Report ISW2 – 2019/2020

Sezione prof. Falessi

Deliverable 1:
Misura della stabilità di un attributo di progetto

Deliverable 2:
Predizione della difettosità di codice software

Cecilia Calavaro

Matricola 0287760

INTRODUZIONE

Il report per il progetto della parte del corso di ISW2 insegnata dal prof. Falessi si divide nell'analisi delle due Deliverables assegnate, esaminate separatamente nei rispettivi paragrafi, in cui si descrive lo studio effettuato su progetti open source di Apache.

I progetti esaminati sono:

- **Calcite** (per la Deliverable1)
- **BookKeeper** e **ZooKeeper** (per la Deliverable2)

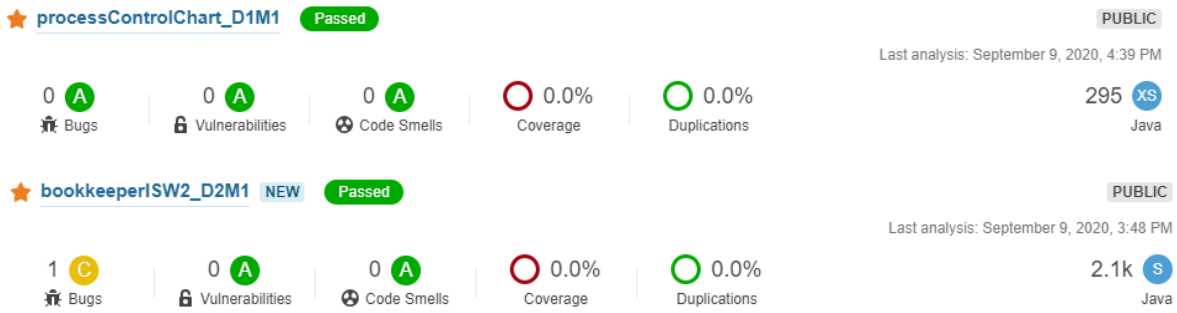
Per entrambe le deliverables sono stati prodotti dei risultati tramite un'automazione fornita da codice Java, che si può trovare nelle seguenti repository di **GitHub**:

- Link **Deliverable1** : https://github.com/ceciliacal/processControlChart_D1M1
- Link **Deliverable2**: https://github.com/ceciliacal/bookkeeperISW2_D2M1

Sono state inoltre sfruttate le integrazioni con TravisCI e SonarCloud, di cui viene riportato il link di seguito:

- **TravisCI**:
  - Link **Deliverable1**: https://travis-ci.org/github/ceciliacal/processControlChart_D1M1
  - Link **Deliverable2:** https://travis-ci.org/github/ceciliacal/bookkeeperISW2_D2M1
- **SonarCloud**:
  - **Link Deliverable1**: https://sonarcloud.io/dashboard?id=ceciliacal_processControlChart_D1M1
  - **Link Deliverable2**: https://sonarcloud.io/dashboard?id=ceciliacal_bookkeeperISW2_D2M1

Per quanto riguarda l'analisi del codice degli applicativi, si hanno in entrambi i casi 0 code smells:



Inoltre, in ognuna delle repository, è presente una cartella chiamata "**outputFinali**" in cui sono presenti i file di tipo .csv e .xlsx che contengono gli **output delle deliverables**. Si è deciso di utilizzare anche file .xlsx per riportare alcuni output grazie alla maggiore leggibilità che offrono rispetto ai .csv.

Nota: La **deliverable2** è interamente contenuta nella repository chiamata "**bookkeeperISW2_D2M1**". Il nome può essere fuorviante ma la repository è stata creata alcuni mesi fa e il nome allora non è stato scelto con attenzione. Si è deciso di non cambiarlo perché le varie integrazioni sono funzionanti, quindi per evitare che sorgano ulteriori problemi a oltre a quelli già riscontrati precedentemente, si è preferito fornire una spiegazione in questa nota.

CONFIGURAZIONE

Per lo sviluppo delle applicazioni e l'analisi dei risultati sono stati utilizzati:

- Eclipse 2020 come IDE (JDK11)
- GitHub per il mantenimento della repository in remoto
- TortoiseSVN come strumenti per il Version Control
- Ant per la build in locale
- TravisCI per la build in remoto
- SonarCloud per la rilevazione di code smells
- Le librerie di json, jgit, weka e SMOTE, tutte incluse in una cartella chiamata "jar" nella repository GitHub
- Excel e JMP per la manipolazione dei dati e per la realizzazione dei grafici

DELIVERABLE 1

INTRODUZIONE

L'obbiettivo della prima deliverable è quello di andare a misurare la stabilità di un attributo di un progetto al fine di formulare osservazioni mediante uno statistical process control. Lo statistical Process Control consiste nell'applicazione di un metodo matematico o statistico che consente di contenere l'esito di un processo all'interno di specifici limiti, determinati attraverso lo studio della variazione naturale dei limiti del processo stesso.

In particolare, l'attributo che si va ad analizzare è il **numero di ticket** risolti nel progetto Calcite di Apache, in un arco temporale che va da luglio 2014 ad aprile 2020.

Al fine di garantire una qualità alta di un prodotto software è importante che alcuni attributi, come appunto il numero di ticket o di bug risolti, siano stabili durante il processo di sviluppo, in modo da riuscire a controllare l'evoluzione del processo stesso e da predire la variazione dei risultati che ci si aspetta da quest'ultimo.

ANALISI DEL SORGENTE

Per effettuare la raccolta dei dati è stato necessario utilizzare le piattaforme di **Jira** e di **GitHub**, facendo uso rispettivamente dell'API di di Json e di JGit. Da Jira sono raccolti, tramite una query, i ticket con risoluzione di tipo "*fixed* " e con stato "chiuso" o "risolto". Viene restituito un file di tipo *.json*, da cui vengono estrapolati gli id dei ticket ed inseriti in una lista. Da GitHub, una volta effettuato il clone della repository, si ottiene il **log** dei commit, da cui vengono prelevati quelli relativi agli ID dei ticket precedentemente importati. Tra i commit d'interesse, si va a considerare, per ogni ticket, soltanto **l'ultimo commit** relativo ad esso, ovvero il più recente, in quanto rappresenta il commit in cui il ticket viene risolto (fixed ticket).

La scelta di assumere che un ticket venga risolto quando non ci sono più commit relativi ad esso è dettata dal fatto che Jira, non essendo completamente automatizzato, potrebbe riportare delle informazioni errate, mentre su GitHub questo scenario non è possibile, in quanto la data in cui avviene un commit viene impostata direttamente dal sistema.

Terminata la raccolta delle informazioni, si contano quanti ticket hanno la stessa data e vengono inseriti in una struttura dati contenente l'anno, il mese ed il numero dei fixed tickets nel rispettivo mese.

A partire da quest'ultima viene estrapolato, come richiesto da specifica, il file calcite_output_D1M1.csv.

In più è stato creato anche il file graficoCalcite.xlsx, in cui sono state calcolati dei dati aggiuntivi e necessari per la realizzazione dello statistical process control: la media, il limite superiore, il limite inferiore, la deviazione standard e un coefficiente della deviazione standard.

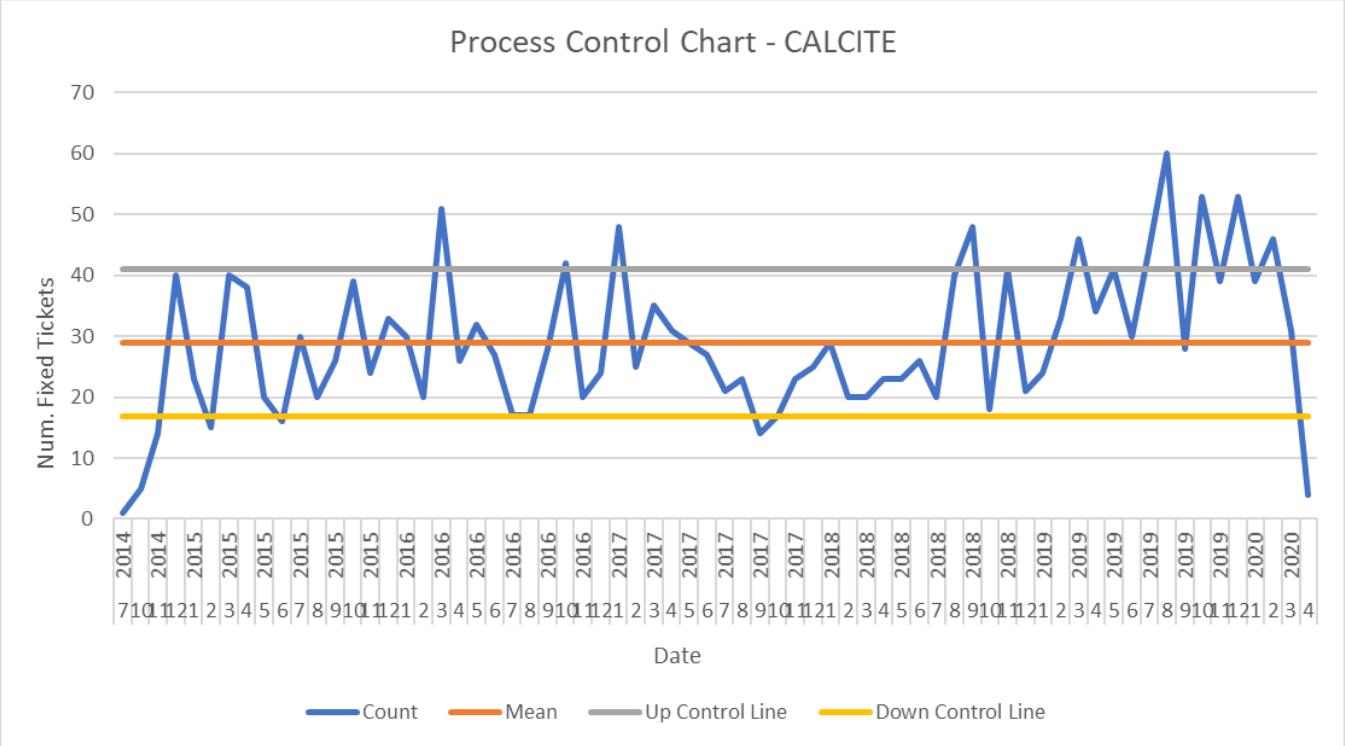


Grafico 1 – Process Control Chart di Calcite con coef_STD = 1

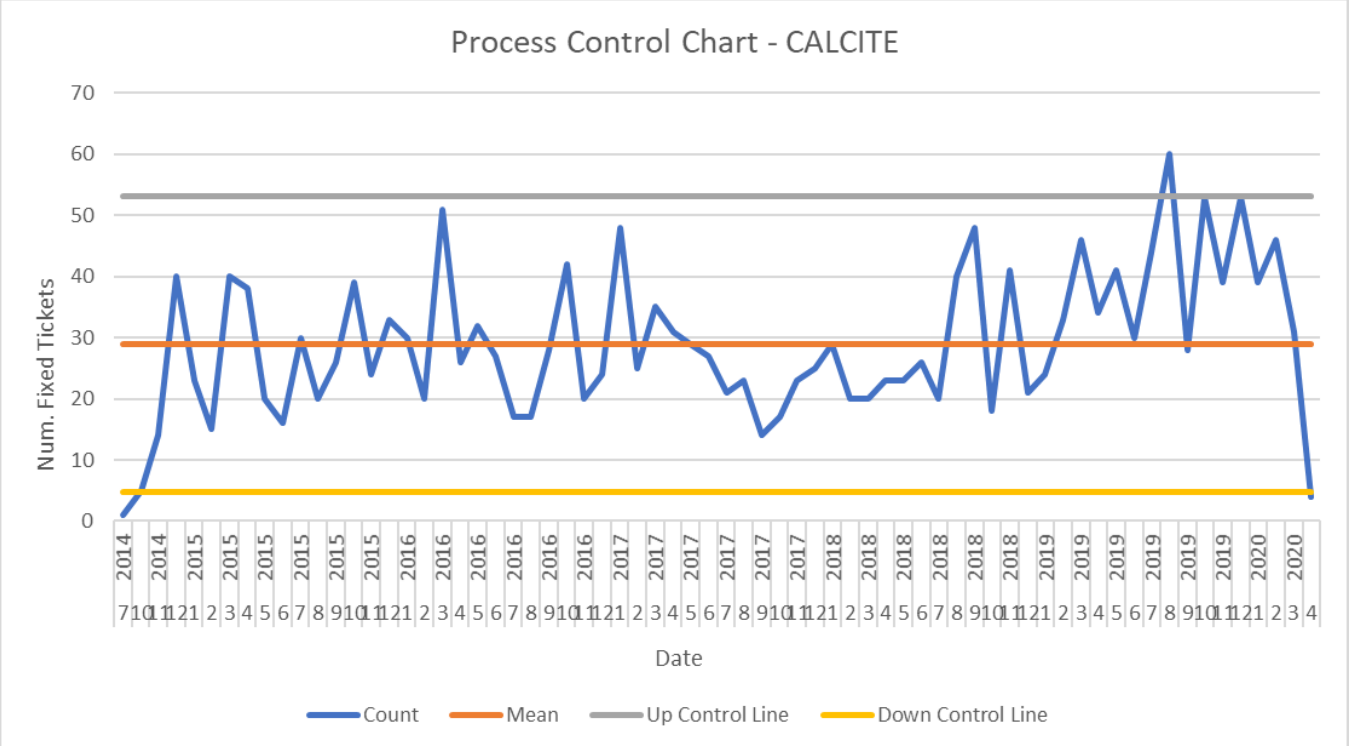


Grafico 2 – Process Control Chart di Calcite con coef_STD = 2

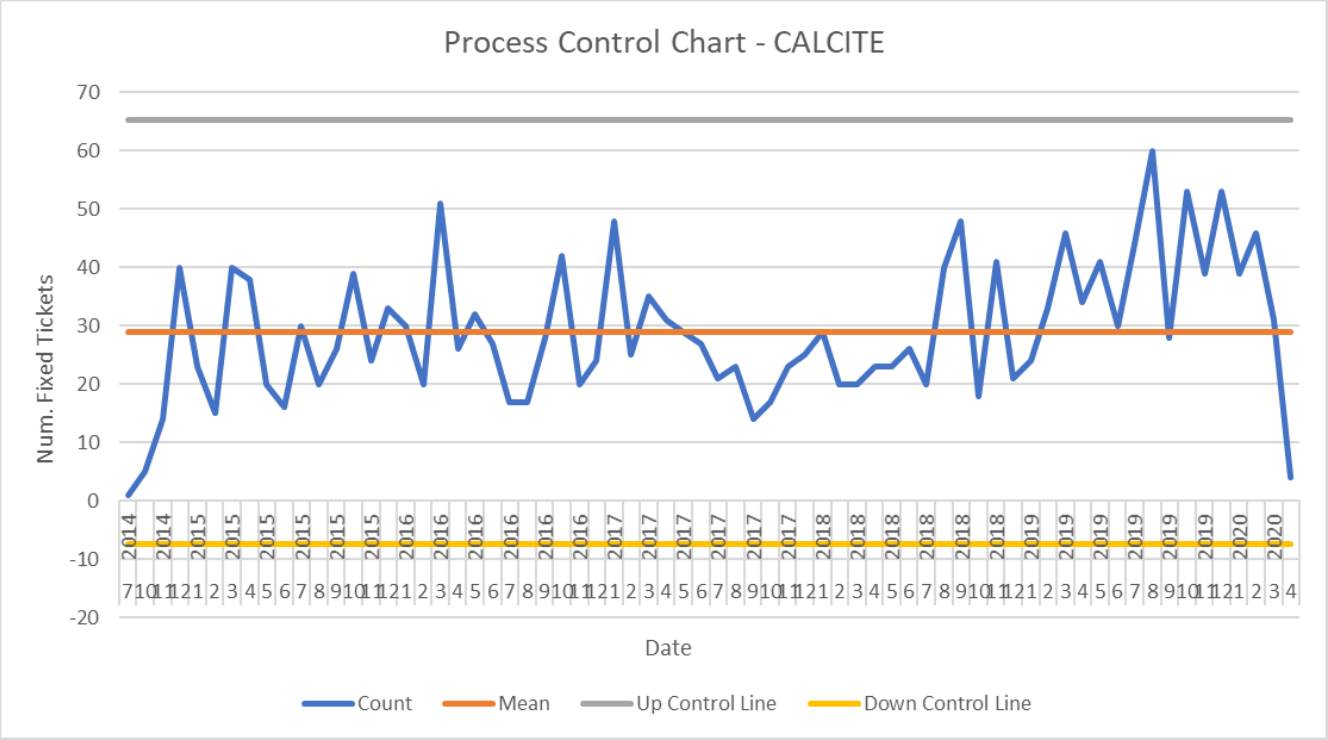


Grafico 3 – Process Control Chart di Calcite con coef_STD = 3

ANALISI DEI GRAFICI ED OSSERVAZIONI

Come si può notare dal Grafico1, oltre al numero di fixed tickets in ogni mese, sono riportate anche la **media**, il **limite inferiore** ed il **limite superiore**. Quest'ultimi rappresentano i limiti entro cui un attributo (o ciò che si sta misurando) sia considerato stabile: quando i valori cadono al di fuori dei due limiti, significa che in quell'intervallo temporale ci si trova in uno stato d'instabilità.

Per computare i due estremi, sono state applicate le seguenti formule:

upper bound = mean + coef_STD * dev_standard

lower bound = mean - coef_STD * dev_standard

Inoltre, si è scelto di introdurre il parametro "**coef_STD**", il coefficiente per cui viene moltiplicata la deviazione standard, per poter mostrare come cambia il grafico al variare di quest'ultima, e quindi per osservare in quali casi la curva va oltre i limiti superiore ad inferiore. Generalmente si considera **coef_STD=3**, e quindi dal Grafico3 si può vedere come il processo di sviluppo sia sempre stabile.

Ovviamente, l'andamento della curva rimane lo stesso nei tre grafici. Si può osservare che la curva che descrive il processo di sviluppo segue sempre un andamento molto **irregolare**: nei primi mesi, dalla fine del 2014 alla prima metà del 2015, la curva interseca la retta della media subendo dei picchi di variazione molto alti. L'andamento poi si **stabilizza** leggermente nella seconda metà del 2015, per poi tornare ad essere nuovamente **irregolare**, fino ad arrivare a circa la metà dell'anno 2017, in cui va al di sotto della media per la durata di un intero anno, in cui vengono risolti molti meno del ticket del solito. Segue poi un continuo **aumento**, fino ad arrivare a luglio 2019 e ai primi mesi del 2020, in cui si ha un'attività molto più intensa: si può osservare come ad agosto 2019, nel grafico 3, la curva raggiunga quasi il limite superiore.

Nello stesso mese, nel grafico 2 la curva supera di poco l'upper bound, mentre nel grafico1 quasi più della metà di essa è al di fuori della retta che delinea l'estremo superiore.

L'elemento comune che si può osservare dai tre grafici è che un **andamento** così **scostante** è indice di un approccio **poco continuativo** nello sviluppo del prodotto software, sebbene prendendo come riferimento il grafico 3, in cui vi è uno scarto considerevole rispetto alla deviazione standard, il processo rimanga sempre stabile.

DELIVERABLE 2

INTRODUZIONE

Nella deliverable2 viene effettuato uno studio di due progetti open source di Apache, BookKeeper e ZooKeeper.

Lo scopo è quello di identificare quali sono le **classi di un progetto più predisposte ad avere dei difetti**. Si vuole quindi andare a predire, con una certa precisione, su quali file sia più probabile l'insorgenza di **bug** nel codice sorgente tramite l'utilizzo di modelli di **Machine Learning**, sulla base del valore di metriche calcolate sui file di progetto. Nei modelli predittivi si è fatto uso di tecniche di **sampling** e di **feature selection**, per poi poter andare ad osservare quale combinazione di tali tecniche produca, per diversi classificatori, il risultato migliore.

Il vantaggio principale che comporta la stima della difettosità di una classe è una **riduzione dei costi** nella fase di **testing** e di **code review**. In questo modo, infatti, essendo al corrente di quali file *.java* presentano dei bug, sarà noto fin dal principio quali di essi è necessario testare in maniera più esaustiva e meticolosa rispetto ad altri.

La deliverable si suddivide in due milestones:

Nella **Milestone1**, l'obbiettivo è quello di produrre un dataset che riporta, per tutti i file di ogni release, la difettosità di una classe ed il valore delle relative metriche. Le fasi principale del lavoro che svolto sono le seguenti:

- Si utilizza **Jira** per raccogliere informazioni su tutte le release del progetto e sui **ticket** dei bug.
- Tramite **Github** viene recuperato il log dei **commit**, e di conseguenza i file per ogni release.
- Si lavora solo con la **prima metà** delle **release**, quindi le più datate. Il motivo per cui si escludono le release più recenti è che, nella pratica, si è visto che quest'ultime avevano sempre meno difetti rispetto a quelle più vecchie, perché i dati recenti sono più "acerbi". Questo perché spesso il fix dei bug deve ancora avvenire, o magari ancora non ci si è accorti della sua presenza, mentre in quelle iniziali è più probabili che sia stato osservato in quanto è passato più tempo.
- Per ogni release, viene calcolata la difettosità di una classe e le metriche relative ad essa, che vengono computate facendo un *merging* delle informazioni presenti sulla repository git del progetto e su Jira tramite una query. Per il calcolo della buggyness ci si ritrova di fronte a due scenari: 1. Su Jira è presente l'**AV** del ticket, quindi abbiamo dei dati concreti riguardo a quando un determinato bug è stato presente nel software; 2. L'AV non è disponibile, pertanto va stimata mediante un **algoritmo** basato sulla **stabilità del ciclo di vita del difetto**. Maggiori dettagli su come vengono effettuate queste operazioni sono riportati successivamente nel report.
- Viene infine elaborato un dataset contenente la bugginess di ogni release e le rispettive metriche, relativamente ad ogni release.

Nella **Milestone2**, il dataset realizzato precedentemente viene utilizzato come "input" di **un'analisi** dei dati ottenuta mediante **modelli di Machine Learning**. Tramite l'impiego della libreria messa a disposizione dal software **Weka**, è stato valutato quale classificatore, associato con determinate tecniche di FS/balancing, produca risultati più accurati nell'etichettare classi come difettose o meno.

Quanto riportato ha lo scopo di fornire una visione generale dell'applicativo, in modo da evidenziare gli step salienti che hanno concorso alla realizzazione del sistema, i quali verranno analizzati in maniera dettagliata nella seguente analisi del codice.

ANALISI DEL CODICE SORGENTE

Il sorgente dell'applicativo è costituito da **due package**, **project.bookkeeper** e **milestone.two**, in cui sono presenti, rispettivamente, le classi necessarie all'implementazione di quanto richiesto dalla milestone1 e dalla milestone2.

Le classi sono state sviluppate rispettando **l'architettura** prevista dal modello **Boundary-Control-Entity**. Il ruolo di una classe è facilmente comprensibile in quanto nel suo nome è presente il suffisso "Boundary", se si occupa di far interagire il sistema con l'esterno, o "Control" se il suo compito è quello di sviluppare la logica che c'è dietro l'implementazione di una *feature* del sistema, quale il calcolo delle metriche o dell'algoritmo Proportion per calcolare la difettosità, oppure quello di guidare il flusso d'esecuzione principale dell'applicativo, facendo da tramite nella comunicazione tra classi di tipo Boundary e di tipo Entity. Tuttavia, le classi Entity non sono etichettate con il suffisso che spetterebbe loro per facilitare la comprensione del sorgente, sia in termini di lettura che nel momento della scrittura vera e propria.

MILESTONE1 : Calcolo della bugginess e delle metriche

L'obbiettivo della milestone1 è la creazione del dataset che deve contenere la buggyness e le metriche dei file presenti in ogni **release** appartenente alla **prima metà del progetto**. Tuttavia, si è scelto di raccogliere **inizialmente tutte le release**, in modo da garantire successivamente un'analisi accurata dei dati da considerare per il calcolo della difettosità, in modo da **evitare** un'eventuale **perdita d' informazioni** importanti: questo passaggio verrà spiegato in maniera più approfondita in seguito, quando verranno analizzati i criteri per cui i ticket verranno presi in considerazione relativamente ai propri AV, OV, e FV.

Tutte le **release** del progetto vengono recuperate da **Jira** effettuando, tramite l'url *https://issues.apache.org/jira/rest/api/2/project/projName*, un'interrogazione che restituisce un file di tipo **json** da cui vengono raccolte le informazioni riguardo l'ID della release, il nome, e la data. Inoltre le release vengono numerate, associandone ognuna ad un **indice**. Successivamente, le release vengono ordinate in modo cronologico per essere poi usate nella raccolta delle informazioni relative ai file.

Oltre a Jira, un altro strumento fondamentale per la raccolta dei dati preliminari è **Git**. Dopo aver effettuato il clone della repository, tramite l'API di JGit viene recuperato il **log** dei **commits** forniti da Github. Scorrendo il log, si considera **l'ultimo commit di ogni release**: a partire da esso, si recuperano tutti i **file** presenti nella repository del progetto in quel momento, andando a ripercorrere il *tree* di tale commit. Infatti, in Git, i "*tree*" (o alberi) sono degli oggetti che creano una gerarchia tra i file presenti in una repository. Inoltre, è bene specificare che i commit considerati sono quelli appartenenti al **branch** di tipo **master**. I files recuperati vengono messi all'interno di una struttura dati che rappresenta una entry del dataset, in cui inizialmente sono presenti soltanto il numero di release e il nome del file, e che poi verrà popolata dal valore delle metriche scelte.

I ticket del progetto sono anch'essi raccolti da Jira tramite una **query** che recupera i **ticket** di bugs aventi *resolution* di tipo ***fixed*** ed uno stato di tipo ***resolved*** o ***closed***. Per effettuare quest'operazione si segue lo stesso approccio usato nella raccolta delle release, ovvero ottenendo un file di tipo json da cui vengono estrapolate le informazioni necessarie, come l'**id** del ticket, l'**AV** e le date di **creazione** e **risoluzione**.

L'AV (**Affected Versions**) è una lista di release **non sempre presente** su Jira in cui vengono riportate le release del progetto affette dal bug relativo ad un certo ticket. Il range compreso nell'AV è **[IV, FV)**, dove l'IV rappresenta **l'Injected Version**, ossia la release in cui il bug è stato introdotto nel software e FV è la **Fixed Version**, la prima release in cui il bug è assente perché è avvenuto il fix. Dalla data di creazione si può definire l'**OV** (**Opening Version**), quella in cui viene creato il ticket perché ci si accorge di un bug, mentre dalla data di risoluzione si definisce l'FV di un ticket. Tuttavia, può capitare che le AV recuperate da Jira siano **incoerenti**: ci sono casi in cui risulta che IV>OV oppure IV>FV.

Come già accennato nella sezione introduttiva del report, per andare a **calcolare la difettosità** di una classe si usa l'**AV** fornita da Jira qualora sia presente, altrimenti viene utilizzato l'algoritmo **Proportion**. Pertanto, è necessario fare un preambolo per spiegare i criteri secondo cui le informazioni riportate dall'AV di un progetto open source, quando disponibili, siano ritenute affidabili ed utilizzate per costruire il dataset dei difetti, e quali scartate poiché inconsistenti.

Per andare ad etichettare una classe come difettosa in una versione, è necessario conoscere l'AV. La **Fixed Version** di un ticket è **sempre nota**, l'IV non lo è: bisogna quindi calcolare una "**predicted IV**". Ci sono diversi metodi finalizzati ad identificare quando un bug è stato introdotto, ad esempio **Simple**, **SZZ** e **Proportion**.

L'algoritmo **SZZ** non viene utilizzato in quanto ha molte **limitazioni**: è difficile da implementare, ha una recall alta ed un'accuratezza non perfetta.

L'algoritmo **Proportion**, invece, è un algoritmo basato su **stabilità del ciclo di vita del difetto**, secondo cui più il difetto ha "**dormito**", cioè è rimasto nel codice sorgente senza che sia stato osservato, e più sarà difficile da identificare. L'intuizione di base è che all'interno dello stesso progetto, la **proporzione** del numero di release che sta tra IV e OV è la stessa che sta tra OV e FV: in altre parole, che ci sia un numero proporzionale di versioni per trovare un difetto e per risolverlo. Tale proporzione può essere calcolata con la formula:

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

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

Inoltre, ci sono tre varianti dell'algoritmo per il calcolo dell'AV: **Cold Start**, che prevede di computare P su altri progetti e di utilizzarla sul progetto attuale; **Increment**, secondo cui P debba essere calcolata come media tra i bug risolti andando avanti con le release; **Moving Window**, che prevede di calcolare P come la media in un sottoinsieme di difetti già risolti, pari all'1%. La variante di Proportion che è stata implementata è l'ultima, ossia Moving Window.

L'idea di fondo, quindi, è quella di andare a calcolare P su dei bug precedenti di cui si conosce l'AV, per poi usarla nel calcolo della PredictedIV di cui non si conosce l'AV, poiché non disponibile.

Una volta presa visione di come funziona Proportion, è ovvio che debbano esserci dei **criteri** per fare una **selezione di quale AV siano utilizzabili** perché riportano informazioni **coerenti**. In particolare, sono state adottate le seguente scelte progettuali:

1. Per ogni ticket, se **OV = 1**, di conseguenza si pone **IV =1**, in quanto è la prima release disponibile. Con 1 ci si riferisce all'indice con cui sono state numerate le release.
2. Sui ticket in cui è presente l'AV, se **IV>OV** o **IV>FV**, l'AV presa da Jira non è affidabile, pertanto viene ignorata e **ricalcolata** con Proportion, come se fosse assente.
3. Se **OV = FV** e il ticket non ha AV (o perché non sia presente, o perché inconsistente - cfr. caso precedente), il **ticket** viene **rimosso** dalla lista di ticket presente nel sistema, in quanto non è possibile calcolare la P.

Una volta effettuata questa "scrematura", i ticket su cui si calcola P sono quelli che hanno IV (come primo elemento dell'AV recuperata da Jira) e di cui FV e OV sono diversi. Sui restanti, viene computata la PredictedIV con l'ausilio della moving window.

Dopo che è stato applicato l'algoritmo Proportion su tutti i ticket e quindi tutti hanno un IV, si è ritenuto opportuno andare a ridefinire l'AV di ogni ticket, in quanto tutti i dati necessari sono finalmente disponibili e coerenti, e viene infine calcolata la bugginess di ogni classe in tutte le release. Il motivo per cui sono state utilizzate **tutte le release fino al momento della scrittura del dataset**, è stato per **evitare di perdere dei ticket** in cui l'AV sia "a cavallo" tra la prima e la seconda metà delle release: ad esempio l'IV fa parte della prima metà, mentre l'FV alla seconda. In tal caso, se si fossero eliminate fin da subito le release più recenti, non sarebbe stato possibile considerare anche quei ticket. Nell'output finale, la bugginess è stata indicata con il valore "Y"/"N".

Si è deciso di svolgere il calcolo delle metriche dopo aver computato la difettosità delle classi. Le 13 metriche considerate sono:

- Size (SLOC);
- Number of Revisions (NR);
- Number of Authors (NAuth);
- LOC touched;
- LOC added;
- Max LOC added;
- Average LOC added;
- Churn;
- Max Churn;
- Average Churn;
- Change Set Size;
- Max Change Set Size;
- Average Change Set Size;

Il calcolo delle metriche avviene grazie al possesso di tutti i file presenti in una release: si sfrutta il meccanismo delle **diffEntry**, ovvero i cambiamenti che avvengono in un file tra un commit e quello precedente (il suo **parent**).

Inoltre, le metriche vengono calcolate in maniera **iterativa**, cioè di volta in volta per ogni release. Di conseguenza, metriche come NR o NAuth o le diverse LOC non hanno un valore necessariamente crescente per lo stesso file all'aumentare del numero di release, in quanto ad ogni release si azzera il conteggio. Per questa ragione, come si potrà osservare nei dataset finali, se in una release un file non è stato coinvolto in alcun commit le metriche saranno tutte nulle. Un esempio è la release 4 di BookKeeper, come si può osservare dalla figura riportata:

Release	Filename	LOC	NR	nAuth	locTouched	locAdded	max_locAdded	avg_locAdded	churn	max_churn	avg_churn	chgSetSize	max_chgSetSize	avg_chgSetSize	Buggy
4	bookkeeper-benchmark/src/main/java/org/apache/bookkeeper/benchmark/BenchBookie.j	191	0	0	0	0	0	0	0	0	0	0	0	0	N
4	bookkeeper-benchmark/src/main/java/org/apache/bookkeeper/benchmark/BenchReadThr	247	0	0	0	0	0	0	0	0	0	0	0	0	Y
4	bookkeeper-benchmark/src/main/java/org/apache/bookkeeper/benchmark/BenchThrough	399	0	0	0	0	0	0	0	0	0	0	0	0	Y
4	bookkeeper-benchmark/src/main/java/org/apache/bookkeeper/benchmark/MySqlClient.ja	121	0	0	0	0	0	0	0	0	0	0	0	0	N
4	bookkeeper-benchmark/src/main/java/org/apache/bookkeeper/benchmark/TestClient.jav	320	0	0	0	0	0	0	0	0	0	0	0	0	N
4	bookkeeper-benchmark/src/test/java/org/apache/bookkeeper/benchmark/TestBenchmark	147	0	0	0	0	0	0	0	0	0	0	0	0	N
4	bookkeeper-server/src/main/java/org/apache/bookkeeper/bookie/Bookie.java	1131	0	0	0	0	0	0	0	0	0	0	0	0	Y
4	bookkeeper-server/src/main/java/org/apache/bookkeeper/bookie/BookieBean.java	41	0	0	0	0	0	0	0	0	0	0	0	0	N
4	bookkeeper-server/src/main/java/org/apache/bookkeeper/bookie/BookieException.java	131	0	0	0	0	0	0	0	0	0	0	0	0	N
4	bookkeeper-server/src/main/java/org/apache/bookkeeper/bookie/BookieMXBean.java	28	0	0	0	0	0	0	0	0	0	0	0	0	N
4	bookkeeper-server/src/main/java/org/apache/bookkeeper/bookie/BookieShell.java	827	0	0	0	0	0	0	0	0	0	0	0	0	N
4	bookkeeper-server/src/main/java/org/apache/bookkeeper/bookie/BufferedChannel.java	170	0	0	0	0	0	0	0	0	0	0	0	0	N
4	bookkeeper-server/src/main/java/org/apache/bookkeeper/bookie/Cookie.java	237	0	0	0	0	0	0	0	0	0	0	0	0	N
4	bookkeeper-server/src/main/java/org/apache/bookkeeper/bookie/EntryLogger.java	508	0	0	0	0	0	0	0	0	0	0	0	0	N
4	bookkeeper-server/src/main/java/org/apache/bookkeeper/bookie/ExitCode.java	38	0	0	0	0	0	0	0	0	0	0	0	0	N

Un'altra osservazione si può fare riguardo al dataset di BookKeeper, è che i files hanno dei valori delle metriche abbastanza alti nelle prime release, mentre dalla quinta release molti non vengono più coinvolti nei commit, pertanto le metriche si azzerano.

I dataset si trovano in dei file chiamati **bookkeeper_buggyDataset_outputD2M2.csv** per bookkeeper e **zookeeper_buggyDataset_outputD2M2.csv** per zookeeper, presente nella repository di github sopra linkata in "\outputFinali\csv".

MILESTONE2: Machine Learning

Nella milestone2 viene utilizzato come input il dataset estrapolato dalla milestone precedente, con lo scopo di dare una risposta alla domanda su cui si basa l'intera deliverable, cioè **se e quali tecniche di Feature Selection o di Balancing aumentano l'accuratezza di un determinato tipo di classificatore**.

Pertanto, secondo la specifica, è stata fatta ogni possibile combinazione di:

Classificatori :

- Random Forest,
- Naive Bayes
- IBk

Tecniche di Feature Selection:

- No FS
- Best First

Tecniche di Balancing:

- No sampling
- OverSampling
- UnderSampling
- Smote

Quindi in totale, si ottengono 3x2x4 = **24 combinazioni**.

Per andare ad utilizzare i dati, essendo salvati in formato csv, è necessario andare a convertire i dataset in formato **arff**, in modo che sia possibile lavorarci tramite l' API di **Weka**. Prima che i dati vengano elaborati, viene impostato il **ClassIndex**, ossia la feature su cui si vogliono effettuare le stime.

Poiché le istanze di un dataset non sono indipendenti tra loro, ma vi è un **pattern temporale** che le lega, come tecnica di valutazione è stata usata quella del **Walk Forward**, in quanto si basa sull'ordinamento cronologico dei dati e lo va a preservare. Non possono essere utilizzate tecniche di validazione che non siano di tipo time-series, poiché in uno scenario come quello in esame non è ragionevole utilizzare informazioni relative al futuro per predire il presente. Per applicare il walk forward, è stato suddiviso il dataset in delle porzioni, secondo il numero di release. Per stimare la difettosità delle classi appartenenti ad una certa release, si usano le **release precedenti** ad essa come **training** per fare addestramento del modello, e quella **corrente** come **testing**, su cui effettuare la predizione. I *walks* trovati vengono salvati in una lista, in modo che, iterativamente, vengano eseguite tutte le possibili combinazioni di classificatore, tecnica di Feature Selection e tecnica di Balancing. Alcune valutazioni hanno portato a dei valori di tipo **NaN**, nel caso il denominatore in un rapporto tra due valori fosse nullo. I risultati ottenuti vengono memorizzati in una struttura dati che viene poi esportata su un file .csv, e i valori NaN vengono scartati.

I dataset si trovano in dei file chiamati **bookkeeper_WekaDataset_m2d2output.csv** per bookkeeper e **zookeeper_WekaDataset_m2d2output.csv** per zookeeper, presente nella repository di github sopra linkata in "\outputFinali\csv".

ANALISI DEI GRAFICI ED OSSERVAZIONI

Di seguito sono riportati due grafici che rappresentano come varia la percentuale di classi *defective* nelle release.

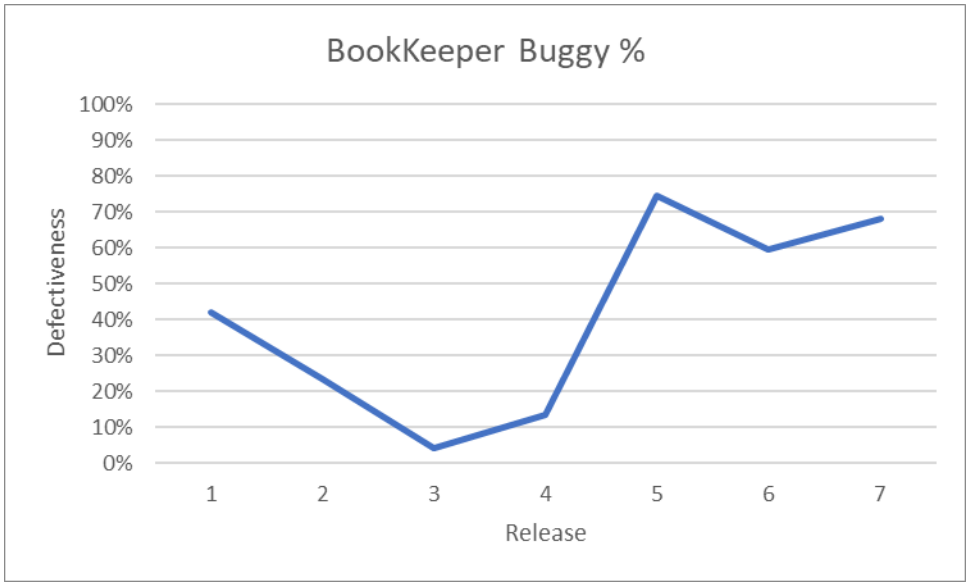


Grafico 0.1

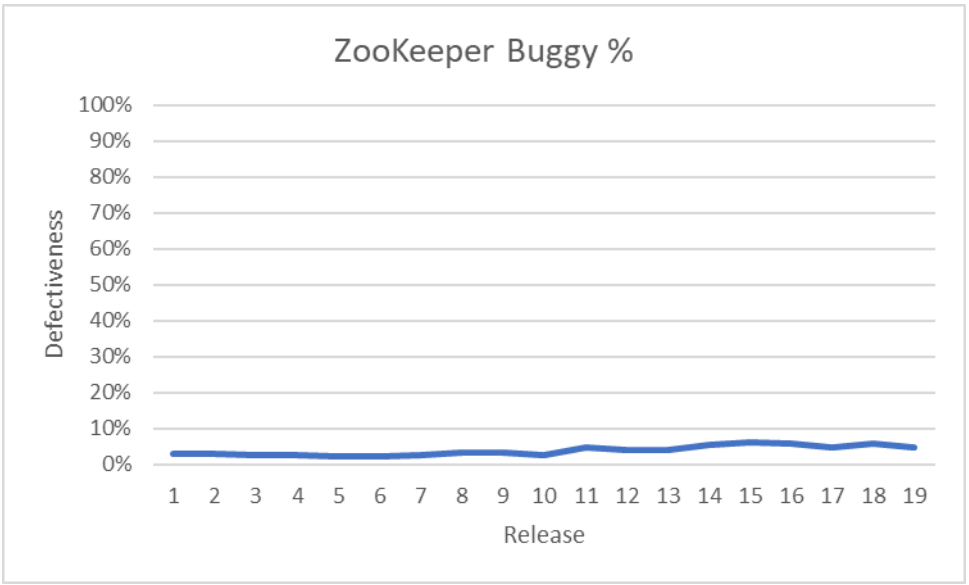


Grafico 0.2

Dai grafici 0.1 e 0.2 è immediato notare che in **BookKeeper** ci sia un andamento **irregolare** ma comunque **crescente**, e che abbia una **percentuale** di **classi defective molto elevata** rispetto a ZooKeeper: lo si può vedere anche nelle tabelle1&2, in cui su circa 3000 classi di bookkeer il **45%** risulta essere *defective*, mentre in Zookeeper, che ha circa il doppio delle classi, lo è solo il **4%**. Inoltre, in BookKeeper si ha una percentuale di classi buggy molto più alta con il passare del tempo, basta guardare le ultime tre release. **Zookeeper** invece resta sempre molto **stabile** e **costante**, con una percentuale che oscilla in un'intervallo tra il **3%** e il **6%**. E' vero che in bookkeeper ci sono molte meno release rispetto all'altro, tuttavia, nelle prime sette release in ZooKeeper si avevano già dei numeri molto più bassi. Questi grafici quindi mostrano come

ZookKeeper sia un **progetto maturo** e di qualità; nonostante questo, nel contesto dei modelli predittivi ciò rappresenta un aspetto negativo, poiché si ha un'analisi **meno accurata** delle metriche: infatti, una scarsa presenza di classi buggy non permette al classificatore di individuare facilmente quali siano le metriche che sono indice della difettosità di una classe.

BookKeeper	Count of Buggy			
Release	N	Y	Tot	Y %
1	139	101	240	42%
2	230	71	301	24%
3	420	19	439	4%
4	380	59	439	13%
5	149	440	589	75%
6	181	265	446	59%
7	209	445	654	68%
Tot	1708	1400	3108	45%

Tabella 1

ZooKeeper	Count of Buggy			
Release	N	Y	Tot	Y %
1	183	6	189	3%
2	186	6	192	3%
3	222	6	228	3%
4	222	6	228	3%
5	272	6	278	2%
6	275	6	281	2%
7	275	8	283	3%
8	335	12	347	3%
9	342	12	354	3%
10	546	14	560	3%
11	337	17	354	5%
12	342	15	357	4%
13	342	15	357	4%
14	369	22	391	6%
15	368	24	392	6%
16	382	24	406	6%
17	343	17	360	5%
18	392	24	416	6%
19	346	17	363	5%
Tot	6079	257	6336	4%

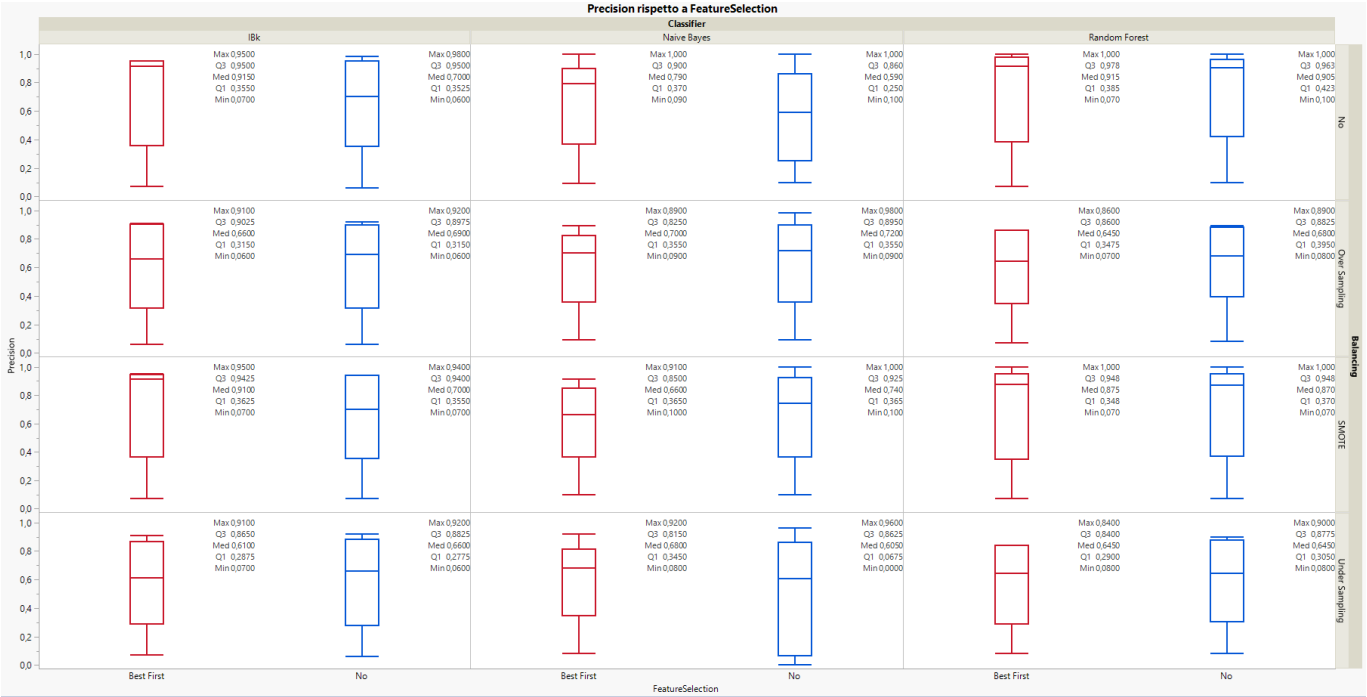
Tabella 2

Da qui in poi viene effettuato uno studio statistico mirato a trovare quale sia il classificatore migliore e a vedere se la sua performance aumenta se in combinazione con Feature Selection e Balancing. Nei grafici che seguono viene esaminata una metrica d'accuratezza per volta in modo da facilitare il confronto fra tutte le combinazioni.

Viene analizzato prima BookKeeper e poi ZooKeeper.

BookKeeper

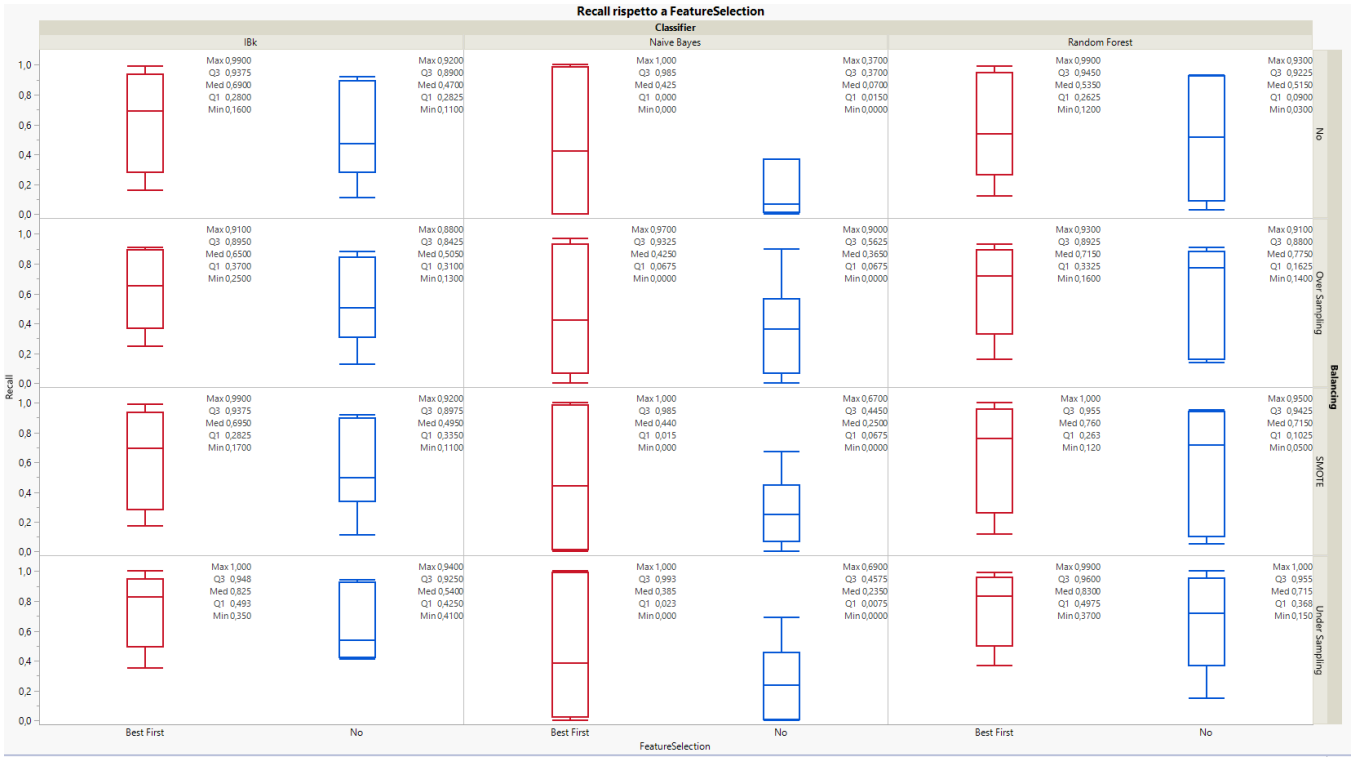
Grafico1.1: Precision rispetto a Feature Selection



Dal grafico1.1 si può dedurre che per ognuna delle combinazioni si ha una varianza elevata: di conseguenza, per ogni classificatore non vi è una combinazione di tecniche di Feature Selection/Balancing che risulta essere nettamente migliore rispetto alle altre. Per quanto riguarda i quantili di primo e secondo ordine, si può osservare che hanno dei valori molto stabili e simili fra loro. Al contrario, la mediana presenta dei valori differenti tra le varie combinazioni. Per **IBk**, la combinazione migliore risulta essere **Best First – No Sampling**, che pur avendo un'alta varianza,

ha una mediana molto alta e il quantile di terzo ordine che coincide con il massimo, mente il minimo è pari a quello delle altre combinazioni di IBk. Per **Naive Bayes** invece, risulta essere **Over Sampling – Best First,** infatti anche se ci sono combinazioni con valori leggermente migliori (mediana, massimo e Q3), questa combinazione presenta la varianza più bassa pur mantenendo dei valori buoni. Infine, **random forest** ha resa migliore **senza utilizzo di teniche né di balancing né di feature selection**, in quanto presenta i valori del minimo e del quantile di primo ordine più elevati, e una mediana quasi prossima al quantile di terzo ordine, che si avvicinano di molto al massimo, pari a 1.

Grafico 1.2: Recall rispetto a Feature Selection



Al contrario della precision, la *recall* mostra che le combinazioni presentano una varianza abbastanza elevata. Salta subito all'occhio che in questo caso Naive Bayes è il classificatore peggiore, in quanto tutte le combinazioni sono poco bilanciate: quelle con varianza bassa hanno dei valori anch'essi bassi, che corrispondono a No Feature Selection; al contrario, quelle con un

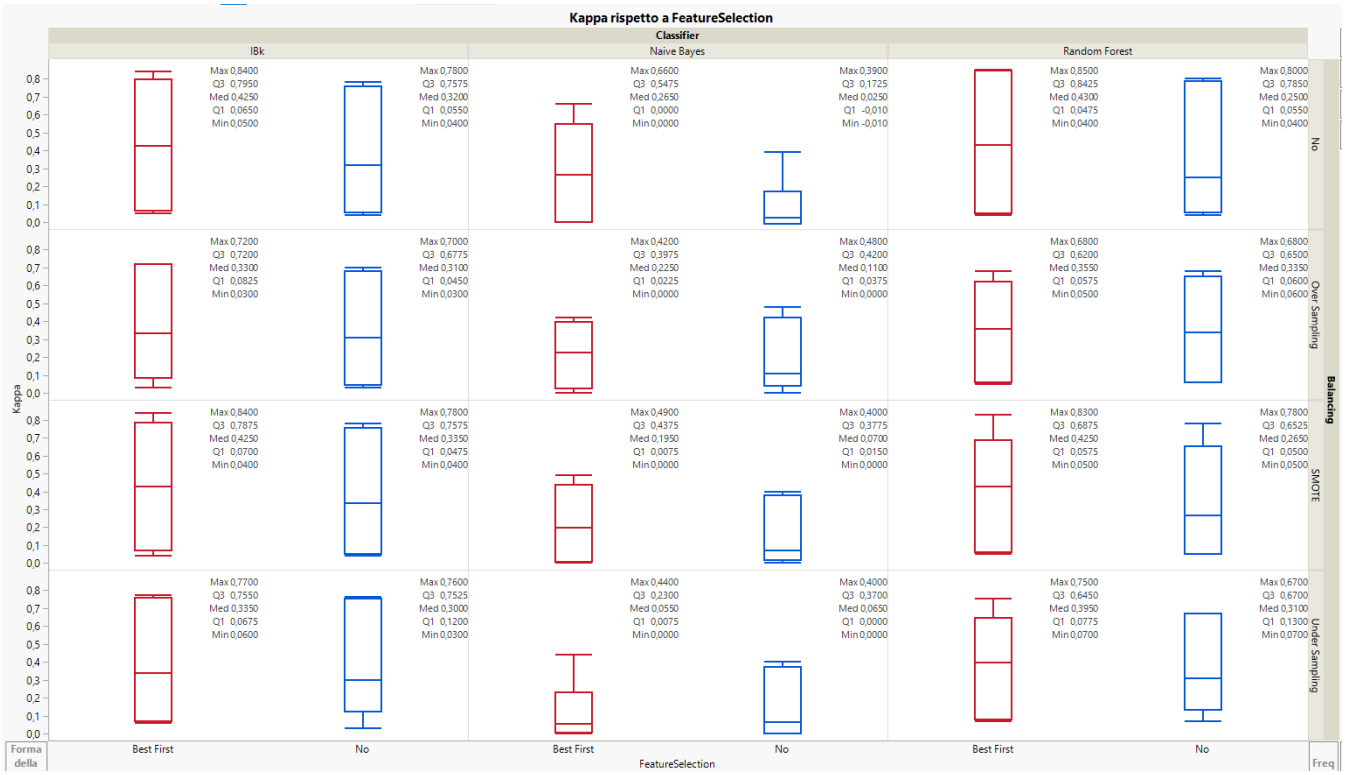
massimo elevato presentano una varianza molto alta ed una mediana bassa, che corrispondono a Feature Selection con Best First. Tra i restanti classificatori Random Forest e IBk, si può notare che in entrambi i casi sono migliori in combinazione con Best First. In particolare, la combinazione migliore risulta essere **Random Forest – Best First – Under Sampling**, poiché è una di quelle aventi varianza più bassa, una mediana all'83% ed il valore minimo più elevato. Tuttavia, si possono trovare dei valori analoghi anche in IBk-Best First-Under Sampling.

Grafico 1.3: AUC rispetto a Feature Selection

Contrariamente alla *precision* e alla *recall*, in tale grafico è immediato osservare che Naive Bayes è il classificatore migliore, dal momento che presenta le combinazioni con varianza più bassa: la varianza minima si ha senza l'utilizzo di alcuna tecnica di Feature Selection. Le quattro coppie Naive Bayes-No Feature Selection presentano delle caratteristiche molto simili tra loro; poiché hanno tutte lo stesso valore massimo, un criterio considerevole per la scelta può essere la mediana. Di

conseguenza, la combinazione con la tecnica di balancing SMOTE risulta essere vincente, quindi **Naive Bayes-No Feature Selection-SMOTE**.

Grafico1.4: Kappa rispetto a Feature Selection

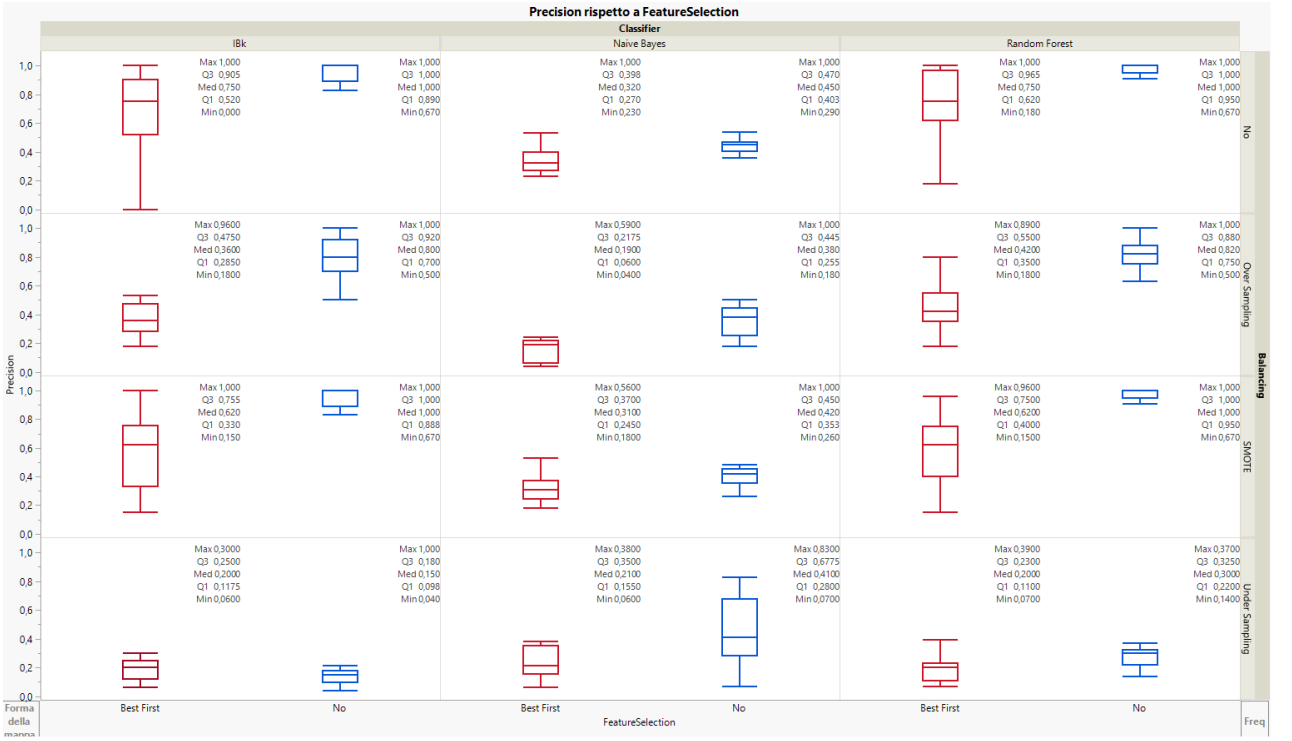


La Kappa, in machine learning, è utilizzata per comparare dei classificatori ed esprime quanto un classificatore si discosta da un classificatore "dummy" (random): il fatto che in IBk e Random Forest non abbiano dei valori bassi, indica che questo discostamento è significativo. Inoltre si può vedere che il grafico è molto simile a quello della Recall (grafico1.2), in quanto è immediato osservare che Naive Bayes può essere scartato e che anche in questo caso Random Forest e IBk presentano delle combinazioni molto simili. Tuttavia, la scelta del miglior classificatore si può direzionare verso Random Forest- No Feature Selection, in quanto presentano le varianze più basse. La combinazione migliore risulta dunque essere **Random Forest- No FS- Over Sampling**, in quanto ha un quantile del terzo ordine molto vicino al massimo e una mediana che, sebbene sia al 33%, resta comunque fra le più alte.

Grafico1.5: Precision, Recall, AUC e Kappa

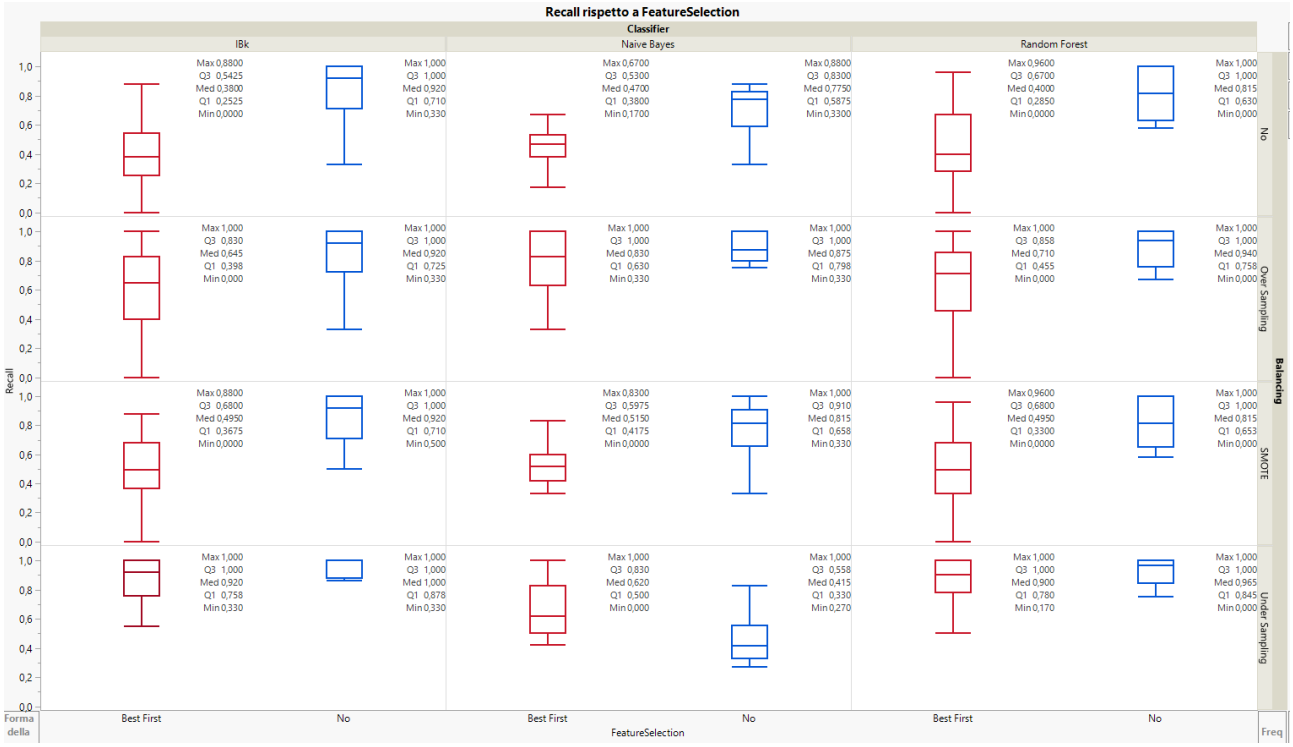
Da questa visione complessiva si può notare che AUC sia migliore rispetto alle altre metriche d'accuratezza grazie alla sua bassa varianza, contrariamente a ciò che accade per Kappa, con valori molto bassi e varianze per lo più elevate. Anche la varianza della Precision e della Recall risulta essere spesso elevata, tuttavia presentano dei valori alti al confronto con Kappa. Osservando il grafico si può notare facilmente che IBk presenta i risultati con varianza più elevata, pertanto è meglio escluderlo dalla scelta del classificatore. Tra i due rimanenti, è vero che Naive Bayes presenta AUC con variazione più bassa in assoluto, tuttavia la recall ha valori o molto alti con varianza altrettanto alta, oppure, al contrario, varianza bassa con valori anch'essi bassi. Random Forest invece presenta delle combinazioni più bilanciate: i valori sono buoni, anche se non ottimi, ma con il vantaggio di avere una varianza nella Recall leggermente inferiore rispetto a quella presente in Naive Bayes. Pertanto si può concludere che uno dei classificatori migliori sia **Random Forest-Best First-SMOTE**, poiché anche se le varianza sono elevate, lo è anche la mediana.

Grafico2.1: Precision rispetto a Feature Selection



La precision presenta dei valori elevati e osservano il grafico è immediato vedere che il classificatore migliore è Random Forest, in quanto si hanno dei valori di massimo, varianza e terzo quantile assai elevati assieme ad una variazione molto bassa, che è indice di consistenza dei risultati ottenuti. In particolar modo, si può notare che le combinazioni migliori prevedono entrambe che non sia utilizzata alcuna tecnica di Feature Selection, e i valori riportati sia in No Balancing che in SMOTE sono identici. Di conseguenza si può concludere con sicurezza che le combinazioni migliori siano **Random Forest – No FS – No Balancing** e **Random Forest - No FS – SMOTE**.

Grafico2.2: Recall rispetto a Feature Selection

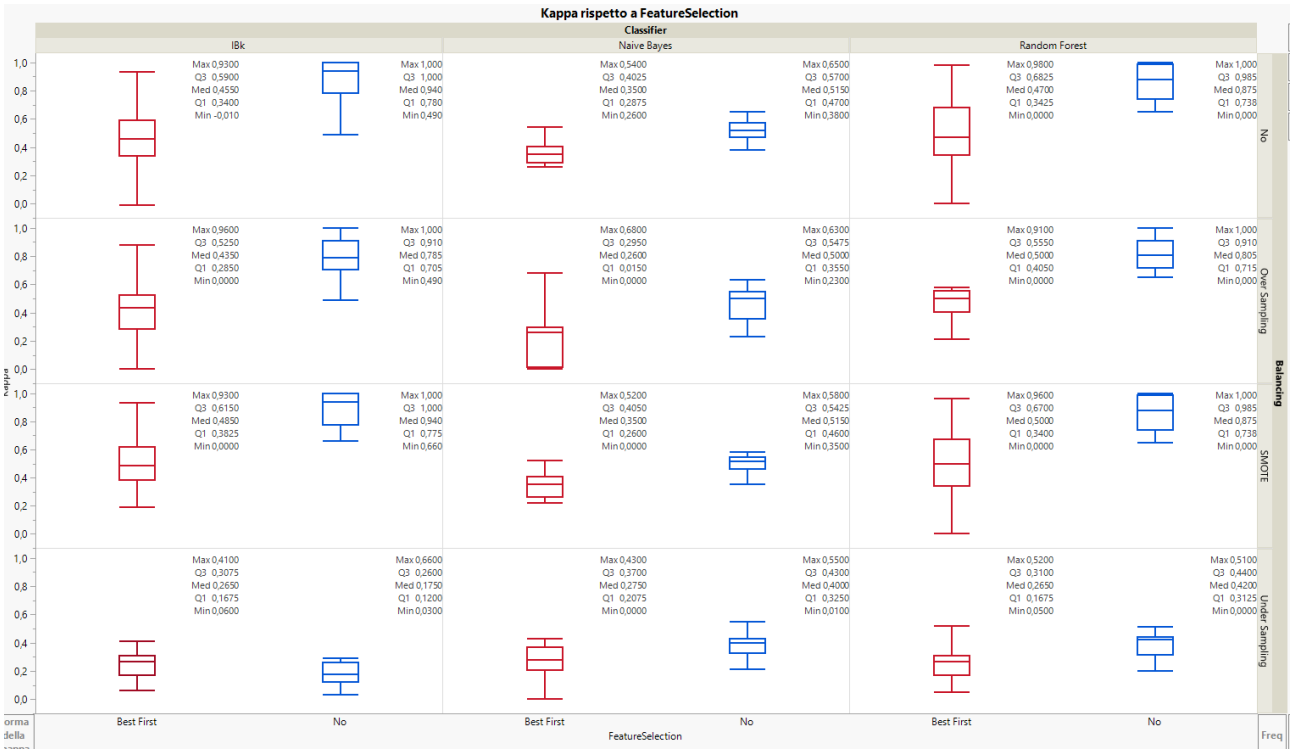


Analogamente alla Precision, anche la Recall presenta un grafico in cui salta subito all'occhio quale sia la combinazione da scegliere, ovvero **IBk – No FS – Under Sampling**, poiché si ha una varianza molto bassa e una mediana, terzo quantile e massimo che coincidono al 100%.

Grafico2.3: AUC rispetto a Feature Selection

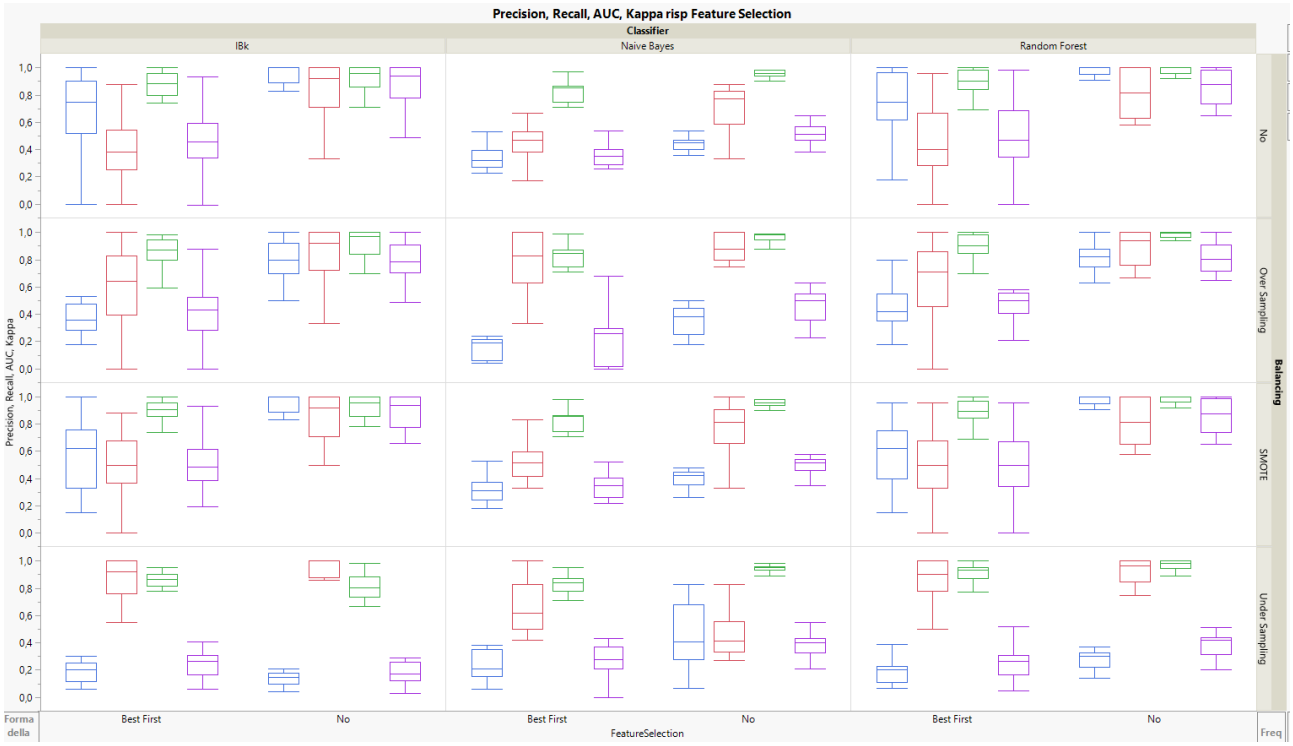
Dal grafico si vede immediatamente che IBk è il classificatore peggiore, pertanto la scelta deve necessariamente ricadere su uno degli altri due. Entrambi i classificatori presentano delle combinazioni molto buone, con varianza bassa e valori elevati in corrispondenza di NO Feature Selection, come la mediana che spesso è al 100%, coincidente con il massimo. Le due combinazioni migliori quindi sono **Naive Bayes – No Feature Selection – No Balancing** e sono **Naive Bayes – No Feature Selection – SMOTE**, in quanto entrambe presentano dei valori molto positivi, tra l'altro identici, come un minimo pari al 90%.

Grafico2.4: Kappa rispetto a Feature Selection



Al contrario della precision, recall, e AUC, in Kappa la scelta della combinazione migliore è meno inutitiva. Nonostante ciò, come nei casi precedenti, anche qui si vede facilmente che le combinazioni migliori corrispondono al mancato utilizzo di tecniche di Feature Selection. Si vuole sempre compiere una scelta che presenti una varianza il più bassa possibile, cercando di avere comunque dei valori buoni; pertanto è da escludere Random Forest, che presenta fra tutti la varianza più elevata poiché le combinazioni vanno quasi tutte da un minimo al 0% a un massimo di 100%. La combinazione migliore risulta essere **IBk – No FS – SMOTE**, poiché si ha una varianza bassa (anche se non minima, ma in casi in cui è minore si ha un massimo circa al 50%, che è come se la stima fosse stata effettuata in modo casuale) ed una mediana al 94%.

Grafico2.5: Precision, Recall, AUC e Kappa



Come prevedibile dalle analisi effettuate sulle singole metriche di accuratezza, è evidente che il classificatore migliore in assoluto sia **Random Forest** senza utilizzare alcuna tecnica di Feature Selection, in linea con i risultati riportati precedentemente. Per quanto riguarda eventuali tecniche di Balancing, si potrebbe sia non farne uso, sia utilizzare SMOTE: dal grafico difatti si può vedere come i "boxes" siano equivalenti. Si può quindi concludere che le due combinazioni migliori risultano essere:

- **Random Forest – No Feature Selection – No Balancing**

- **Random Forest – No Feature Selection - SMOTE**

Da cui è interessante notare che le tecniche di Feature Selection non contribuiscano ad un miglioramento, nonostante prevengano l'Overfitting, in cui si ha una bassa accuratezza a causa di uno scarso numero di esempi su cui effettuare addestramento, che è proprio il caso di ZooKeeper.

Conclusioni

BookKeeper e ZooKeeper, sebbene abbiano un'architettura progettuale molto simile, presentano attributi di progetto molto differenti tra loro, a partire dal **numero di release**, dove in ZookKeper sono circa il quintuplo, alla **quantità di classi defective** presenti. Tuttavia, si possono vedere sia i pro che i contro di queste differenze: innanzitutto, un numero di release alto comporta l'utilizzo di molti più walk rispetto a BookKeeper, e quindi di avere un **campione** molto più ampio su cui effettuare l'analisi; tuttavia lo stesso BookKeeper possiede, a parità di versioni, molte più classi buggy, che permettono al classificatore di effettuare una **stima** migliore.

Nell'analisi di BookKeeper non si è trovato un **pattern** tra i diversi scenari esaminati, difatti il grafico finale (*Grafico1.5*) non presenta dei risultati decisivi e non vi è una combinazione classificatore-tecniche che risulti nettamente migliore alle altre. Al contrario, come si può osservare nel *Grafico2.5*, in ZooKeeper si è vista una **linearità** e continuità man mano che si procedeva con l'analisi dei dati e questo ha comportato anche dei risultati molto esaustivi e ben definiti rispetto a BookKeeper.