intervallo `[lower limit, upper limit]` sono simbolo di **instabilità** del processo in un intorno temporale del punto. Per quanto riguarda il progetto **PARQUET** abbiamo il seguente andamento:



Il grafico dell'andamento di PARQUET. Il lower limit è stato portato da un valore negativo a 0 poiché non ha senso parlare di valori negativi quando si parla di conteggi

E' evidente come con il tempo il processo di sviluppo di **PARQUET** si sia stabilizzato per quanto riguarda il numero di bug fix, infatti dall'inizio del 2018, i valori si attestano tutti vicino alla media (linea tratteggiata in rosso) senza grosse variazioni, tutti abbondandemente dentro i limiti superiori/inferiori, il che sta a significare una maggiore *maturità* del progetto e dell'organizzazione.
Durante le **fasi iniziali** dello sviluppo e facile vedere come il processo era più **immaturo** e **instabile**, comportando oscillazioni più marcate. Inoltre, a *luglio 2015* viene registrato anche un valore fuori dall'*upper bound* (linea in giallo), un ulteriore sintomo di *instabilità*. Questo picco va però constestualizzato: è evidente che in quel determinato periodo c'è stato un incremento della produttività significativo, vegnono infatti registrate molte più commit (linea azzurro chiaro).

Mese-Anno	*Bug Fix*	*Commit Totali*
dic-14	2	6
gen-15	7	9
feb-15	2	18
mar-15	13	31
apr-15	8	21
mag-15	5	17
giu-15	10	20
lug-15	16	25
ago-15	1	5
set-15	5	7
ott-15	3	4
nov-15	1	4

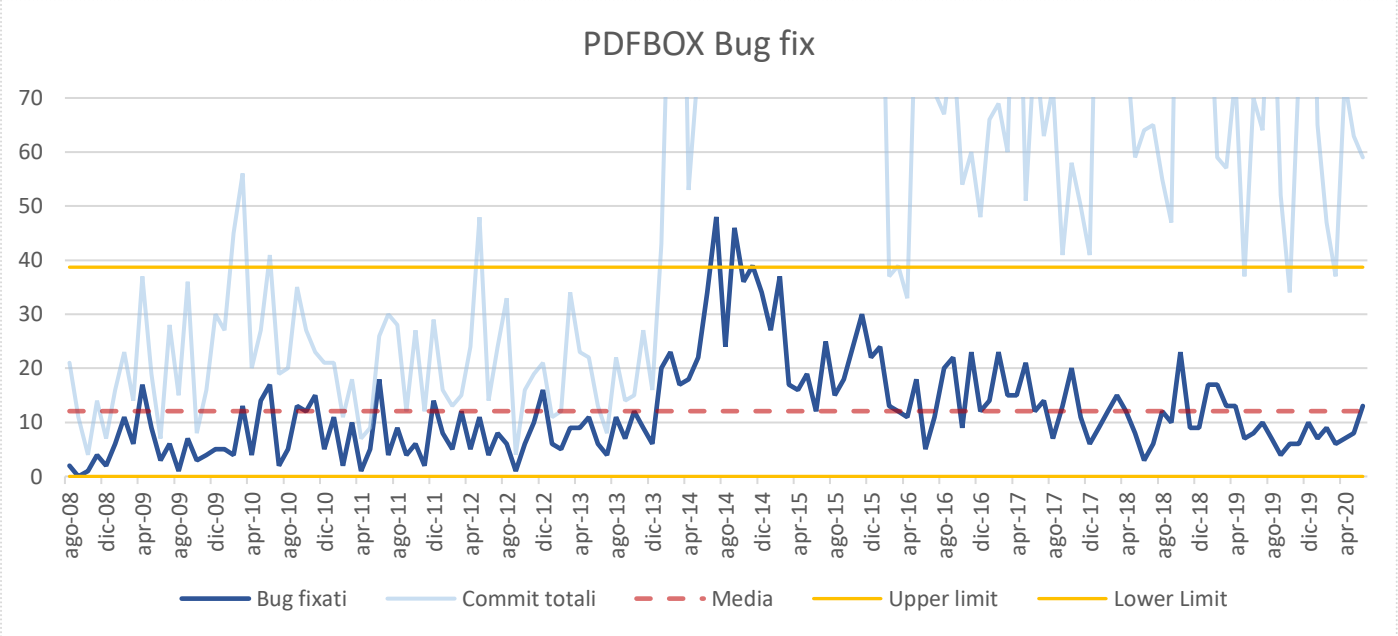
*Valori riportati dal **.csv** per mostrare numericamente il picco visualizzato sul grafico*

Inoltre, come è possibile vedere dalle informazioni sulle release raccolte, durante il 2015, a distanza di un anno dalla prima versione rilasciata a maggio 2014, sono state rilasciate numerevoli releases in un periodo relativamente breve, ed è quindi ragionevole pensare che in un periodo di **forte produttività** siano stati risolti più bug, mettendolo anche in relazione al fatto che con ogni probabilità il numero alto di release in quel periodo corrisponde anche ad un **incremento** di **nuove feature** con i relativi bug e bugfix.

Versione	*Data*
1	21/05/2014
2	15/04/2015
3	18/05/2015
4	11/07/2015
5	21/07/2015
6	17/12/2015
7	23/10/2016

Con una **percentuale di ticket non linkata** su Github del 57,42% (317 tickets non linkati), una percentuale molto alta, è come se ci fossimo persi *più della metà* dei dati. Questo dato può anche essere interpretato come la conferma della discrepanza tra le informazioni presenti su JIRA e Github

Per quanto invece riguarda il progetto **PDFBOX** abbiamo il seguente andamento:



Con una **percentuale di ticket non linkata** su Github del 19,62% (421 tickets non linkati), una percentuale decisamente più bassa e accettabile di PARQUET, anche se corrisponde a *circa 1/5* dei *dati*.

Il progetto è stabile per la maggior parte del tempo, ma tra il 2014 e il 2016 ha un picco di bug fix (di cui alcuni punti cadono sopra il limite superiore), sempre sintomo di instabilità, ma proprio come per PARQUET, si può fare un discorso analogo, infatti sempre nello stesso periodo dove c'è un incremento di bug fix, c'è anche un incremento significativo di commit in generale (registrando un picco di 400 commit in un mese), e quindi più in generale un incremento di produttività. Anche per PDFBOX l'incremento di bug fix corrisponde ad un incremento di versioni rilasciate nell'unità di tempo, infatti come è possibile vedere nella tabella sottostante, la densità di versioni rilasciate in quel periodo è più alta

Versioni	*Data*
15	25/07/2012
16	23/03/2013
17	10/04/2013
18	02/06/2013
19	29/11/2013
20	31/01/2014
21	02/05/2014
22	22/06/2014
23	19/09/2014
24	13/12/2014
25	28/03/2015
26	22/07/2015

Versioni	*Data*
27	18/01/2016
28	18/03/2016
29	26/04/2016
30	09/06/2016
31	17/09/2016
32	09/12/2016
33	16/12/2016
34	17/03/2017
35	15/05/2017
36	20/07/2017
37	02/11/2017
38	27/02/2018

Successivamente al periodo dove viene presentato il picco, è evidente un andamento più costante che si attesta vicino alla media e con un numero di bug fix molto stabile, indicando un approccio alla bugginess più maturo e continuativo, e quindi di un prodotto software supportato in maniera appropriata.

2. DELIVERABLE 2

2.1 Descrizione Iniziale

Il secondo Delivarable ha come scopo quello di **eseguire uno studio empirico,** basato su modelli di **Machine Learning,** al fine di identificare le **Classi defective** di un progetto, con tecniche di **Sampling** e **feature selection**,offerte dall'API di **Weka**, basandosi sulle metriche dei file che compongono il progetto stesso, per poi estrapolare quale compinazione di **classificatore/tecnica di sampling/tecnica di feature selection** produce risultati migliori. Questo studio proviene dalla necessità di volere catalogare i file di progetto come *potenziali defective*, basandosi sulle proprietà del file stesso e alcune altre caratteristiche. Le motivazioni per cui si vuole avere la possibilità di marcare i file come *potenzialmente defective* sono molteplici, un esempio è capire quali sono i file che devono essere soggetti a **test più intensivi** rispetto ad altri, rendendo quindi possibile fare una scelta *motivata da dati e evidenze oggettive* su quali componenti **fare testing esaustivi**. I progetti presi in analisi sono **BOOKKEEPER** e **SYNCOPE**.

Il deliverable si divide in due fasi:

- **Acquisizione** delle metriche e della **defectiveness** dei file `.java` del progetto mediante **JIRA** e **GitHub**
- **Analisi** dei dati ottenuti successivamente all'elaborazione del dataset, ottenuto nella prima fase, con tecniche di **Machine Learning**

La **prima fase** consiste nell'acquisire la conoscenza se una determinata classe è **Defective** su una determinata **Release,** insieme all'acquisizione di altre **metriche**, la cui scelta è stata effettuata prendendo in considerazione quali caratterstiche possono essere più significative per la *defectiveness* di un file (es. una metrica come la **size** di un file è più significativa per la bugginess piuttosto che il **numero di commenti**).
Per l'obiettivo posto in questo studio, ha senso considerare solo ed esclusivamente **files** contenenti **codice java**, scartando quindi i file restanti. Inoltre, vengono **scartate** il **50% delle release**: il motivo è che, prima di poter definire un file buggy, è necessario accoregersi della presenza di un bug, ma spesso il tempo che intercorre tra l'introduzione di un bug e la sua rilevazione è non trascurabile (**Class Snoring**), portando così all'incertezza nel definizione di un file defective o meno nelle release più recenti. Questo non vuol dire che se si analizzano release passate si ha la certezza che un file sia defective o meno, ma piuttosto è meno probabile che il bug non sia stato scoperto, infatti è stato visto che il **Missing rate** diventa del *10%* quando si vanno ad eliminare il *50% delle releases*, portando a risultati decisamente più accurati.

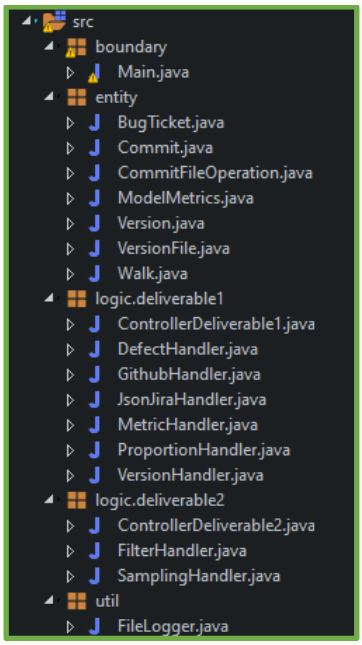
L'output della prima fase contiene, per ogni file di ogni release, l'insieme delle metriche calcolate e la sua eventuale *defectiveness*. Questo output viene poi usato come input della seconda fase.

L'analisi di dati ottenuti mediante modelli **statistici**, o, come nel nostro caso, mediante modelli di **ML**, permette di effettuare operazioni di **predizione** di dati futuri di dati non noti. La **seconda fase** utlizza le informazioni ottenute nella prima per fare valutazioni su quale modello di Machine Learning produca risultati **più accurati** con i **classificatori scelti** nel marcare un file come buggy o meno. In altre parole, si cerca di capire quale modello effettua una **predizione più accurata** sulla presenza di bug in un file.
La **bugginess** è una di quelle caratteristiche che non è sempre possibile definire per un file, specialmente in mancanza di dati essenziali (come per esempio **IV, OV** e **FV**) o nell'eventualità in cui la si vuole definire per una classe in un periodo relativamente "recente". Esistono molteplici metodi per lo studio della bugginess, ma non è sempre possibile stabilire con assoluta certezza se si ha a che fare con una classe defective o meno (prendendo ad esempio il metodo **SZZ**, il quale fa uso del **blame**, oltre ad essere molto complesso da implementare, è soggetto a limitazioni importanti come una **accuratezza non perfetta**, non rileva bug di tipo **regressivo** e bug dove il fix è stato ottenuto con **l'aggiunta di codice**), ecco perché l'impiego di modelli di ML permette di avere una maggiore *confidenza* sull'attribuzione della qualità defective.

Il link della dashboard su **Sonar cloud è**: https://sonarcloud.io/dashboard?id=andreapaci_deliverable_2

2.2 Analisi del sorgente

Anche se il codice sorgente del secondo deliverable, la sua analisi, proprio come per il paragrafo precedente, viene divisa tra le due parti principali che compongono l'applicativo. I file dell'applicativo sono stati **organizzati** in **packages** a seconda dello scopo e se sono relativi al primo o al secondo deliverable. Il codice è stato creato pensando ad **un'alta modularità** al fine di poter **riutilizzare** le componenti, infatti alcune classi sono state riprese dalla prima deliverable.



Struttura del sorgente dell'applicativo della seconda deliverable suddivisa in packages

2.2.1 Analisi del sorgente – Calcolo metriche e defectiveness

La prima parte del Deliverable consiste nel ottenere le **metriche** e la **bugginess** di una **classe** in una determinata **release**, *iterato per tutti i file presenti per tutte le release*. Le **release** sono state raccolte usando **JIRA**, più in particolare, è stata fatta un interrogazione usando il seguente **URL**:

`https://issues.apache.org/jira/rest/api/2/project/projName`

L'interrogazione restituisce un file **json** che viene scansionato per ottenere le informazioni volute e salvarle su istanze della classe `Version.java`.
Una volta raccolte, le release vengono ordinate cronologicamente per essere poi successivamente usate per raccogliere le informazioni relative ai **file** e alle **metriche** presenti in ognuna di esse. Inoltre viene generato un file **.csv** situato nella cartella "**output/deliverable_1**" contenente la lista delle versioni con il relativo *nome, id e data di publicazione*.

Le **metriche** sono state computate usando di *log* delle commit fornite da **Github** (interfacciata sempre usando la API di **JGit**), quindi anche qui, per accedere a queste informazioni, è necessario fare prima il *clone* della repository e per poi estrapolare le metriche scorrendo ogni commit. Le **commit** utilizzate sono solamente quelle del **master branch**, mentre quelle relative a cicli paralleli di sviluppo e successivamente **merged** nel master, sono state scartate. Anche se vengono scartate, non vi è effettiva perdita di informazione: quando i due flussi di lavoro vengono "merged" nel master branch, vengono prese in considerazione tutte le operazioni fatte nel flusso parallelo, ma con l'unica differenza che vengono calcolate tutte insieme e non commit per commit. Inoltre sono stati rilevati *numerosi* **branch orfani**, che causano forti discrepanze tra le metriche attuali e quelle calcolate.
Nonostante è necessario solo calcolare le metriche per i file appartenenti alle sole release della prima metà, l'applicativo sviluppato le calcola per tutte le release, insieme alla buggyness ove possibile, per poi scartarle durante la fase di esportazione dell'output. La scelta di calcolarle per tutte le release invece che solo per la metà è per collegare lo studio mostrato in questo report con quello effettuato riguardante i **test**: le classi da testare sono infatti scelte sulla base delle

metriche calcolate e sulla base di quante volte la classe in questione è stata identificata come *defective*.
Prima di procedere con il calcolo delle metriche bisogna avere la **lista dei file** su cui calcolarle. Per fare ciò, si prendono, per ogni versione esistente, l'*ultima commit publicata prima del rilascio della verisione* e si ottiene il **file tree**, che ritorna *tutti i file presenti* in quel momento dello sviluppo del progetto. Come verrà spiegato più avanti nel report, la costruzione della lista contenete tutti i file viene effettuata in due modi diversi.
Le **metriche** considerate sono le seguenti:

- **Size(SLOC):** numero di linee di codice di un file, scartando commenti e righe vuote.
- **Age:** "età" del file espressa in *settimane*
- **Number of Revision:** numero di commit che hanno interessato il file in questione
- **Number of Authors:** numero di autori che hanno effettuato operazioni sul file in esame
- **LOC Touched:** somma di linee di codice *aggiunte, eliminate e modificate*
- **LOC Added:** numero di linee di codice *aggiunte*.
- **Max LOC Added:** il numero *massimo* di linee di codice *aggiunte* in una revisione
- **Average LOC Added:** numero *medio* di linee di codice *aggiunte*
- **Change Set Size:** somma del numero di file *committed* insieme al file in esame
- **Max Change Set Size:** numero *massimo* di file *committed* insieme al file in esame in una revisione
- **Average Change Set Size:** numero *medio* di file *committed* insieme al file in esame
- **Churn:** somma di linee di codice *aggiute* meno il numero linee di codice *eliminate*
- **Max Churn:** il massimo tra le revisioni di linee di codice *aggiute* meno il numero linee di codice *eliminate*
- **Average Churn:** la media tra le revisioni di linee di codice *aggiute* meno il numero linee di codice *eliminate*

Per un totale di **14 metriche**. Le metriche sono state scelte considerando quali sono i fattori che più possono influenzare la bugginess di una classe. Le metriche vengono ricalcolate *release per release*, infatti i valori di **number of revision** e **number of authors** non vengono incrementati tra una release e un'altra, bensì vengono riazzerate e calcolate solo per l'intervallo tra la release presa in esame e quella subito precedente.
Le metriche vengono calcolate mettendo insieme le varie operazioni fatte con le **commit** tra una release e un'altra, sfruttando il meccanismo di **git diff** per estrapolare le operazioni fatte in una determinata commit. Nel caso di operazioni di **RENAME**, il "nuovo file" non viene considerato come nuovo, ma le metriche calcolate fino a quel momento vengono trasferite sotto il nuovo nome del file. Nel caso di una operazione di **COPY** invece, le metriche vengono azzerate per il nuovo file e mantenute per il vecchio (o *file sorgente*), questo perché, a tutti gli effetti, si parla di un file nuovo

La **Bugginess** viene calcolata mediante l'impiego di ticket su **JIRA** ottenuti con un interrogazione usando il seguente URL:

`https://issues.apache.org/jira/rest/api/2/search?jql=project%20%3D%20projName%20AND%20issuetype%20%3D%20Bug%20AND%20status%20in%20(Resolved%2C%20Closed)%20AND%20resolution%20%3D%20Fixed%20ORDER%20BY%20updated%20DESC&fields=key`

Per ogni ticket presente, viene prelevato **l'ID del ticket**, **affected versions** e **data di creazione**. **L'ID del ticket** viene usato per trovare le commit che sono state effettuate per fixare il bug citato nel ticket. Queste commit sono di interesse poiché i file che vengono **modificati** o **eliminati** per fixare un bug sono **defective**. L'**affected version** è l'intervallo di versioni **[Injected Version, Fixed Version)** e serve per trovare quali versioni sono affette dal bug in esame, ed è quindi necessario avere la conoscenza di questo dato per sapere quando il bug è stato introdotto, e quindi da quale release il file va marcato come defective. La **data di creazione** del ticket viene usata per *computare* l'**Opening Version**, valore necessario per l'applicazione del **Proportion**.
L'unico dato mancate per computare l'intervallo di influenza di un determinato bug è la **fixed version**, la quale viene calcolata andando a prendere *la prima versione subito dopo l'ultima commit relativa al ticket*. Questo è necessario poiché la fixed version riportata come campo su JIRA non è altro che una *predizione* su quando si pensa il bug venga risolto, e non l'effettiva versione di risoluzione.

Quando non presente l'**affected version** nei ticket raccolti su JIRA viene usato il **Proportion**, gestito da un **Handler**, chiamato `ProportionHandler`, che si occupa di effettuare le varie operazioni di calcolo oltre che di tenere traccia delle

informazioni relative al calcolo ed all'applicazione del proportion. Sono stati implementati sia il **Proportion Cold Start** e sia il **Proportion Increment**. Nelle fasi iniziali non si hanno sufficienti dati per poter utilizzare il **Proportion Increment**, perciò viene applicato il **Cold Start** usando i ticket e la repository **13 altri progetti,** calcolando per ogni bug *closed and fixed* la proporzione **P** con la seguente formula:

$$P = (FV - IV) / (FV - OV)$$

Per poi prendere la **mediana** tra tutti le proporzioni dei bug dei progetti analizzati. Essendo il computo della proporzione con Cold Start molto *time consuming* (è necessario clonare tutte le repository di ogni progetto), il valore viene calcolato solamente se l'applicativo non rileva un file denominato `coldStart_p.txt` che contiene la mediana delle proporzioni calcolate in qualche run precedente. Nel momento in cui l'applicativo non rileva questo file, procede con il computo del Cold Start per poi salvare il valore su di un file per essere usato nel run successive.

Quando il numero di **ticket** analizzato è *sufficientemente* rappresentativo del ciclo di vita dei bug nel progetto considerato, allora vi è il passaggio da **Cold Start** a **Increment**. Per utlizzare **Increment** è necessario avere la **media** delle **proporzioni** dei **ticket precedenti cronologicamente** (motivo per cui, i ticket, prima di essere utilizzati, vengono *ordinati* in *base alla data di creazione* mediante l'impiego delle funzioni *lambda* fornite da Java 8), ed è quindi necessario tenere traccia di tutte le proporzioni calcolate, ovviamente usando solo ed esclusivamente **ticket conformi** (`IV <= OV < FV`). Estrapolao il valore di **proportion**, è possibile calcolare l'**IV predetto** usando la formula:

$$IV = FV - (FV - OV) * P$$

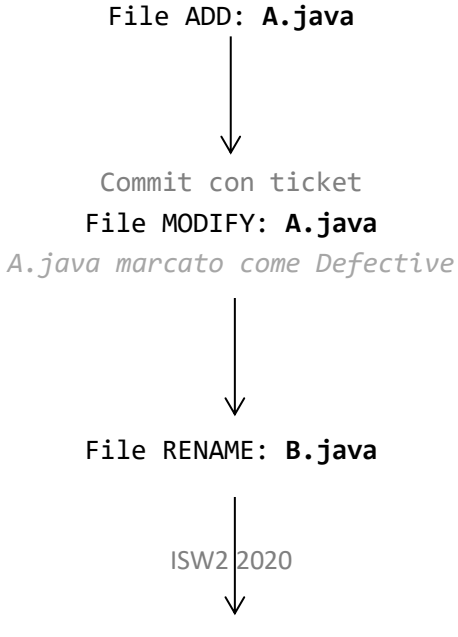
Nel caso di **ticket non conformi**, si evita di *computare la proporzione* per usarla nel **proportion incremental**, però si usano comunque i dati a disposizione per calcolare l'intervallo `[IV, FV)`. Un esempio potrebbe essere il caso:

$$OV < IV < FV$$

In questo particolare esempio è chiaro che computare e usare P per i ticket successivi è un'idea da scartare, però è comunque possibile usare l'intervallo `[IV, FV)` per marcare eventualmente i file toccati come *defective*. Un altro caso contemplato di ticket non conforme è:

$$FV <= IV$$

In una casistica di questo tipo, oltre a non avere senso computare e usare P, non si può nemmeno trovare l'intervallo di **affected version** necessario per trovare la buggyness di un file. A tal proposito, si assume FV come valore *sempre attendibile* (ricordiamo che FV proviene da un calcolo che stabilisce la versione come quella subito dopo l'ultima commit del ticket) e l'*injected version* viene invece *calcolata* usando il proportion (che può essere sia *Incremental* che *Cold Start*). Un *ulteriore caso* è stato considerato riguardante il calcolo della *bugginess*: bisogna tenere traccia di rinominazioni o copie del file definito *defective*. Una situazione illustrativa è la seguente:



Release publicata

File presenti: **B.java, Others.java,…**

Nonostante il file abbia cambiato nome, bisogna comunque tenere traccia che si tratta di un file *defective*. Per ottenere ciò, ogni volta che si rileva un operazione di **DELETE** o **MODIFY** su commit che presentano un ticket (i file toccati da operazioni *delete* e *modify*, come già accennato poco sopra, sono identificati come defective per ovvie ragioni), prima di marcare i file presenti nella versione di interesse come defective, si cercano tutti gli **alias**, cioè tutti quei file che, di conseguenza, sono defective. La **costruzione della lista di alias** avviene scorrendo la lista di commit (precedentemente ordinate cronologicamente con un *bubble sort*) *nell'intervallo [IV, FV)*, e aggiungendo a tale lista i file che sono defective di conseguenza ogni qual volta vengono trovati. I file che sono *defective di conseguenza* vanno ricercati nei RENAME (il rename non è solo il caso di una rinominazione vera e propria, ma anche il caso in cui il file viene spostato) e nei COPY (se un file viene prima modifcato e poi copiato, entrambi i file vanno considerati defective).

Proprio come per il primo deliverable, è stata posta più **enfasi** sulla **validità** dei dati piuttosto che **dell'ottimizzazione** (anche qui, il tempo di esecuzione si aggira intorno ai 2 minuti, il quale è un tempo ragionevole considerando che va eseguito una sola volta per raccogliere i dati, e che la maggior parte del tempo è speso per fare il *clone*, il quale non può essere ottimizzato), vengono quindi effettuati numerosi controlli per evitare dati **falsati**, che andrebbero poi a compromettere i risultati ottenuti applicando i modelli di **Machine Learning**.

*Un esempio di **check aggiuntivo** che è stato implementato è che i file **presenti in una versione** (cioè l'ultima commit prima della release) sono gli stessi che vengono trovati se si segue il flusso di **git diff** tra una commit e un'altra partendo dalla commit iniziale fino a l'ultima prima della versione, o anche se tutti i file rilevati in una versione sono univoci e non ci sono duplicati*

Come già detto per la prima deliverable, tutte le librerie necessarie sono state **embedded** nel sorgente per garantire massima **portabilità** e l'implentazione di un **logger**, sempre implementato come **Singleton**, per tenere traccia delle operazioni effettuate.

Come per la prima deliverable, l'**output** principale è un **file .csv** contenente per ogni **file** di ogni **versione** le **metriche** e la **bugginess**, espressa come valore **booleano**

Version	FileName	Size	Age	Number_revision	Number_Authors	Loc_Touched	Loc_Added	Max_Loc_Added	Avg_Loc_Added	Change_Set_size	Max_ChgSetSize	Avg_ChgSetSize	Churn	Max_Churn	Avg_Churn	Defective
1	bookkeeper-benchmark/src/m	121	35	4	2	333	235	137	58.75	559	281	139.75	137	137	34.25	false
1	bookkeeper-benchmark/src/m	215	35	5	2	436	340	252	68.0	560	281	112.0	244	252	48.8	false
1	hedwig-client/src/main/java/o	44	35	3	2	64	56	48	18.666.666	481	281	16.033.333	48	48	16.0	false
1	hedwig-client/src/main/java/o	58	35	2	2	71	67	63	33.5	452	281	226.0	63	63	31.5	false
1	hedwig-client/src/main/java/o	222	35	2	2	277	257	237	128.5	452	281	226.0	237	237	118.5	true
1	hedwig-client/src/main/java/o	113	35	3	2	145	139	133	46.333.332	481	281	16.033.333	133	133	44.333.332	true
1	hedwig-client/src/main/java/o	115	35	4	2	163	150	136	37.5	578	281	144.5	137	136	34.25	true
1	hedwig-client/src/main/java/o	153	35	3	2	185	181	176	60.333.332	549	281	183.0	177	176	59.0	true
1	hedwig-client/src/main/java/o	41	35	2	2	48	47	46	23.5	452	281	226.0	46	46	23.0	false
1	hedwig-client/src/main/java/o	101	35	5	3	146	137	127	27.4	579	281	115.8	128	127	25.6	false
1	hedwig-client/src/main/java/o	129	35	3	2	155	152	148	50.666.668	549	281	183.0	149	148	49.666.668	false
1	hedwig-client/src/main/java/o	53	35	2	2	60	59	58	29.5	452	281	226.0	58	58	29.0	false
1	hedwig-client/src/main/java/o	140	35	2	2	155	152	149	76.0	452	281	226.0	149	149	74.5	true
1	hedwig-client/src/main/java/o	64	35	2	2	82	78	74	39.0	452	281	226.0	74	74	37.0	false
1	hedwig-client/src/main/java/o	32	35	1	1	37	37	37	37.0	281	281	281.0	37	37	37.0	false
1	hedwig-client/src/main/java/o	33	35	2	2	42	40	38	20.0	452	281	226.0	38	38	19.0	false
1	hedwig-client/src/main/java/o	34	35	2	2	41	40	39	20.0	452	281	226.0	39	39	19.5	false
1	hedwig-client/src/main/java/o	82	35	4	2	118	107	95	26.75	578	281	144.5	96	95	24.0	false
1	hedwig-client/src/main/java/o	77	35	3	2	94	91	87	30.333.334	549	281	183.0	88	87	29.333.334	false
1	hedwig-client/src/main/java/o	64	35	4	2	85	78	70	19.5	578	281	144.5	71	70	17.75	false
1	hedwig-client/src/main/java/o	102	35	4	2	132	123	113	30.75	578	281	144.5	114	113	28.5	true
1	hedwig-client/src/main/java/o	317	35	6	3	492	414	329	69.0	580	281	96.666.664	336	329	56.0	true
1	hedwig-client/src/main/java/o	76	35	4	2	102	93	83	23.25	578	281	144.5	84	83	21.0	false
1	hedwig-client/src/main/java/o	49	35	3	2	68	63	58	21.0	481	281	16.033.333	58	58	19.333.334	false
1	hedwig-client/src/main/java/o	112	35	4	2	137	130	122	32.5	578	281	144.5	123	122	30.75	false

*Esempio del file **.csv** estrapolato usando **BOOKKEEPER.***

2.2.2 Analisi del sorgente – Predizione e applicazione di modelli di Machine Learning

La **seconda parte** dell'applicativo, sebbene sia più piccola di dimensioni e più semplice da un punto di vista algoritmico, presenta la **parte principale** della **seconda deliverable**. Il focus è quello di usare i dati raccolti nella prima parte e valutare quale *modello di Machine learning* produce **stime più accurate**, provando tra le possibili combinazioni di **classificatori, feature selection** e **tecninche di balancing**. In particolare, come **evaluation tecnique** è stata scelta la **walk forward** piuttosto che la ***k-fold* cross validation**. La scelta di usare il **walk forward** è da ricercare nel fatto che quest'ultima è la scelta migliore da applicare nel caso si voglia *preservare l'ordine dei dati* (**Time-series validation**), mentre la *cross-validation* non tiene in considerazione questo fattore. Per applicare il *walk forward* bisogna **dividere** il **dataset** in porzioni, e il criterio scelto è la **versione** (o **release**): le singole unità che compongono il walk forward (rappresentate dalla classe `walk.java`) contengono la porzione del dataset relative a tutti i file contenuti in una specifica versione. In altre parole, il *walk k-esimo* contiene i file, con relative metriche e defectiveness, appartenenti alla *versione k*. Come già accennato prima, lo scopo è quello di provare ogni possibile combinazione di:

```
{classifier, balancing tecnique, feature selection}
```

bisogna quindi fare il **prodotto cartesiano** tra questi insiemi

Classifier
Random Forest
Ibk
Naive Bayes

Feature selection
No selection
CFS con Best First
Wrapper con Best First

Balancing
No sampling
Oversampling
Undersampling
Smote

E' stato aggiunto anche il **wrapper con best first** come **feature selection** per poter fare confronti più significativi ed eventualmente trovare una combinazione dei 3 ancora migliore, portando ad un totale di:

```
3 x 3 x 4 = 36 possibili combinazioni
```

Per le varie possibilità di **Feature Selection** e **Balancing** è stato fatto un leggero *tuning* impostando i parametri fondamentali usando i valori considerati di *default*.

Questo processo utilizza quindi i dataset estrapolati dalla parte precedente, e quindi viene eseguito sia utlizzando BookKeeper sia Syncope. I **dataset** vengono quindi **caricati** e **convertiti** in modo che l'API di weka riesca ad operarci sopra e vengono anche esportati come file **.arff** nella cartella "**output/deliverable_2**". I dati, prima di essere processati, vanno attraverso una fase di **preprocessamento** dove viene impostato il **Class index** (l'attributo che su cui si intende effettuare le stime) e i dati vengono **normalizzati.** La normalizzazione è un passo essenziale per garantire che ogni attributo contribuisce con **lo stesso peso** nelle fasi di **training** del modello, evitando quindi di avere *bias* di questo tipo. Successivamente alla fase di preprocessamento, vengono **defini** i **walk** e memorizzati in strutture dati pronti per essere utlizzati, decretando la fine della fase di inizializzazione.

Iterativemente per ogni walk, vengono **scelte** ed **eseguite** le tutte varie **combinazioni** di **modello** di **ML**, applicando gli eventuali **filtri** a *classificatori/attributi*, per poi fare il **training** e l'**evaluation** sulla porzione di testing del walk. I risultati ottenuti vengono memorizzati in una struttura dati che viene poi esportata su di un file **.csv** pronto per essere analizzato. Alcune **evaluations** sono state scartate poiche presentavano valori *non rappresentabili* (es. **NaN**): questo è il caso di metriche che provengono dal rapporto di due valori, dove il valore al denominatore è 0, risultando in un'operazione non effettuabile. La scelta di scartarli proviene dal fatto che non era possibile assegnare un valore *dummy* che non comprometteva la validità del dataset, e si è quindi pensato che la scelta migliore fosse scartarli.

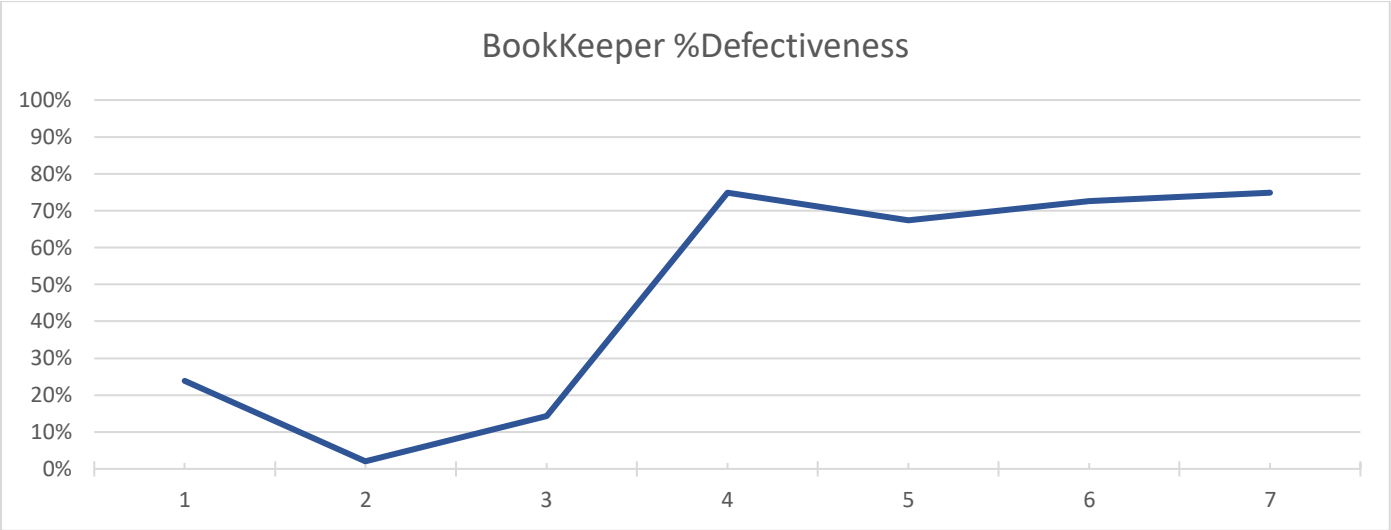
Nel calcolo delle metriche è stato assunto il valore di **defective** come il valore **positivo**, cioè con un **indice di classe** pari a **1**.

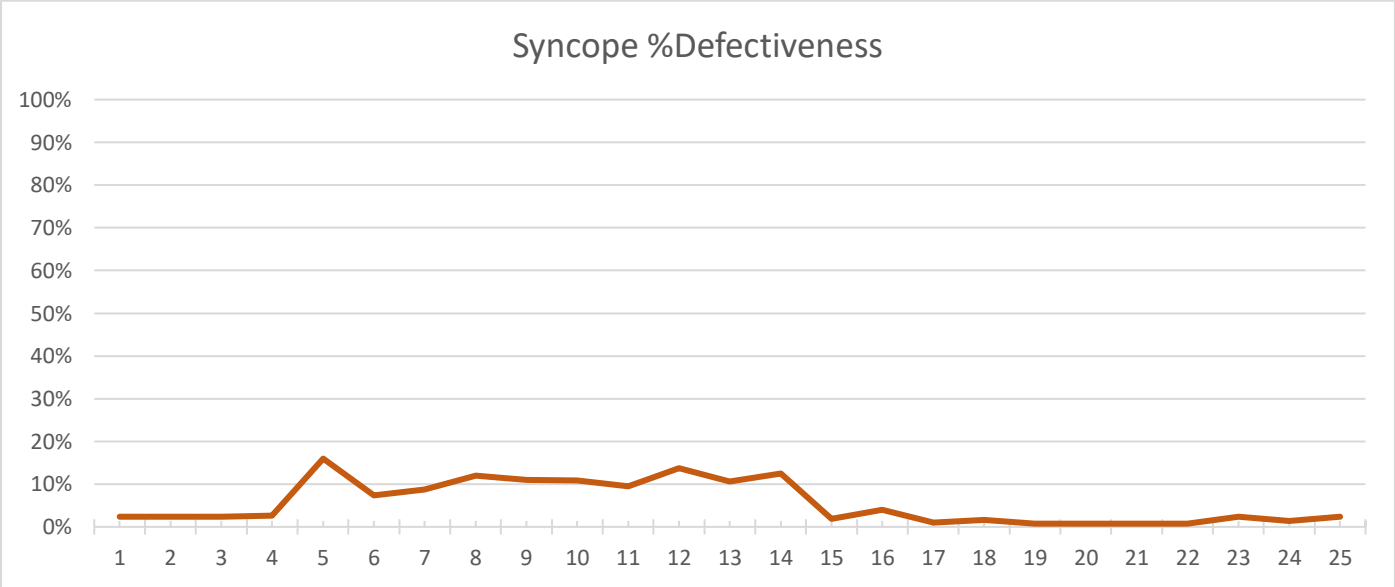
Dataset	Training_Release	%Training	%Defective_training	%Defective_testing	Classifier	Balancing	Feature_Selection	TP	FP	TN	FN	Precision	Recall	ROC_Area	Kappa
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	Random Forest	No sampling	No selection	26	18	211	46	0.5909090909090909	0.3611111111111111	0.7981865599223678	0.3259622113365991
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	Random Forest	Oversampling	No selection	35	25	204	37	0.5833333333333334	0.4861111111111111	0.7589459000485201	0.39978129422359443
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	Random Forest	Undersampling	No selection	29	22	207	43	0.5686274509803921	0.4027777777777778	0.7756853469189714	0.3407796758650899
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	Random Forest	SMOTE	No selection	29	18	211	43	0.6170212765957447	0.4027777777777778	0.7927886948083455	0.3679735637327459
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	Random Forest	No sampling	CFS con Best First	31	22	207	41	0.5849056603773585	0.4305555555555556	0.8028869480834546	0.36775247557763474
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	Random Forest	Oversampling	CFS con Best First	26	22	207	46	0.5416666666666666	0.3611111111111111	0.7847525473071325	0.2992330868252724
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	Random Forest	Undersampling	CFS con Best First	28	23	206	44	0.5490196078431373	0.3888888888888889	0.7923641436196022	0.3204959735840156
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	Random Forest	SMOTE	CFS con Best First	31	22	207	41	0.5849056603773585	0.4305555555555556	0.7922125181950509	0.36775247557763474
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	Random Forest	No sampling	Wrapper con Best First	35	29	200	37	0.546875	0.4861111111111111	0.7614628820960698	0.3737074401008827
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	Random Forest	Oversampling	Wrapper con Best First	48	71	158	24	0.40336134453781514	0.6666666666666666	0.7298944687045124	0.2914137033824806
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	Random Forest	Undersampling	Wrapper con Best First	25	26	203	47	0.49019607843137253	0.3472222222222222	0.7588245997088792	0.25964486674079307
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	Random Forest	SMOTE	Wrapper con Best First	31	29	200	41	0.5166666666666667	0.4305555555555556	0.7844492964580301	0.3223337192847035
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	IBk	No sampling	No selection	14	11	218	58	0.56	0.19444444444444445	0.5732047549733139	0.1886158534203229
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	IBk	Oversampling	No selection	40	30	199	32	0.5714285714285714	0.5555555555555556	0.7209485686559922	0.42863266180882975
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	IBk	Undersampling	No selection	40	30	199	32	0.5714285714285714	0.5555555555555556	0.7188864628820961	0.42863266180882975
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	IBk	SMOTE	No selection	40	30	199	32	0.5714285714285714	0.5555555555555556	0.7113658418243571	0.42863266180882975
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	IBk	No sampling	CFS con Best First	34	50	179	38	0.40476190476190477	0.4722222222222222	0.6292455118874333	0.24016064257028094
BOOKKEEPER	1	0.07321537522879805	0.42083333333333334	0.23920265780730898	IBk	Oversampling	CFS con Best First	38	68	161	34	0.3584905660377358	0.5277777777777778	0.6531720038816109	0.1986741139009239

*Esempio del file **.csv** estrapolato usando **BOOKKEEPER.***

2.3 Analisi dei dati e dei grafici

Oltre allo studio improntato sulla scelta del migliore terna di classificatore, feature selection e tecnica di balancing, si è scelto di riportare anche la **percentuale di classi defective** per ogni **release**.



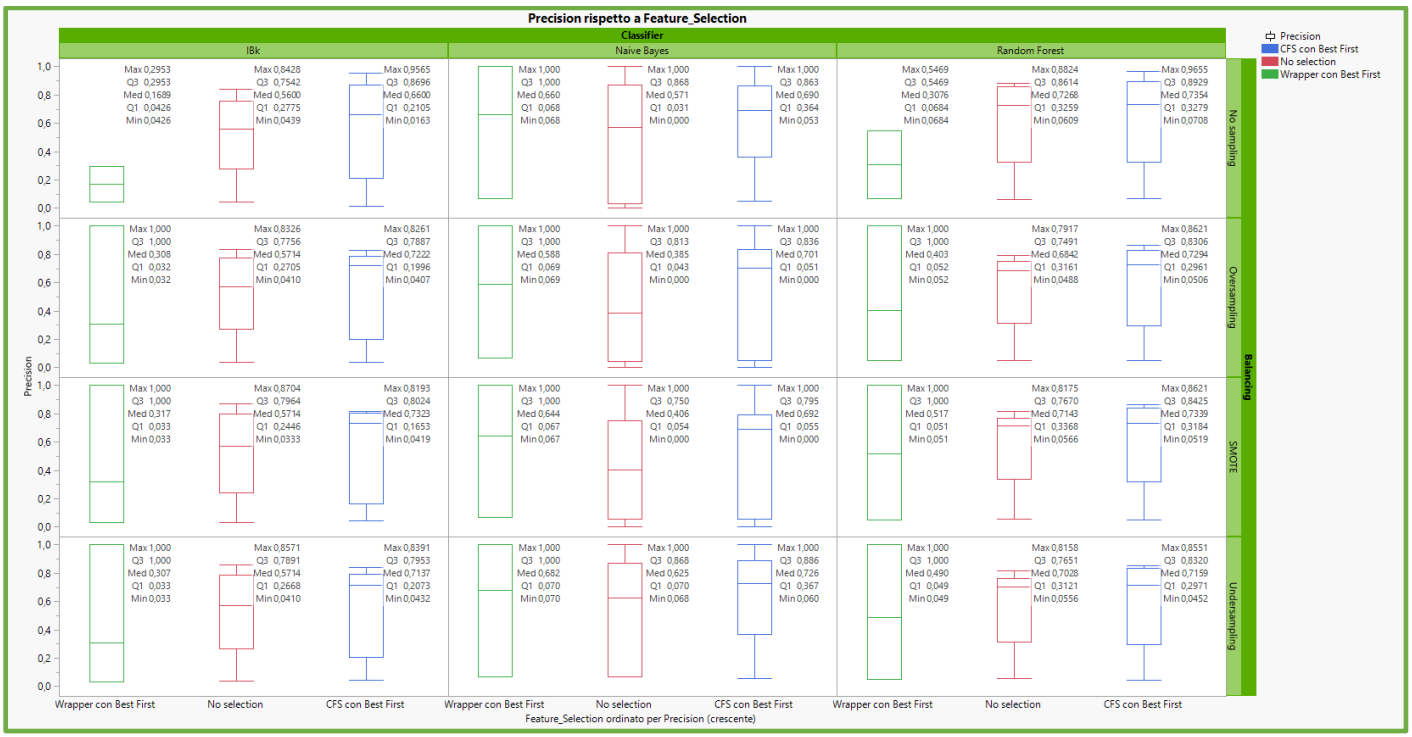


E' immediato notare come **BookKeeper** ha un andamento molto più **instabile** e **crescente nel tempo**, anche se lo scarso numero di release non rendono possibile un analisi più esaustiva sull'andamento nel lungo termine. **Syncope** invece, oltre nel che nel rimanere molto **stabile**, si tiene con valori di defectiveness molto bassi. Per quanto un valore di defectiveness basso sia un fattore positivo per il progetto in se, rende però più difficile il **train** del modello, infatti potendo attingere a pochi esempi di **positività**, è più difficile per il modello capire con una certa accuratezza quali sono gli attributi e i rispettivi valori che concorrono alla difettività della classe. BookKeeper, data la percentuale decisamente più alta di defectiveness permette un train più accurato rispetto a Syncope.

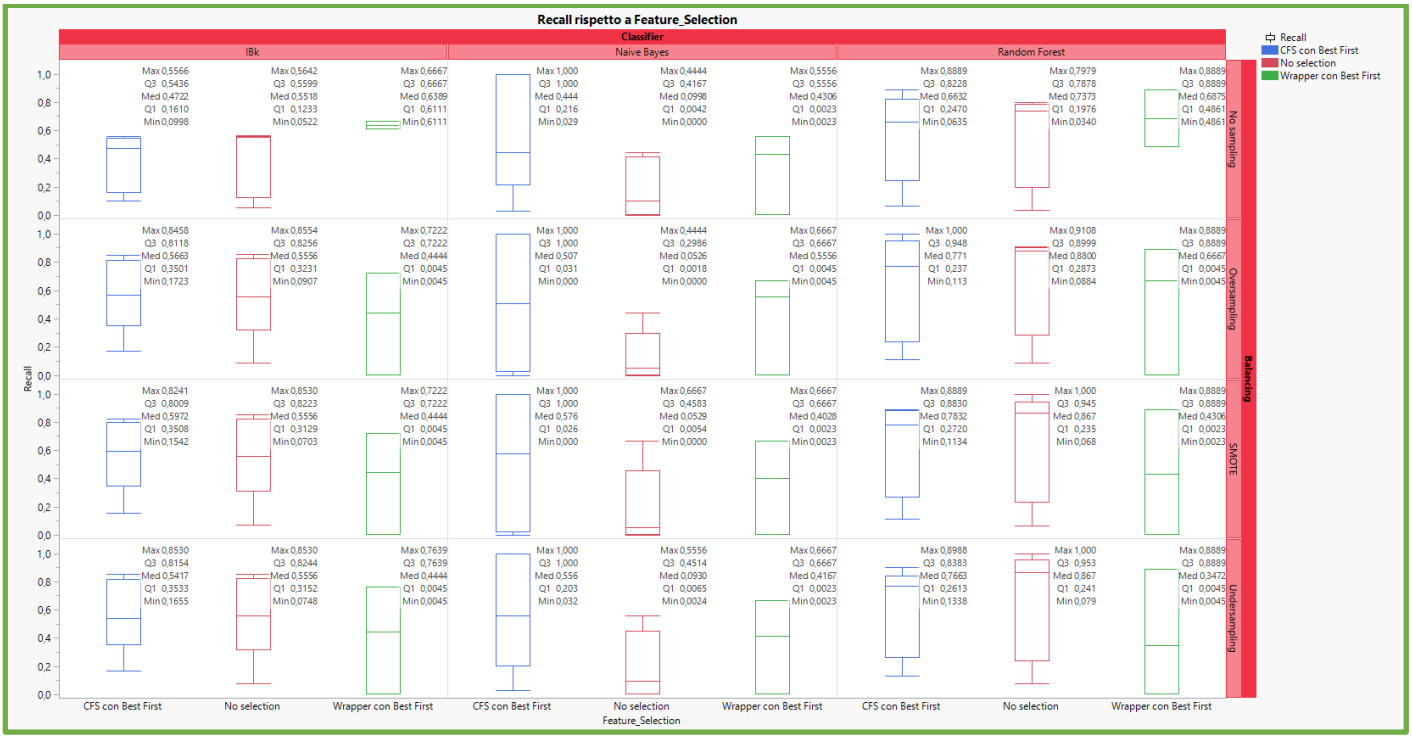
Nella parte successiva, viene presentato uno studio focalizzato sul ricercare la migliore combinazione delle tre variabili del modello di ML per i classificatori misurati: **Precision, Recall, AUC** (o ROC Area) e **Kappa**. Si è ritenuta come scelta migliore quella di usare il **software di JMP**, in particolare è stato creato uno *script* che genera i grafici che verrano presentati di seguito. Per garantire un ordine maggiore, le comparazioni vengono fatte principalmente a parità di classificatore, in modo da evidenziare con maggiore facilità la scelta migliore per il classificatore preso in esame. I due progetti (BookKeeper e Syncope) verrano analizzati separatamente.

Nella cartella /output/grafici sono presenti i grafici riportati nelle pagine successive a risoluzione più alta

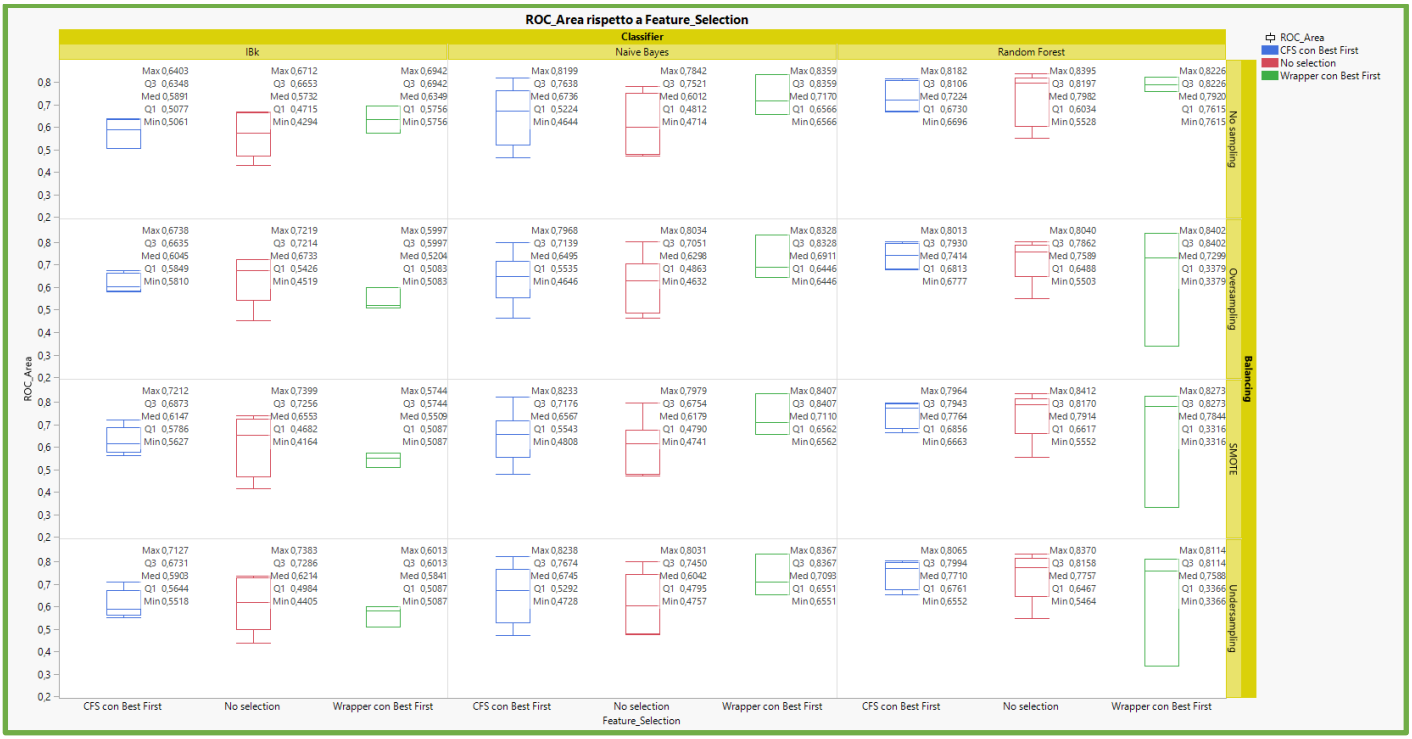
Viene presentato per prima lo studio effettuato su **BookKeeper**:
Nonostante BookKeeper presentasse un buon bilanciamento dei dati (mediamente, 50% dei file defective), il numero di *walk* è molto ridotto rispetto a Syncope, quindi uno studio statistico con così pochi campioni può risultare poco significativo.



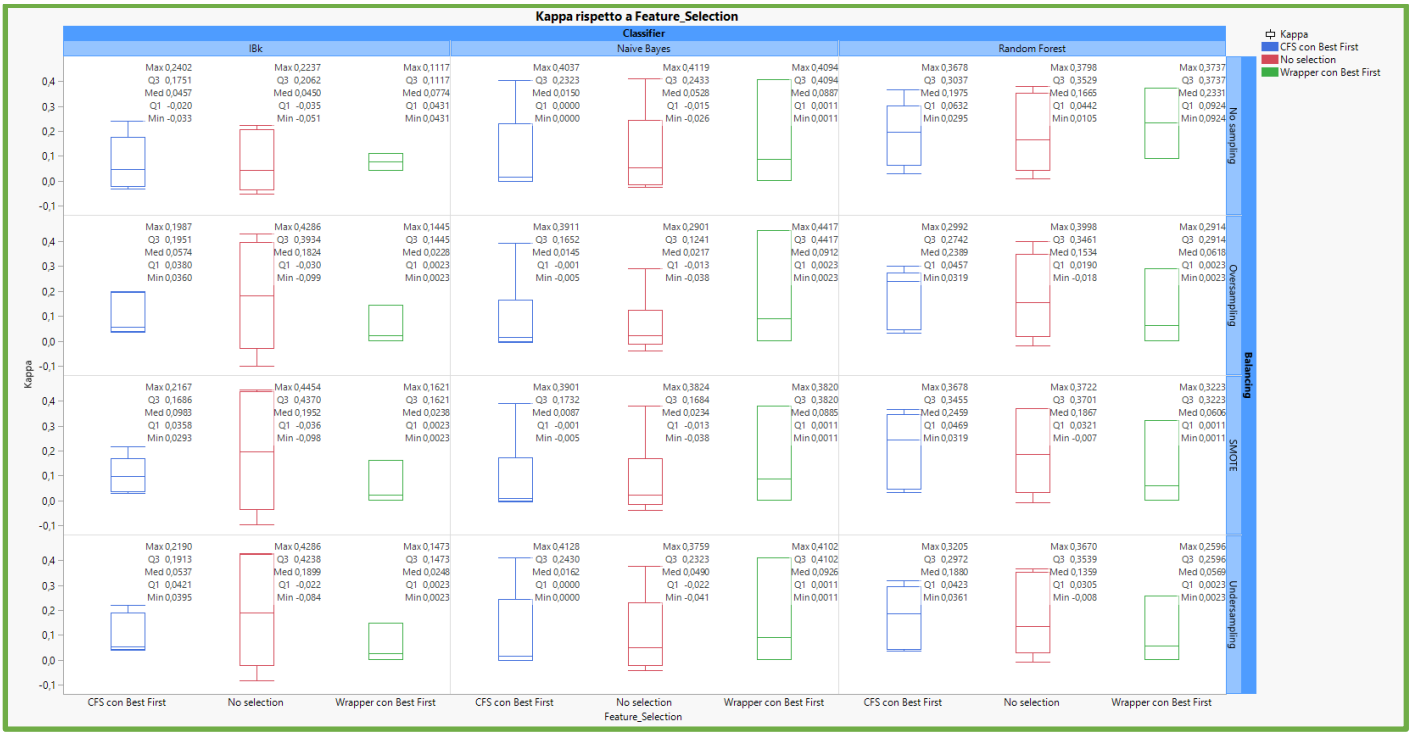
E' immediato osservare che quasi per ogni possibile combinazione, la **precisione** presenta una forte **varianza**, coprendo la maggior parte dei valori possibili. Data la grande varianza, l'unico fattore su cui è possibile fare comparazione sono i vari **quantili** di **primo, secondo** e **terzo ordine**. La combinazione che sembra prestarsi meglio è {**Random Forest**, **SMOTE, No selection**} che presenta una variazione più bassa rispetto alle altre e una mediana sufficientemente buona del 71%.
Per il **nessun classificatore** si riesce a stabilire con assoluta certezza quale combinazione di Balancing e Feature selection fornisca una precisione migliore. Per **IBk** si potrebbe suggerire la combinazione **Oversampling** e **CFS con Best First**, che nonostante compre un ampio spettro di valori, ha una mediana del 72.22% ed i valori dei quantili non discostanti da quelli registrati per le altre combinazioni. Discorso analogo si può fare anche per **Naive Bayes** e **Random Forest**, anche se è piu marcata le migliorie apportante usando sempre **CFS** per *feature selection* e **Undersampling** per **Naive Bayes** e **No sampling** per **Random Forest.**



La **recall** presenta, rispetto alla *precision*, variazioni più forti a seconda delle combinazioni. La scelta che sembra avare più senso è quella di usare {**Ibk, No sampling, Wrapper con Best First**}: nonostante non è la cobinazione che presenta i valori più alti, è l'unica con una variazione molto bassa (e di conseguenza una probabilità di cadere in un intervallo ristrretto intorno al valor medio più alta) e un valore mediano tra i più alti, di preciso del 63%. E' immediato vedere che la combinazione migliore per **IBk e Random forest** è **No sampling** e la feature selection **Wrapper con BF**. Per quanto riguarda **Naive Bayes** è evidente come la **No selection** porta a risultati peggiori, mentre **CFS**, anche se i valori coprono quasi tutto l'intervallo, sembra performare meglio di Wrapper. Non vi è una grande distinzione tra le varie tecniche di sampling, anche se quella che ottiene una **mediana maggiore** è con **SMOTE**, mentre con **Undersampling**, anche se abbiamo una mediana *leggermente inferiore*, abbiamo un valore del **primo quantile** più alto rispetto a SMOTE

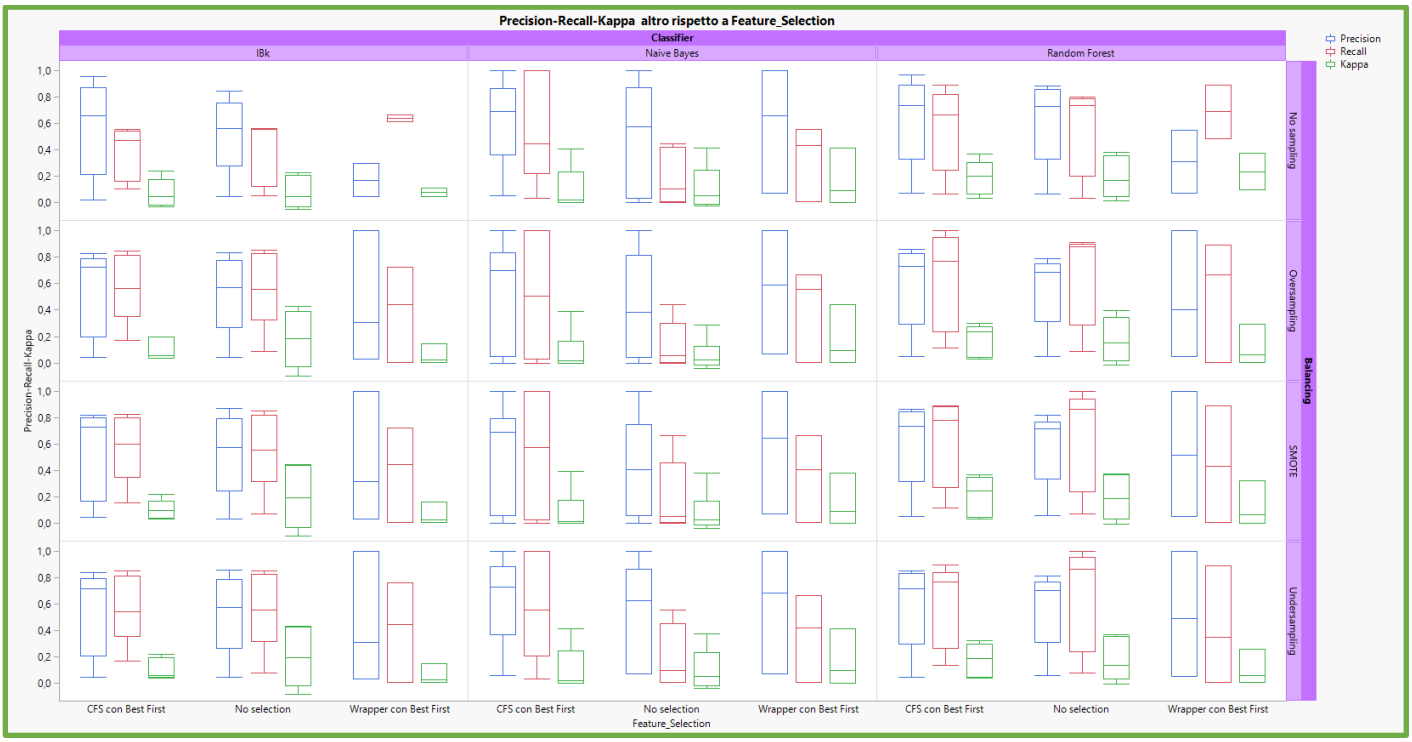


Discorso analogo è possibile fare per il classificatore **AUC** (area sotto la funzione ROC), ed è abbastanza immediato vedere come la combinazione di **Random forest** e **No sampling** presenta risultati migliori e più stabili. Tra le possibili *feature selections*, quella che fornisce valori più accurati è ancora una volta **Wrapper con Best First**, portando alla scelta finale di usare la terna {**Random Forest, No Sampling, Wrapper con BF**}. Ancora una volta, per **IBk e Random Forest** sembra che la coppia **No sampling** e **Wrapper** produca un'valore di AUC maggiore. Per **Naive Bayes** è evidente come il **Wrapper** fornisca un AUC più alto, ma tra le possibilità di balancing non si riscontra una differenza netta.

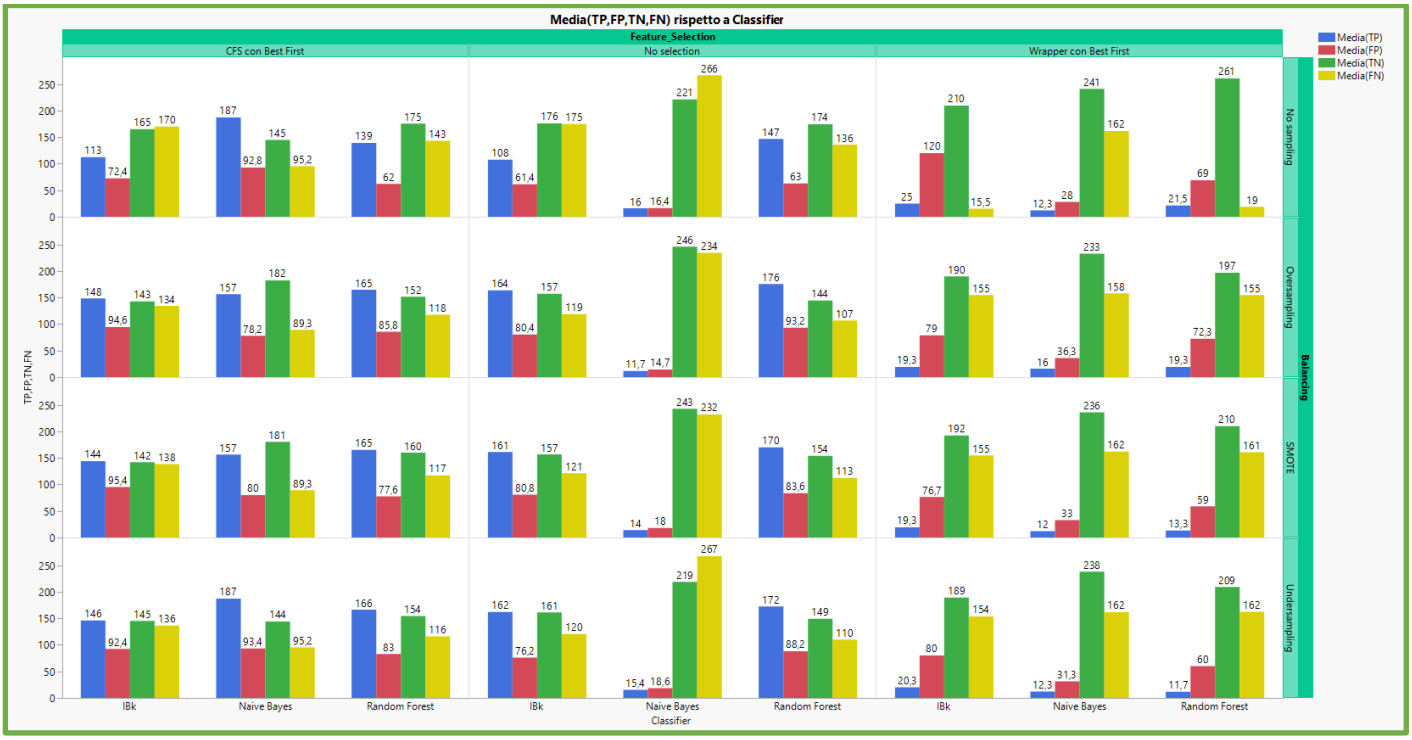


Per il classificatore **Kappa** la situaziuone è simile alla *precision*, infatti i valori risultano molto simili tra loro nella maggioranza dei casi, e quasi tutti con forti oscillazioni. La kappa esprime *quante volte si è stati più accurati di un classificatore dummy* (vedi **Rzero**), ed è immediato osservare come i valori sono mediamente molto bassi, il che fa intendere un guadagno non molto significativo nell'usare i classificatori proposti. In alcuni casi isolati, ci sono stati anche valori negativi, indicando addirittura un **peggioramento** rispetto ad un dummy classifier. Proporio come per AUC, la scelta migliore ricade su {**Random Forest, No Sampling, Wrapper con BF**}, anche se in queste circostanze è opportuno andare a scegliere altre combinazioni fuori da quelle elencate per incrementare l'accuratezza. Per **IBk**, sia CFS che Wrapper perfomano in maniera molto simile, ma la coppia **CFS** e **SMOTE** è quella che sembra fornire una kappa più alta con poche variazioni dalla mediana. Con **Naive Bayes** performano tutti abbastanza male, ma si riesce comunque ad identificare che **No selection** e **No sampling** produca i risultati migliori. Per **Random forest** è abbastanza immediato verificare che la combinazione migliore è **CFS** e **SMOTE**.

Questa trattazione si può concludere andando ad analizzare quello che è il complesso dei classificatori rispetto alle possibli combinazioni, mettendo in un unico *boxplot* **precision-recall-kappa**.



Dal grafico si evince che nel caso di **No selection** e **CFS con BF** si hanno *varianze minori* per *precision, recall e kappa*. Insieme a **Random Forest** e **Undersampling** sembra essere la combinazione di equilibrio migliore, rendendo la scelta {**Random Forest, Undersampling, CFS con BF**} una delle migliori possibili. Ma nonostante ciò, vi è una notevole complessità a stabilire quale combinazione possibile sia la migliore, poiché per possibile combinazione diversa viene migliorato un classificatore e viene peggiorato un altro. A tal proposito, la scelta migliore sarebbe non quella di trovare la migliore combinazione per tutti i classificatori, bensì, scegliere su quali classificatori volersi concentrare e scegliere di conseguenza.

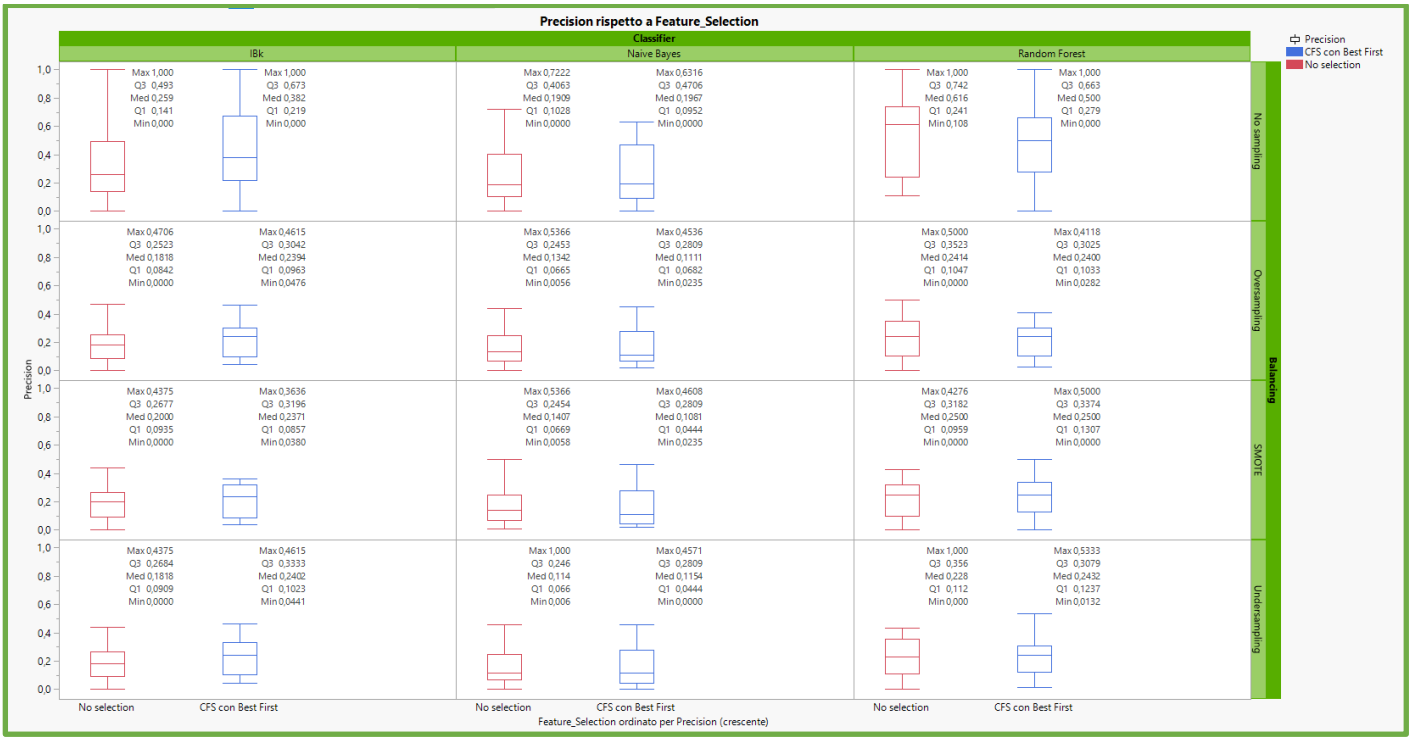


Nell'ultimo istogramma viene presentata la **media tra tutti i walk** di **true/false positve/negative**. La scelta migliore anche qui va fatta in relazione a cosa si vuole ottenere: in alcuni contesti ci si può voler focalizzare solo sul **TP** e **TN**, in altri invece

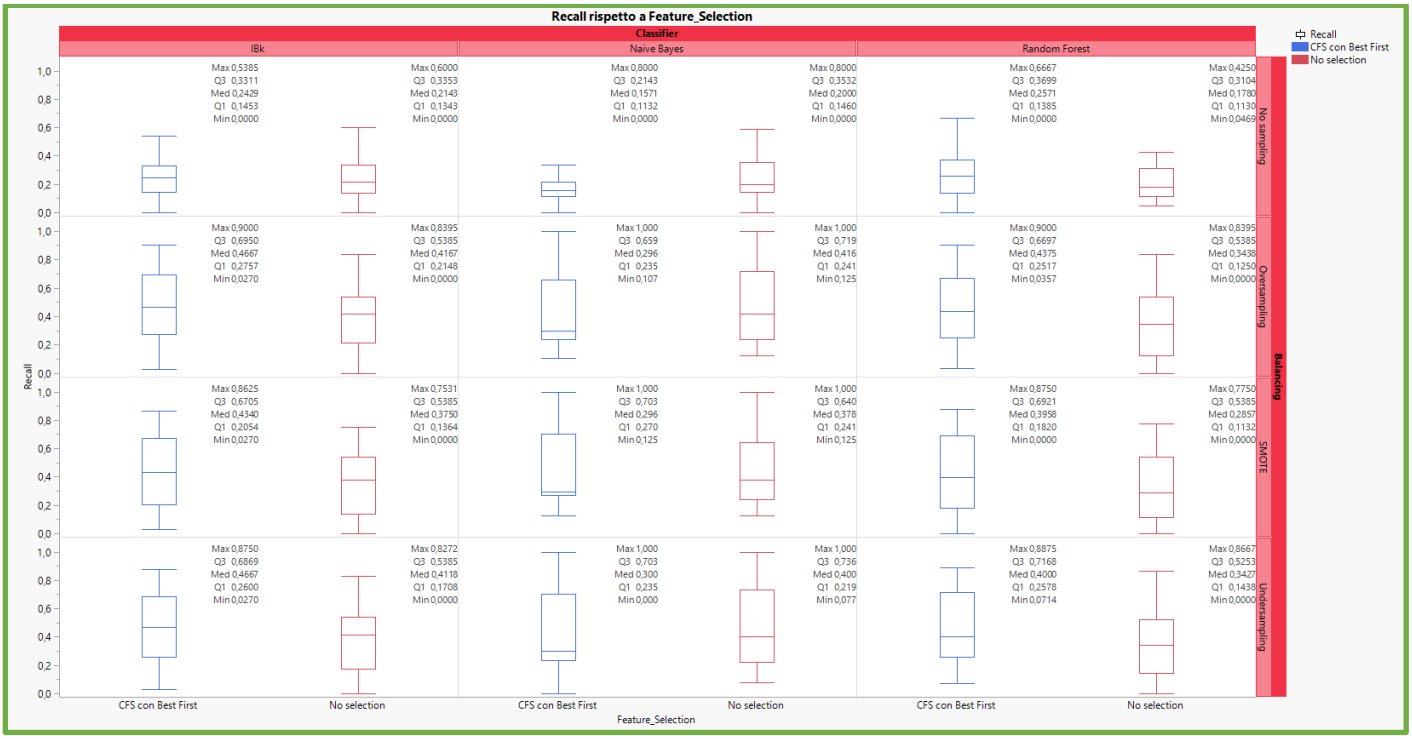
possono essere di interesse altre combinazioni. Nello studio della **defectiveness** si cerca di avere **TN e TP alti** mentre **FN e FP** bassi. E' evidente che la sceltga di usare **Naive Bayes/Random forest/IBk con Wrapper** porta ad un rapporto molto basso tra **TP e FP**, cioè le classi identificate come defective nella maggior parte dei casi non lo sono. Inoltre anche il rapporto tra **TN e FN** non è favorevole: nonostante vengano identificati sempre più TN rispetto a FN, il loro rapporto, eccetto per due casi **senza sampling**, è molto prossimo a 1, e non è un buon valore se si vuole fare una classificazione accurata. La scelta migliore sembra ricadere su {**Naive Bayes, Smote, CFS con BF**}, dove abbiamo più TP/TN rispetto a FP/FN e il rapporto tra essi è il più basso misurato.

Discorso completamente opposto per quanto riguarda invece **Syncope**, infatti, sebbene abbiamo **molte più versioni**, e di conseguenza molti più walk, rispetto a BookKeeper (il che ci consente di avere un campione su cui effettuare l'analisi più ampio e quindi più significativo), il **rapporto** tra classi **defective e non**, come riportato nel grafico a inizio paragrafo, è molto basso. Un rapporto così basso, mentre da un lato indica la qualità di Syncope, dall'altro rende meno accurata il modello di ML che opera sopra alle sue metriche poiché la scarsa presenza di classi defective non permette si creare stime molto accurate su quali metriche e quali valori siano sintomo di buggyness.

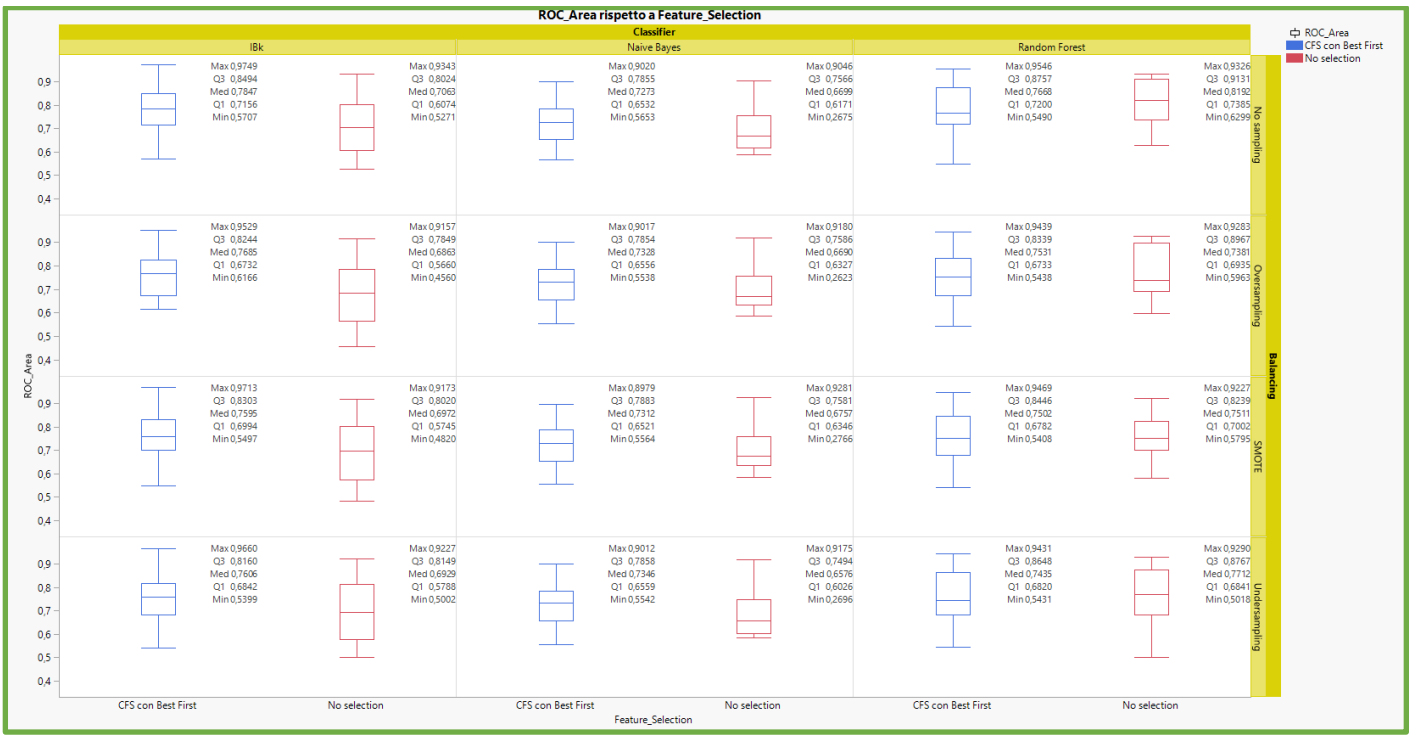
Una *menzione importante* da fare è che, quando applicato feature selection, rimanevano come soli attributi la **Defectiveness** o **Version; Defectiveness**, automaticamente il *walk* viene scartato per ovvie ragioni. Usando **Wrapper** con **Syncope** è stato visto che ritornava sempre una feature selection tra le due sopracitate, comportando lo scarto del Wrapper come feature selection, cosa che invece non è accaduta con **BookKeeper**.



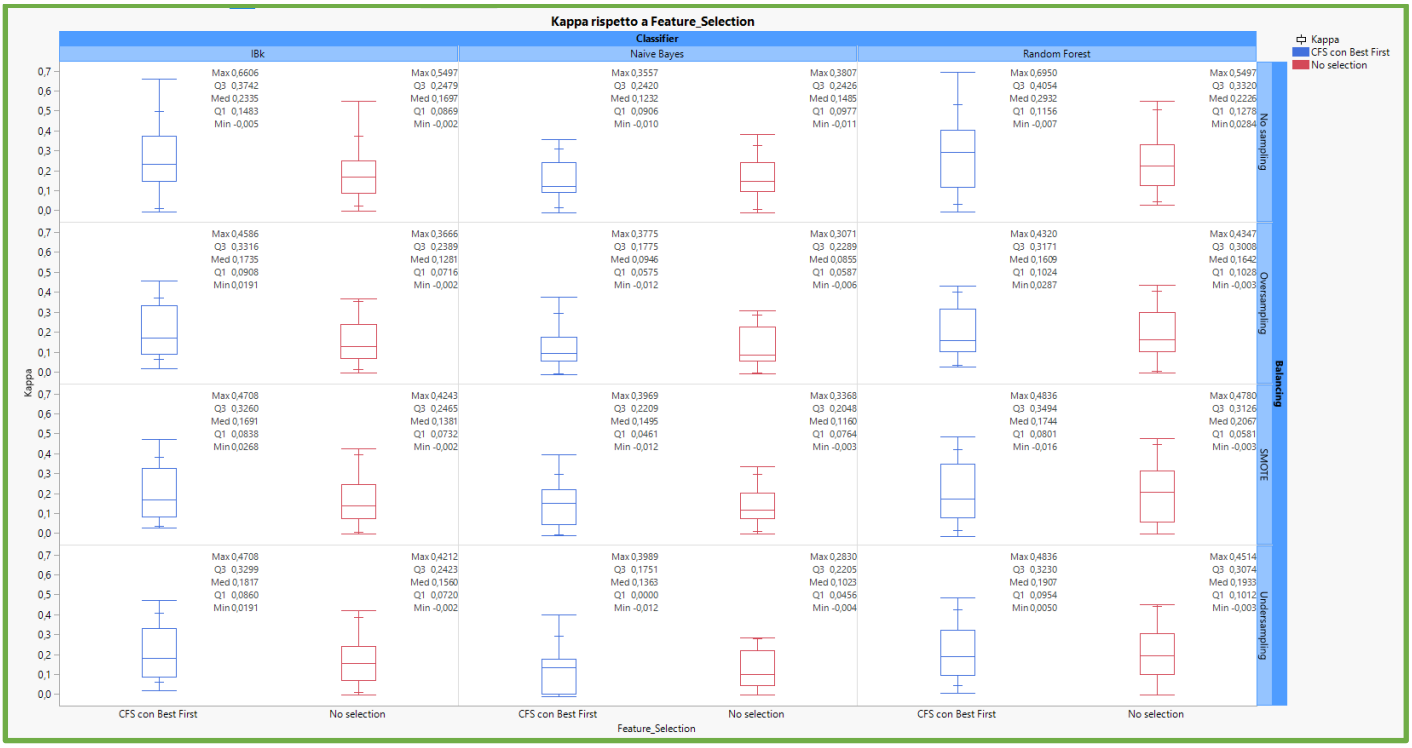
La **precision** rimane molto bassa, specialmente se vengono applicate tecniche di **balancing**, forzando quindi la scelta sul **no sampling**. La scelta del classificatore migliore è abbastanza chiaro che ricade su **Random Forest**, scelta più ardua invece è se usare o meno **feature selection**. Mentre da un lato il **CFS** presenta una variazione leggermente più bassa, dall'altro presenta una mediana leggermente più alta. Con **CFS** si è pero registrato un picco più basso (0% di precisione), quindi sembra ragionevole come scelta usare la terna {**Random Forest, No Sampling, No selection**}



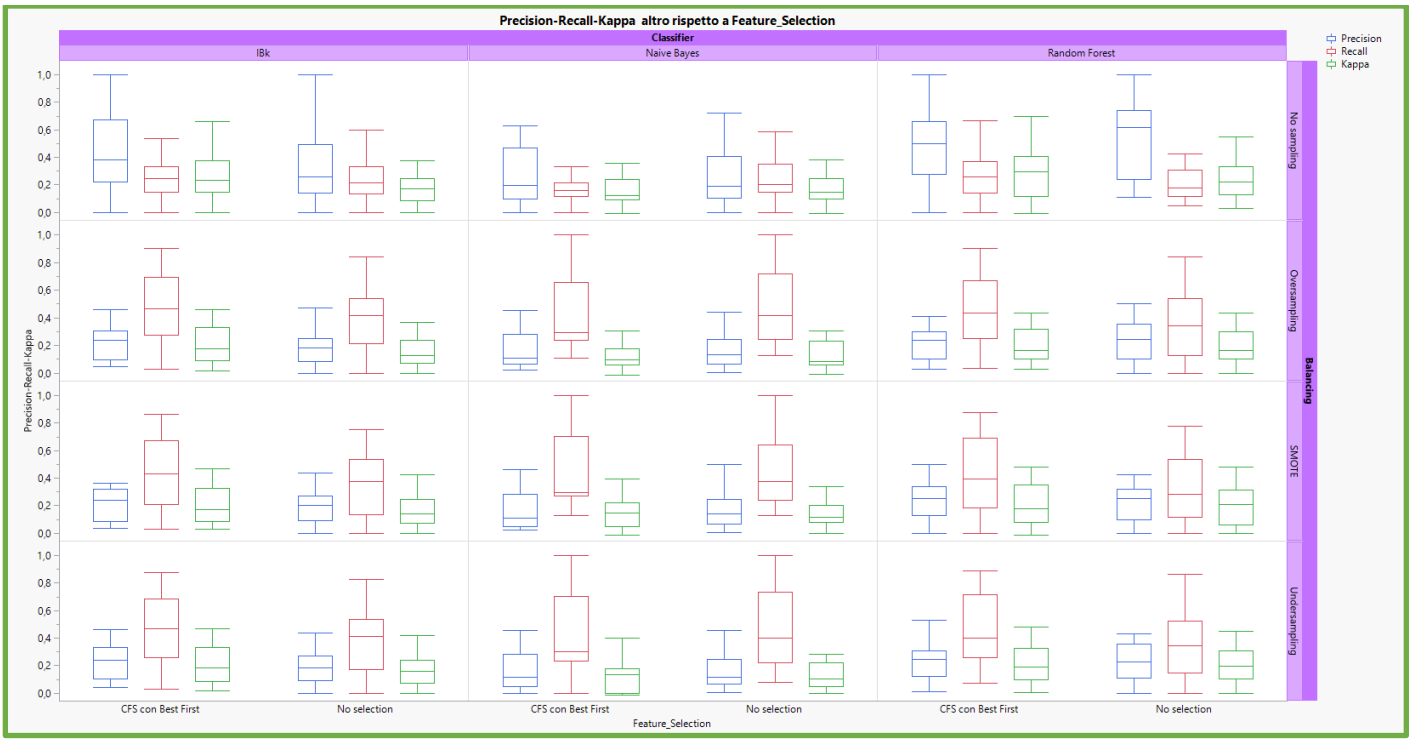
La **Recall** presenta mediamente valori molto bassi e varianza nella maggior parte dei casi alta, ma vengono comunque rilevati 3 possibili candidati, tutti con **CFS** come *feature selection*. Il primo/secondo è **IBk con Oversampling/SMOTE** mentre il terzo è **Random Forest con Oversampling**. Nonostante presentino valori di quartile molto simili tra loro, il migliore sembra essere **IBk con Oversampling**.



A differenza della *precision e recall*, l'**area sotto la funzione ROC** presenta un valore che risalta maggiormente, anche se mediamente tutti raggiungono valori molto buoni, cioè la terna {**IBk, No sampling, CFS con BF**}. Molto analogo è il risultato ottenuto con le altre tecniche di balancing, e leggermente più basso se si usa **Naive Bayes** invece di IBk. Quello che si comporta peggio è invece Random Forest, che è stato il classificatore che ha avuto performance migliori nella maggior parte delle metriche analizzate.



Proprio come per BookKeeper non sembra che mediamente si ottengano grossi guadagni rispetto ad un classificatori dummy, portando il valore di **kappa** relativamente vicino allo 0, ad eccetto per alcuni picchi che raggiungono un valore prossimo a 1. Le combinazioni che presentano *varianza più bassa* sono anche quelle che hanno la *mediana e gli altri 2 qualtili* più in basso. In questo caso, nonostante la marcata oscillazione dei valori, la scelta migliore sembrerebbe essere {**Random Forest**, **No sampling, CFS con BF**}, nonostante anche la scelta di **IBk** rispetto a *Random forest* sembra essere quantomeno valida.



Si è visto come complessivamente, ad eccezione dell'AUC, molte delle combinazioni possibili variano di poco i classificatori di **Precision Recall e Kappa**, infatti è possibile vedere dal grafico riportato che non si notano nette differenze tra le varie possibilità, rimandando quindi la scelta di quale combinazione scegliere dopo la scelta sul focus che si vuole avere. Nel caso particolare di Syncope, dove abbiamo un basso numero di classi defective, non ha molto senso concentrarsi sulla *precisione*, infatti si evince che le combinazioni che hanno mediana più alta, sono anche quelle che hanno varianza più alta, e di

conseguenza poco affidabili. Discorso analogo si può fare per la *recall*, rimanendo con solo la **kappa** e la **AUC**, che sembrano essere parametri molto più significativi in questo contesto. Sia la kappa che la AUC sembrano giovare nel non usare nessuna tecnica di sampling e il **CFS con BF** come tecninca di feature selection. Per la scelta del classificatore invece si opta per **IBk,** il quale ha delle buone performances in entrambi i casi.

Anche dalla media dei **true/false positive/negatives** si evince che usare {**IBk, No sampling, CFS con BF**} è la scelta migliore, infatti ha tra i migliori rapporti tra TP e FP, ed è anche una delle combinazioni che ha catalogato con **maggiore precisione** le classi **non defective.** Questa imprecisione nel catalogare le classi positive e il grande numero di TN è da ricercare nel fatto che, come già accennato, le classi defective rispetto a quelle non defective sono in numero estremamente inferiore (mediamente intorno al 5% per ogni release).

L'applicazione dei risultati ottenuti durante questo studio va ovviamente orientata nei progetti stessi per fare predizione di defectiveness e in progetti con struttura simile. L'aver analizzato dei progetto con una struttura molto diversa tra loro rende possibile una scelta su cosa è più appropiato usare a seconda del prodotto software che ci si trova ad analizzare. Di conseguenza, non è possibile fare un paragone tra i risultati ottenuti tra i due progetti, proprio per la loro struttura fondamentalmente diversa, l'unica similitudine riscontrata è stata che fare **Feature selection con CFS** è risultata nella maggioranza dei casi un successo. Questo è un risultato facilmente intuibile, infatti a seconda del tipo di processo di sviluppo che viene adottato, è facile avere alcune metriche che hanno un'influenza decisamente minore sulla defectiveness. Inoltre il Feature selection previene **situazioni di Overfitting**, che va assolutamente evitata. E' stato interessante notare come le tecniche di **Sampling** non hanno portato ad un miglioramento in tutti i casi, ma molte volte portavano ad un peggioramento, specialmente su BookKeeper. Altro fatto degno di nota, è curioso come tra tutte le tecniche di sampling, SMOTE non si è distinto, nonostante la sua complessità ed i lunghi tempo di esecuzione che fanno sperare in un miglioramento sostanziale. Per quanto riguarda i classificatori, è stato evidente come **Naive Bayes** non sia la scelta da fare nella maggior parte dei casi, mentre **Random Forest** e **IBk** hanno presentato molte volte risultati analoghi.