Ingegneria Del Sowtware 2

-

Process Control Chart & Predizione Difettosità Utilizzando Machine Learning

Progetto Parte Falessi

Ezio Emanuele Ditella - 026766
Università Di Roma Tor Vergata

PROCESS CONTROL CHART

***Abstract*—Nel seguente report si discuterà del processo di controllo statistico noto come process control chart, su come utilizzarlo per distinguere processi sotto controllo e non, quali considerazioni trarre dai risultati di tale strumento, gli errori tipici da non commettere e la sua applicazione su un caso di studio.**

I. CONTESTO E SCOPO

L'intento del process control chart è quello di graficare l'andamento di una *statistica* nel corso del tempo, e vedere se i valori da essa assunta sono compresi tra un dato limite *superiore* ed *inferiore*. L'**intervallo di controllo superiore** si ottiene aggiungendo dalla media il triplo della deviazione stadard della statistica, mentre L'**intervallo di controllo inferiore** è pari alla media meno il triplo della deviazione standard. Se la statistica in oggetto di studio non ha valori che *cadono fuori tali limiti*, allora il processo il cui stato dipende da tale statistica è **stabile**, sotto controllo, altrimenti il processo è **instabile**. Nel caso in cui il numero dei campioni della statistica è elevato (in genere maggiore di 40), ed i campioni son *indipendenti*, è possibile effettuare un'analisi più accurata, introducendo 3 zone di controllo:

- La zona A data dall'intervallo:
  $[\mu$ -3σ, μ -2$\sigma]$ U $[\mu$+2σ, μ+3$\sigma]$
- La zona B data dall'intervallo:
  $[\mu$ -2σ, μ -$\sigma]$ U $[\mu$+σ, μ+2$\sigma]$
- Le zone C- e C+, risperrivamente:
  $[\mu$ -σ, $\mu]$, $[\mu$, μ +$\sigma]$

Date tali zone, un processo è detto **fuori controllo** se: due punti su tre cadono nella zona A, quattro punti su cinque cadono nelle zone A o B, otto punti consecutivi cadono nelle zone C+ o C-, sei punti consecutivi sono in ordine crescente o decrescente, quattordici punti consecutivi si alternano a zig-zag (crescente-decrescente o viceversa).

Nel caso di studio di questo report utilizzeremo come statistica il *numero di difetti*, o bug, che affliggono un'applicazione software, in particolare, l'applicazione **Apache Daffodil**. L'unità del periodo temporale d'osserazione sarà il *mese*. Il processo che si vuole monitorare, è il processo di sviluppo del software *Daffodil*, ovvero il processo in cui il numero di difetti può essere utilizzato per descrivere il suo stato. La statistica considerata genererà dei campioni **non indipendenti**[1] tra di loro, e ciò non ci consentirà di analizzare il process control chart secondo le modalità *accurate* precedentemente descritte, ma utilizzando solo i limiti di controllo superiori ed inferiore. Il **fine ultimo** di tale analisi è osservare, qualora presenti, i periodi temporali di sviluppo software in cui ci sono state delle *anomalie*, cercando di capire cosa ha portato il manifestarsi delle stesse per intraprendere le giuste strategie correttive. Un processo software sotto controllo è proprio di organizzazioni con livelli di maturità elevati[2], e comprendere **perché** un processo software è instabile è volto al miglioramento dei processi stessi al fine di incrementare la propria produttività ed efficienza, e dunque, guadagno.

II. PROGETTAZIONE

Il numero di difetti mensili del l'applicazione *Daffodil* sono stati calcolati nel metodo *getBugsPerMonth* della classe *GitInteractor*. Tale metodo è in grado di *combinare* le informazioni memorizzate nella piattaforma **JIRA** e quelli mantenuti da **GitHub**. In particolare da JIRA sono stati estratti tutti i tickets di tipo *BUG* con stato *CLOSED* o *RESOLVED*. Da GitHub invece è stata estratta la *data* associata all'*ultimo commit* avente nei commenti l'ID del ticket prelevato su JIRA.
Durante l'interazione con la piattaforma GitHub è stato mantenuto un dizionario avente come chiave la coppia mensilità-anno e come valore il numero di commit, secondo i requisiti sopra descritti. Come ultima operazione è stato necessario controllare che *anche* i mesi in cui non è stato riscontrato nessuno di questi commit, devono essere presenti

[1] Il numero di bug in un certo periodo temporale può dipendere dal numero di bug riscontrati nei periodi precedenti

[2] Secondo il CMMI (Capability Measurement Model) dal livello 4 in poi

nel dizionario, se pur associati con un valore pari a zero. Questo perché *nel calcolo della media e varianza* i valori nulli **non** devono essere trascurati. Il caso contrario porterebbe ad un valore errato dei limiti superiore e inferiore di controllo.
L'interazione con JIRA è stata implementata sfruttando la sua REST API, in modo da ottenere una risposta in JSON dei tickets con le caratteristiche di sopra descritte. Per estrarre le informazioni necessarie su GitHub si è deciso di utilizzare la sua *REST API*, la quale se pur limitante nel numero di richieste[3], è molto completa e dispone di tutti gli strumenti necessari per arrivare allo scopo descritto. L'URL (API) utilizzato è il seguente: ***https://api.github.com/search/commits?q=repo:apache/"+ gitProjName+"+"%s"+sort:committer-date.***
L'ultimo parametro è servito per ordinare i risultati per data decrescente, dunque il commit più vecchio è il primo di tutti i commit contenuti nella risposta.
Dato che il **numero** di commit avente per commento l'id del ticket scritto secondo la convenzione (PROJNAME-1234) è risultato **molto basso**, è stato implementato un meccanismo che effettua una seconda ricerca nel caso nessun commit sia stato trovato con quel partiolare ticket id. Per la seconda ricerca è usato come identificativo del ticket solo la sua parte numerica preceduta da un trattino (-1234). Utilizzando questa strategia si è stati in grado di aumentare il numero totale dei commit trovati.
Una volta raccolti tali dati, sono stati scritti su un file in formato csv, dal quale è stato possibile estrarre i grafici esposti in Figure 1, 2.

III. Analisi Dei Risultati

Dai dati raccolti, sono stati calcolati i seguanti valori

- Media di difetti: **7.38**
- Deviazione standard: **7.26**
- Limite di controllo superiore: $\lceil 29.16 \rceil = \mathbf{30}$
- Limite di controllo inferiore: **-14.39**[4]

Da tali valori possiamo affermare che qualora il numero di bug in una mensilità dovesse essere maggiore o uguale a 30, il processo software sarebbe instabile. Dai grafici è possibile vedere che ciò è successo per 2 volte, in particolare: ad aprile 2013 con 34 bug; gennaio 2015 con 37 bug.
Per vedere il **motivo** per cui in tali mensilità si è andati oltre il limite di controllo superiore bisognerebbe analizzare lo stato del team di sviluppo in termini di: numero di autori differenti che hanno lavorato sul software in quella mensilità, numero di commit effettuati, numero di ticket emessi, rapporto tra ticket e commit ecc.
E' interessante notare come i valori fuori dal limite di controllo superiore sono *preceduti* da una situazione *crescente* di numero di bug, questo prova se pur non in modo rigoroso l'esistenza della **dipendenza** dei campioni rispetto i suoi valori passati.

[3] 10 richieste al minuto. Per aumentarle fino a 30 è stato aggiunto nell'header un token associato ad un account GitHub. Per evitare che il programma terminasse bruscamente a causa delle limitazioni delle velocità è stato inserito un meccanismo in grado di limitare il numero di richieste al minuto.

[4] Essendo il valor minimo dei fifetti mensili zero, questo limite di controllo inferiore non è applicabile al caso di studio

IV. Conclusioni

Come si può osservare dal Control Chart del processo di sviluppo del software *Daffodil*, negli ultimi 5 anni (dal 2015 al 2020) non si sono verificati casi al di **sopra** del limite superiore di controllo, questo può essere un sintomo di come il processo si sia *stabilizzato* proprio negli ultimi anni, dopo un periodo iniziale che potremmo definire di assestamento.
Infatti l'intero processo di sviluppo dell'applicazione ha una durata di 8 anni, e poter affermare tale processo sia stabile sarebbe alquanto utopico. Difatti anche se la fondazione Apache si occupa dello sviluppo software dagli anni 2000, l'applicazione *Daffodil* è open source, permettendo a chiunque autorizzato a contribuire nel suo sviluppo. Ciò significa che contrariamente ad aziende ed organizzazioni private, che possiedono dei processi ben definiti e sono dotati di personale istruito a produrre software seguendo schemi ben definiti, il processo di sviluppo software di apache proprio per sua natura è più *malleabile*.
Ciò da un lato favorisce un continuo susseguirsi di apprendimento di tecniche ed idee da persone qualificate provenienti da tutto il mondo, dall'altro rende più difficile fissare degli standard ben precisi.

PREDIZIONE DIFETTOSITÀ CON ML

***Abstract*—In questo report si andrà a discutere di alcuni concetti e strumenti usati nel software analysis il cui obiettivo sarà, mediante l'applicazione di algoritmi di Machine Learning, predirre la difettosità delle classi di applicazioni software. Questo faciliterà la fase del Software Quality Assurance consentendo di concentrare l'effert del testing sul solo sottoinsieme di classi per le quali è stato predetto esserci qualche bug. La fase della costruzione di un modello predittivo sarà preceduto da una lunga e meticolosa procedura di raccolta e raffinamento di dati. Successivamente si discuterà di tecniche di validazione e metriche usate per quantificare la bontà delle predizioni fatte da modello costruito. Verranno inoltre applicate tecniche di sampling e feature selection, al fine di aumentare i valori di tali metriche.**

I. CONTESTO E SCOPO

Il passare degli ultimi decenni è stato spalleggiato dal progresso tecnologico che ha visto come protagonista l'avvento di nuovi strumenti, idee, piattaforme ecc. il cui corretto funzionamento dipende soprattuto dal **software**. Lo stesso sviluppo tecnologico ha inoltre reso alla portata di più o meno tutti la possibilità di sviluppare del software per trarne **profitto**.
Proprio per questo motivo il mercato del software è incredibilmente vasto, ma proficuo solo per coloro che sono in grado di sviluppare del software di migliore qualità. Andando più nello specifico tanto maggiore è la **qualità esterna** di un'applicazione, tanto meno comportamenti insapettati (*bug, difetti*) saranno percepiti dall'utente finale, e dunque manggiore sarà il gradimento della clientela e maggiore il profitto dell'azienda che ha costruito quel software.
In questo contesto è dunque di importanza cruciale concentrare una parte rilevante di enegie (in termini di budget), alla fase di *testing*[5]. Per aumentare l'**efficacia** di questa fase entra in gioco il **Software Analysis**. A questo proposito, nell'ambito di tale report, si discuterà del processo impiegato per la costruzione di un *Modello Predittivo* in grado di predirre le classi *difettose* di un'applicazione, consentendoci di ottimizzare tempo e denaro destinato alla fase di testing.
I modelli predittivi sviluppati saranno applicati alle applicazioni open-source di *Bookkeeper* e *Syncope*, tuttavia il processo utilizzato ed implementato può essere esteso ad una qualsiasi applicazione[6] a condizione che

- L'applicazione sia Open-Source, o che sia permesso accedere al suo sorgente
- L'applicazione usi **Jira** come Issue tracking system
- Il repository dell'applicazione sia pubblico ed accessibile su **GitHub**

Il processo di sviluppo del modello predittivo può essere riassunto nei seguenti passi

1) Raccolta Dati
2) Validazione del dataset costruito
3) Miglioramento dell'efficacia del predittore mediante l'ausilio di Sampling e Feature Selection

[5]il testing in realtà è solo una piccola parte dell'insieme di pratiche e procedure definite dal Software Quality Assurance, il cui fine ultimo è verificare proprio la qualità dell'applicazione

[6]Come PoC lo stesso processo di sviluppo di un modello predittivo è stato applicato alle applicazioni **Avro** e **Falcon**

II. RACCOLTA DATI

Gli algoritmi basati su **Machine Learning** costruiscono dei modelli predittivi che sono in grado di prendere una decisione in modo che la procedura di produzione di questa stessa decisione non sia stata esplicitamente programmata. In particolare la decisione presa da questi algoritmi è un possibile valore di un **attributo** di una certa **entità**. La decisione verrà presa basandosi sull'**esperienza pregressa**, acquisita analizzando molteplici **istanze** dell'entità sotto studio.
Il primo passo verso la costruzione di un modello predittivo consiste quindi nella creazione di un **dataset** contenente informazioni su molte istanze di una certa entità sulla quale poi si potranno fare delle predizioni.
Nel nostro caso di studio l'entità in analisi è del *codice sorgente* o semplicemente un *file* implementato usando un certo linguaggio di programmazione. Nel caso di *Bookkeeper* e *Syncope*, le *istanze* dei rispettivi dataset saranno delle Classi Java.
Gli attributi dell'entità sotto studio sono dunque delle **metriche software**, ed è stato deciso di selezionare le seguenti

1) Line Of Codes (LOC)
2) LOC touched
3) NR, numero di revisioni (commit)
4) NFix, numebro di bux risolti
5) NAuth, somma di diversi autori che hanno apportato modifiche al file
6) LOC added, somma delle linee di codice aggiunte (rispetto le revisioni) in una release
7) MAX LOC added
8) AVG LOC Added
9) Churn, somma delle linee aggiunte meno le linee rimosse in una release (rispetto i commit)
10) MAX Churn
11) AVG Churn
12) ChgSetSize, numero di file committati assieme al file
13) MAX ChgSetSize
14) AVG ChgSetSize
15) Difettosità

La fase sicuramente più insidiosa e al tempo stesso cruciale per la riuscita del modello predittivo è stata la stima della **difettosità**. Per riuscire in tale operazione si è sfruttato il seguente concetto: convenzionalmente nel momento in cui un bug dovvesse essere risolto e quindi chiuso, il relativo ticket (indipendentemente dall'issue tracking system usato) dovrebbe contenere un sottoinsieme di versioni (chiamate affected versions) durante le quali il bug è persistito. In particolare dalla teoria, le **affected versions** (AV), sono tutte quelle versioni che vanno da quella in cui il bug è stato immesso nell'applicazione, **Injected Version** (IV), fiano a quella (esclusa) in cui il bug è stato risolto, **Fixed Version** (FV). A questo punto possiamo dire che un file è marcato come difettoso nelle versioni definite dall'AV.
Purtroppo raramente le AV sono eplicitamente definite in un ticket, e per questo motivo bisogna ricorrere ad altre metodologie per stimare le AV. Nell'ambito di questo progetto

si è utilizzato il metodo **Proportion**. L'idea struttata da questo metodo è che la distanza tra la FV e la **Opening Version** (OV, versione in cui si è identificato il bug ed aperto il relativo ticket), è uguale alla distanza tra OV e IV. Sfruttando tale ragionamento è possibile stimare il pezzo mancante per la stima delle affected version, ovvero l'IV, nel seguente modo: $IV = FV - (FV - OV) * P$ con $P = (FV - IV)/(FV - OV)$ dove il parametro P è stato calcolato usando il metodo **Increment**, ovvero sulla media dei bug[7] associati a ticket contenenti le AV.
Nell'ambito di questo progetto l'implementazione del metodo proportion per la stima delle AV e quindi della difettosità di un file è stato fatto seguendo i passi qui di seguito

1) Inizializzazione del parametro P mediante:
   a) Raccolta delle informazioni del primo 10%[8] dei ticket di tipo bug chiusi o risolti dell'applicazione
   b) Per ogni ticket al passo a) se contiene le affected version, è estratta l'IV come la più vecchia versione presente tra le AV. Sono poi estratte l'OV e FV, e aggiornato il parametro P. Se le AV non sono presenti il ticket è saltato
2) Estrazione tramite API di Jira degli ID dei ticket associati a bug chiusi e/o risolti
3) Restringimento dell'insieme di ticket selezionando solo quelli la cui data di rilascio rientra nella prima metà di 'vita' dell'applicazione
4) Salvataggio dei nomi di tutte le classi[9] contenuti nei commit associati ai ticket al punto precedente
5) Calcolo delle affected versions associate ad ogni ticket, usando proportion laddove necessario
6) Creazione di una classe di supporto, la *AnalyzedClass*, per l'associazione delle AVs alle classi trovate al punto 4

Al termine di questa procedura è stata ottenuta una lista di *AnalyzedClass* e per ognuno di essi la lista delle versioni in cui è stato difettoso. La AnalyzecClass è proprio la classe che rappresenta l'entità che si vuole studiare con il modello predittivo. E' composta da tanti attributi quante le metriche software descritte precedentemente.
Il calcolo del valore delle metriche di ogni file, e quindi della singola istanza della classe *AnalyzedClass* è stato calcolato con la seguente procedura

1) Per ogni release dell'applicazione, è stato estratto da Jira l'intervallo temporale in cui la release ha vissuto
2) Usando le API di GitHub sono stati salvati come oggetti di tipo *AnalyzedClass* tutti i file presenti nei commit nel periodo temporale indicato precedentemente
3) Per ognino di questi file sono state calcolate le metriche software
4) E' stata salvata una lista contenente tutti gli oggetti di tipo AnalyzedClass per ogni release, fino ad arrivare alla release coincidente con la metà del ciclo di vita dell'applicazione, per evitare il fenomeno delle **Snoring** classes
5) I metadati salvati per ogni file di ogni release sono stati trascritti in un database in formato csv

Uno snapshot del risultato finale (dataset) prodotto da tale procedura è nelle Figure 4 e 5. L'operazione di estrazione e calcolo di 'file' difettosi è stata implementata nel metodo *getBuggyFiles* della classe *Buggy*. Mentre la raccolta delle metriche di ciascun file di ogni release di un applicazione è innescata dal metodo *getFinalTable* della classe *GitFilesAttributesFinder*.

III. Feature Selection

La **Feature Selection** può essere definita come una tecnica di raffinamento di un dataset usato in algotirmi di ML. L'idea centrale è qualla di evitare di distrubare il predittore fornendogli attributi che **non sono correlati** con quello che si vuole stimare. In genere un predittore sarà più accurato per un certo sottoinsieme degli attributi di partenza. In particolare fissata l'entità che si sta studiando ed i suoi attributi, tale sottoinsieme varierà a seconda del classificatore usato. Ciò significa che il processo di applicazione di feature selection è non banale e può essere computazionalmente oneroso.
Nell'ambito di questo progetto si è seguito l'approccio basato su **filtro**, vale a dire che sono stati usati dei modelli statistici che, dal dateset, sono in grado di stimare la correlazione tra gli attributi. Il fine ultimo è quello di selezionarne un sottoinsieme tale per cui, detto A l'attributo da stimare e A_i gli altri, la correlazione tra gli A_i sia bassa, mentre la correlazione tra A e gli A_i sia elevata.
Questo approccio è intrinsecamente più articolato e complesso, ma computazionalmente meno oneroso degli approcci basati su **wrapper**. Secondo questi ultimi il predittore è istruito iterativamente su ogni combinazione di attributi, ed è selezionato il sottoinsieme che ha portato ad una bontà di predizione migliore.
Un altro vantaggio implicito offerto dalla Feature Selection è che con il restringimento dell'insieme di attributi, il dataset diviene più leggero in termini di metadati che dovranno poi essere processati. Ciò si traduce in un costo computazionale minore nell'eseguire il processo di apprendimento di modelli predittivi.
Nell'implemetazione l'applicazione di un filtro per la feature selection si è tradotto nell'utilizzo di API di Weka ai training set e test set, prima di inizializzare il processo di apprendimento.

IV. Sampling

La problematica che si vuole cercare di aggirare con il **Sampling**, è data dal fatto che il training set fornito ad un algoritmo di ML potrebbe **NON** essere **omogeneo** rispetto l'attributo per il quale di vuole fare una predizione. Ciò significa che, fissato l'attributo di valori possibili a e b, potrebbero esistere molteplici istanze ad evere un valore pari a b e molte poche ad avere valore pari ad a. Cio porta alla

[7] il 10% dei bug contenuti nella prima metà (in termini di release) dell'applicazione

[8] Parametro contigurabile, in base al progetto

[9] più nello specifico, di tutti i file terminanti con una certa estensione

costruzione di modelli predittivi molto accurati per la stima di **solo uno** dei valori dell'attributo stimato, ovvero, quello maggiornmente presente.
Nell'ambito di tale progetto difatti i modelli predittivi sono stati molto efficienti nel riconoscimento di una classe NON difettiva[10].
L'**obiettivo** del sampling è dunque avere un training set con una distribuzione omogenea di istanze rispetto un certo attributo. Nel progetto sono stati applicati i seguenti metodi di Sampling

- Nessuno
- **Oversampling**: *estenzione* della cardinalità delle istanze con valore dell'attributo monoritario, mediante la ripetizione nel trainig set di istanze di questo tipo
- **Undersampling**: carinalità delle istanze con valore dell'attributo maggioritario *ristretta* alla cardinalità delle istanze con valore dell'attributo attributo minoritario
- **SMOTE**: estensione della cardinalità delle istanze con valore dell'attributo minoritario mediante la creazione di entità *sintetiche*, con valore dell'attributo pari a quello minoritario

Anche in questo caso per applicare in pratica questi metodi, sono state usate delle semplici API messe a disposizione da Weka.

V. Validazione

Dopo aver costruito il dataset, raffinato con Feature Selection e migliorato con Sampling, è necessario *validare* i modelli predittivi risultanti, ovvero valutare la bontà della predizione fatta. La bontà della predizione è stata misurata in termini di

- **Precision**: $\frac{TP}{TP+FP}$, può essere interpretato come l'accuratezza nella predizioni di istanze positive
- **Recall**: $\frac{TP}{TP+FN}$, esprime la percentuale delle istanze correttamente stimate come positive
- **Kappa**: esprime, in termini percentuali, quante volte si è stati più precisi di una predizione puramente casuale
- **AUC**: Area Under Curve, misura l'area dela curva **ROC**. Un valore prossimo o pari ad 1 è sinonimo di modello predittivo perfetto. Un valore prossimo o uguale a 0.5 è sinonimo di un modello predittivo sostanzialmente inutile. Un valore prossimo o uguale allo 0 è tipico di predittori che effettuano una stima opposta di un attributo rispetto l'effettivo valore

Per il calcolo di queste metriche è necessario avere una porzione di dataset, il **test set**, su cui il modello possa effettuare la predizione.
Nell'ambito di questo progetto è stato usato il metodo del **Walk Forward** come processo di validazione. Questo metodo è un metodo di tipo time-series, ovvero utilizzato proprio per validare i dataset le cui istanze non sono indipendenti, ma hanno una dipendenza temporale. Nel nostro contesto difatti, il valore della *difettività* di un'istanza può dipendere dal valore passato dell'istanza stessa o da altre. Il *Walk Forward* consiste nel dividere il dataset in K sottoinsiemi, in modo che ogni insieme comprenda delle istanze indipendenti e appartenenti allo stesso intervallo temporale. Dopodichè viene seguito questo pseudocodice

Algorithm 1 Applicazione del metodo Walk Farward

Require: Dataset suddiviso in K sottoinsiemi temporalmente distinti

for $i = 1; i \leq K-1; ++i$ **do**
  TrainigSet $\leftarrow$ Unione dei Sottoinsiemi fino all' i esimo
  TestSet $\leftarrow$ Sottoinsieme $i+1$ esimo
  Istruisci il classificatore
  Calcola e memorizza metriche di accuratezza
end for
return media di ciascuna metrica calcolata nel ciclo for

Nel progetto proposto l'intero processo di validazione è stato implementato nella classe *DatasetAnalyzer* seguendo questi passi

1) Selezionamento di una terna fatta da: *Classificatore*, tipo di *Sampling*, tipo di *Feature Selection*
2) Applicazione del *Walk Forward*, considerando come insieme di istanze temporalmente indipendenti, i file di una release dell'applicazione
3) Per ogni passo del *Walk Forward*, raccolta in un file in formato *csv* dei seguenti dati
   - Numero di release usate nel training set
   - Grandezza del training set rispetto al test set in termini percentuali
   - Percentuale di classi difettive in training e test set
   - Tipo di classificatore, FS e Sampling applicati
   - Numero di TP, TN, FP, FN
   - Valori di: Recall, Precision, Kappa, AUC

Questo procedimento è stato iterato per tutte le terne del prodotto cartesiano tra (Classificatore={Random Forest, Native Bayes, IBk})x(FS={SI, NO})x(Sampling={No, Smote, Undersampling, Oversampling}).
Un esempio del dataset risultante è esposto nelle figure 6 e 7.

VI. Analisi Risultati

Dal processo di validazione e dalle metriche di accuratezza calcolate durante ogni step del *Walk Forward*, possiamo fare le seguenti osservazioni

- Il numero delle classi difettose in ciascuna fase del Walk Forward è rimasta perlopiù stabile: intorno al 0.3% per bookkeeper, ed intorno allo 0.04% in Syncope (similmente ad Avro e Falcon)
- La Precision e Recall di modelli *in cui è stato usato Sampling* è maggiore rispetto quelli in cui non è stato usato
- La percentuale delle classi difettive nel training set non è cambiato se applicato Sampling, dunque si deduce che weka non vada a modificare fisicamente il traning set, ma che usi qualche metadato che viene processato durante la fase di apprendimento

[10]cioò si può notare dai valori elevati di TN ma altrettanto elevati di FP, e bassi di TP

- La percentuale delle classi difettive in training e test set è prevalentemente diversa. Ciò in quanto non è stata applicata la **stratificazione**, con conseguente perdita di precisione.
- Ci sono stati casi in cui TP = FP = 0, e non è stato possibile calcolare Precision e recall (Generalmente in Syncope)
- Sono presenti test set (release) che non contengono classi difettose, o perlomeno release in cui con il processo applicato non sono state rivelate classi con difetti (Syncope, release 4)
- I valori di AUC sono relativamente elevati, questo è dovuto all'elevata precisione nei confronti dei True Negative, ovvero delle classi NON difettose
- L'applicazione della feature selection (approccio basato su filtro) non ha portato a dei benefici migliorativi rilevanti
- La precisione e recall si è mostrata più alta (specie in Bookkeeper) usando un training set con poche release. Questo possiamo associarlo al fatto che le ultime release hanno una distribuzione di classi difettive differente rispetto le prime release. Ciò significa che, selezionando le prime e poche[11] release come training set, è avvenuta una sorta di **stratificazione instrinseca**, ovvero che la percentuale di classi difettive nel training set è simile alla percentuale di classi difettive nel test set.

Il basso numero delle classi difettive ha portato, generalmente, a modelli predittivi molto accurati nel determinare la *NON* difettosità di una classe, ma non molto efficaci nella determinazione delle classi difettose.

Un sommario generale dei risultati ottenuti al variare di Sampling, Feature Selection e Classificatore, è stato messo insieme nelle Figure 8, 9. Dai grafici esposti in queste figure possiamo concludere che la configurazione che ha ottenuto risultati migliori è stata **RandomForest con FS e Oversampling per Bookkeeper** e **NativeBayes con FS e SMOTE per Syncope**.

VII. Conclusioni

Durante lo sviluppo di tale progetto è stato possibile apprendere come **non sempre**, durante l'implementazione di un'applicazione (open-source), ci si attiene a standard e pratiche per facilitare il processo del **Software Analysis**.
Alcuni esempi riscontrati sono: la mancanza delle *Affected Version* nei ticket di tipo bug, il non rispetto delle *convenzioni* nell'associare un ticket ID con un commit, l'utilizzo di più di un Issue Tracking System e la conseguente *perdita di consistenza* tra le alternative (GitHub e JIRA).
Una dimostrazione da poter toccare com mano è in Figura 3. In questa immagine sulla DESTRA sono riportati i commit (gli ultimi di un mese) associati ad un ticket di tipo bug il cui ID rispetta le convenzioni (eg. [PRJNAME-1234]). Mentre sulla SINISTRA sono stati raccolti i commit degli stessi ticket, ma il cui ID non rispetta pienamente le convenzioni (eg. -1234). Come si può notare solo a partire dalla fine del 2017, per il progetto Daffodil, la convenzione è stata iniziata ad esser rispettata.
Il non rispetto di tali pratiche spinge un Software Analyst ad impiegare molto più effort nel applicare il suo lavoro. L'uso di meccanismi per rimediare a tali mancanze possono portare allo sviluppo di modelli imprecisi, o peggio, non consistenti.
Per questo motivo l'**enfasi** va all'**attenersi** a tali convenzioni, pratiche o standard.

[11] In Bookkeeper ciò è avvenuto (Fig. 4) usando le prime 3 o 4 release come training set

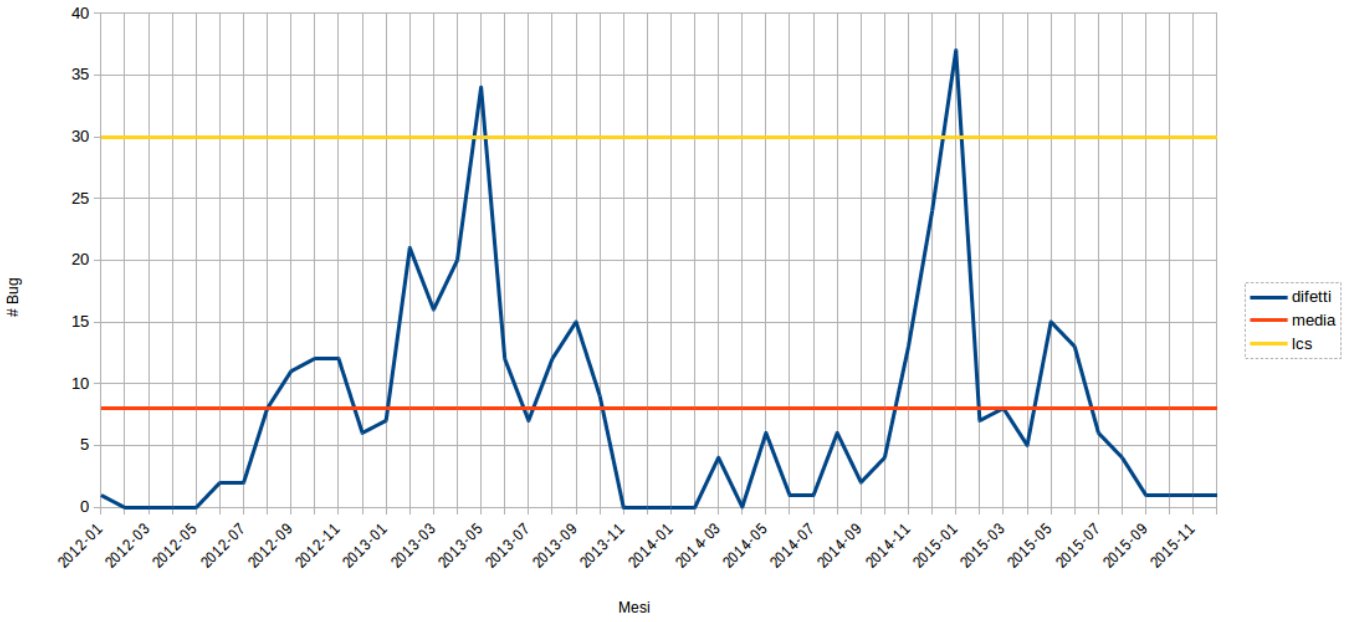

Fig. 1. Numero di difetti degli anni 2012-2015

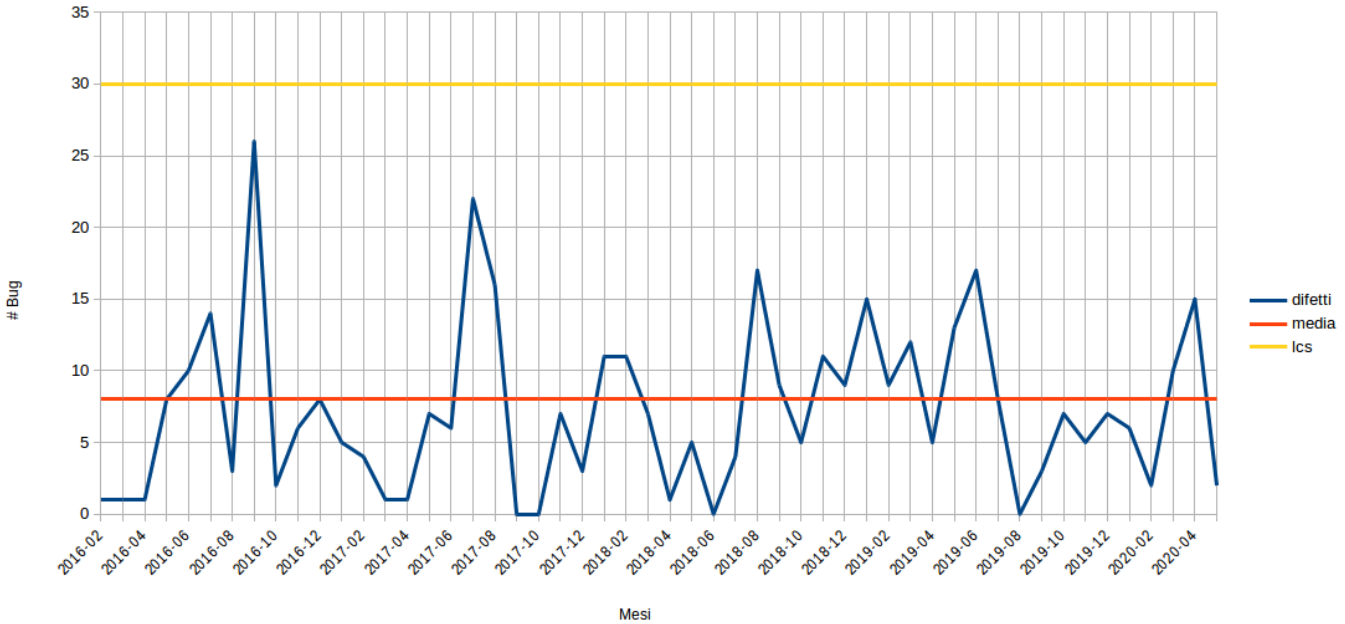

Fig. 2. Numero di difetti degli anni 2016-2020

Date	Num Bugs
2012-01	1
2012-02	0
2012-03	0
2012-04	0
2012-05	0
2012-06	2
2012-07	2
2012-08	8
2012-09	11
2012-10	12
2012-11	12
2012-12	6
2013-01	7
2013-02	21
2013-03	16
2013-04	20
2013-05	34
2013-06	12
2013-07	7
2013-08	12
2013-09	15
2013-10	10
2013-11	0
2013-12	0
2014-01	0
2014-02	0
2014-03	4

Date	Num Bugs
2017-11	6
2017-12	3
2018-01	11
2018-02	10
2018-03	7
2018-04	1
2018-05	5
2018-06	0
2018-07	3
2018-08	17
2018-09	9
2018-10	4
2018-11	11
2018-12	8
2019-01	15
2019-02	9
2019-03	12
2019-04	5
2019-05	13
2019-06	16
2019-07	8
2019-08	0
2019-09	3
2019-10	7
2019-11	5
2019-12	7
2020-01	6

Fig. 3. Differenza del dataset usando modalità di acquisizione dati ’lenta’ vs ’veloce’ (bugsPerMonthDAFFODIL.csv vs bugsPerMonthDAFFODIlFAST.csv)

IMMAGINI MILESTONE 2

Release Index	File	LOC	LOC Thouched	LOC Added	MAX LOC Added	AVERAGE LOC Added	Num Revisions	Num Authors	Churn	MAX Churn	AVERAGE Churn	ChgSe Size	MAX ChgSetSiz	AVERAGE ChgSetSize	Buggy
1	hedwig-server/src/main/java /org/apache/hedwig/server /common/UnexpectedError.java	27	74	36	35	18.0	2	2	35	35	17.5	454	282	227.0	No
1	bookkeeper-server/src /main/java/org/apache /bookkeeper/tools /BookKeeperTools.java	76	1614	68	51	11.33333333333	6	3	-671	2	-111.8333333	366	172	61.0	No
1	hedwig-client/src/test/java /org/apache/hedwig /util/TestFileUtils.java	32	82	41	41	41.0	1	1	41	41	41.0	282	282	282.0	No
1	hedwig-server/src/test/java /org/apache/hedwig/zookeeper /ZooKeeperTestBase.java	74	188	93	92	46.5	2	2	92	92	46.0	454	282	227.0	No
1	bookkeeper-server/src /main/java/org/apache /bookkeeper/bookie /MarkerFileChannel.java	118	0	0	0	0.0	1	1	0	0	0.0	62	62	62.0	No
1	hedwig-client/src/main/java /org/apache/hedwig/client /data/PubSubData.java	136	310	152	149	76.0	2	2	149	149	74.5	454	282	227.0	No
1	hedwig-server/src/test/java /org/apache/hedwig /ServerControl.java	204	1624	406	231	101.5	4	3	0	231	0.0	505	282	126.25	No
1	bookkeeper/src/test/java /org/apache/bookkeeper /test/CloseTest.java	60	148	74	74	74.0	1	1	74	74	74.0	282	282	282.0	No
1	hedwig-server/src/main/java /org/apache/hedwig/server /benchmark /BookieBenchmark.java	81	382	148	103	29.6	5	3	105	103	21.0	624	282	124.8	Yes

Fig. 4. Snapshot del Dataset usato per il training e testing del modello predittivo per BOOKKEEPER (finalTableBOOKKEEPER.csv)

Release Index	File	LOC	LOC Thouche	LOC Added	MAX LOC Added	AVERAGE LOC Added	Num Revisions	Num Authors	Churn	MAX Churn	AVERAGE Churn	ChgSet Size	MAX ChgSetSize	AVERAGE ChgSetSize	Buggy
1	console/src/main/java/org/apache /syncope/console/pages/panels /UserDetailsPanel.java	88	214	107	107	107.0	1	1	107	107	107.0	300	300	300.0	No
1	console/src/main/java/org/syncope /console/rest/WorkflowRestClient.java	27	158	58	38	14.5	4	3	37	38	9.25	1026	300	256.5	No
1	core/src/main/java/org/syncope /core/workflow/prcsiam /OpenAMActivate.java	70	576	144	106	36.0	4	1	0	106	0.0	87	46	21.75	No
1	console/src/main/java/org/syncope /console/wicket/markup/html/tree /LeafEditablePanel.java	47	232	58	58	29.0	2	1	0	58	0.0	107	59	53.5	No
1	core/src/main/java/org/syncope /core/persistence/beans /AttributeValueAsBoolean.java	57	188	7	6	2.3333333	3	1	-80	0	-26.6666666	74	32	24.66666666	No
1	core/src/main/java/org/syncope /core/validation/AlwaysTrueValidator.java	33	84	42	42	42.0	1	1	42	42	42.0	1	1	1.0	No
1	core/src/main/java/org/syncope /core/persistence/validation/entity /SyncTaskCheck.java	31	82	38	37	19.0	2	2	35	37	17.5	307	300	153.5	No
1	core/src/main/java/org/syncope/core/rest /controller/TaskController.java	362	3348	1073	228	29.805555	36	3	472	228	13.1111111	966	126	26.83333333	No
1	client/src/main/java/org/syncope/client /to/TaskExecutionTO.java	58	154	77	77	77.0	1	1	77	77	77.0	60	60	60.0	No

Fig. 5. Snapshot del Dataset usato per il training e testing del modello predittivo per SYNCOPE (finalTableSYNCOPE.csv)

Num Training Releases	% Training	% Defective In Training	% Defective In Testing	Classifier	Balancing	Feature Selection	TP	FP	TN	FN	Precision	Recall	ROC Area	Kappa
1	0.611788617886	0.279069767441	0.523560209424	RandomForest	None	None	55	31	60	45	0.639534883	0.55	0.64219780	0.2079013423551239
2	0.578143360752	0.373983739837	0.406685236768	RandomForest	None	None	110	71	142	36	0.607734806	0.753424	0.77596630	0.40482793882958107
3	0.771532184950	0.387779083431	0.349206349206	RandomForest	None	None	48	12	152	40	0.8	0.545454	0.89360449	0.5098743267504487
4	0.771532184950	0.387779083431	0.349206349206	RandomForest	None	None	48	12	152	40	0.8	0.545454	0.89360449	0.5098743267504487
5	0.901880621422	0.378966455122	0.25	RandomForest	None	None	22	35	55	8	0.385964912	0.733333	0.71222222	0.2649572649572651
6	0.890750182083	0.366312346688	0.026666666666	RandomForest	None	None	2	32	114	2	0.058823529	0.5	0.71575342	0.060427413411938184
7	0.915943962643	0.329206117989	0.031746031746	RandomForest	None	None	0	17	105	4	0.0	0.0	0.46106557	-0.05418326693227061
1	0.611788617886	0.279069767441	0.523560209424	NaiveBayes	None	None	97	89	2	3	0.521505376	0.97	0.57329670	-0.008378285320785024
2	0.578143360752	0.373983739837	0.406685236768	NaiveBayes	None	None	140	202	11	6	0.409356725	0.958904	0.73479001	0.008708581138487605
3	0.771532184950	0.387779083431	0.349206349206	NaiveBayes	None	None	5	0	164	83	1.0	0.056818	0.92090493	0.07270792693740014
4	0.771532184950	0.387779083431	0.349206349206	NaiveBayes	None	None	5	0	164	83	1.0	0.056818	0.92090493	0.07270792693740014
5	0.901880621422	0.378966455122	0.25	NaiveBayes	None	None	8	8	82	22	0.5	0.266666	0.75555555	0.21052631578947364
6	0.890750182083	0.366312346688	0.026666666666	NaiveBayes	None	None	0	15	131	4	0.0	0.0	0.71917808	-0.04395604395604462
7	0.915943962643	0.329206117989	0.031746031746	NaiveBayes	None	None	1	7	115	3	0.125	0.25	0.53073770	0.12983425414364594
1	0.611788617886	0.279069767441	0.523560209424	IBk	None	None	62	64	27	38	0.492063492	0.62	0.45835164	-0.08438160970722484
2	0.578143360752	0.373983739837	0.406685236768	IBk	None	None	93	95	118	53	0.494680851	0.636986	0.59548845	0.18271035225349946
3	0.771532184950	0.387779083431	0.349206349206	IBk	None	None	22	7	157	66	0.758620689	0.25	0.60365853	0.24544708777686636
4	0.771532184950	0.387779083431	0.349206349206	IBk	None	None	22	7	157	66	0.758620689	0.25	0.60365853	0.24544708777686636
5	0.901880621422	0.378966455122	0.25	IBk	None	None	16	34	56	14	0.32	0.533333	0.57777777	0.1272727272727273

Fig. 6. Snapshot del Dataset prodotto dopo il processso di validazione (wekaResultsBOOKKEEPER.csv)

Num Training Releases	% Training	% Defective In Training	% Defective In Testing	Classifier	Balancing	Feature Selection	TP	FP	TN	FN	Precision	Recall	ROC Area	Kappa
1	0.94744218	0.00221893491	0.01333333333	RandomForest	None	None	0	0	74	1	NaN	0.0	0.5	0.0
2	0.94378306	0.00280308339	0.01176470588	RandomForest	None	None	0	0	84	1	NaN	0.0	0.44642857142	0.0
4	0.82509505	0.00329163923	0.01863354037	RandomForest	None	None	0	0	316	6	NaN	0.0	0.40822784810	0.0
5	0.73463687	0.00597501357	0.07368421052	RandomForest	None	None	0	0	616	49	NaN	0.0	0.61486217863	0.0
6	0.84891598	0.02394253790	0.08520179372	RandomForest	None	None	0	0	408	38	NaN	0.0	0.68063080495	0.0
7	0.94494238	0.03319783197	0.14534883720	RandomForest	None	None	0	0	147	25	NaN	0.0	0.66639455782	0.0
8	0.90655832	0.03937259923	0.04968944099	RandomForest	None	None	0	0	306	16	NaN	0.0	0.59906045751	0.0
9	0.99422965	0.04033662217	0.05	RandomForest	None	None	0	0	19	1	NaN	0.0	0.73684210526	0.0
10	0.98186968	0.04039238315	0.03125	RandomForest	None	None	0	1	61	2	0.0	0.0	0.70967741935	-0.02127659574468085
11	0.96500820	0.04022662889	0.0234375	RandomForest	None	None	0	0	125	3	NaN	0.0	0.67333333333	0.0
13	0.90677134	0.03923160173	0.07105263157	RandomForest	None	None	0	0	353	27	NaN	0.0	0.76203966005	0.0
14	0.89977924	0.04219823356	0.11453744493	RandomForest	None	None	0	1	401	52	0.0	0.0	0.74115001913	-0.0043409299607649245
15	0.92694904	0.04944812362	0.01120448179	RandomForest	None	None	0	1	352	4	0.0	0.0	0.67209631728	-0.004501969611705188
16	0.99592418	0.04665438919	0.05	RandomForest	None	None	0	0	19	1	NaN	0.0	0.55263157894	0.0
17	0.92081065	0.04666802527	0.06635071090	RandomForest	None	None	0	1	393	28	0.0	0.0	0.66996011602	-0.004596946314233183
18	0.95382137	0.04822668418	0.03100775193	RandomForest	None	None	0	1	249	8	0.0	0.0	0.46625	-0.006938421509105175
19	0.98258881	0.04743153749	0.16161616161	RandomForest	None	None	0	0	83	16	NaN	0.0	0.57040662650	0.0
20	0.97933172	0.04941962715	0.05	RandomForest	None	None	0	0	114	6	NaN	0.0	0.49195906432	0.0
22	0.88014311	0.04166666666	0.00426439232	RandomForest	None	None	0	0	934	4	NaN	0.0	0.41059957173	0.0

Fig. 7. Snapshot del Dataset prodotto dopo il processso di validazione (wekaResultsSYNCOPE.csv)

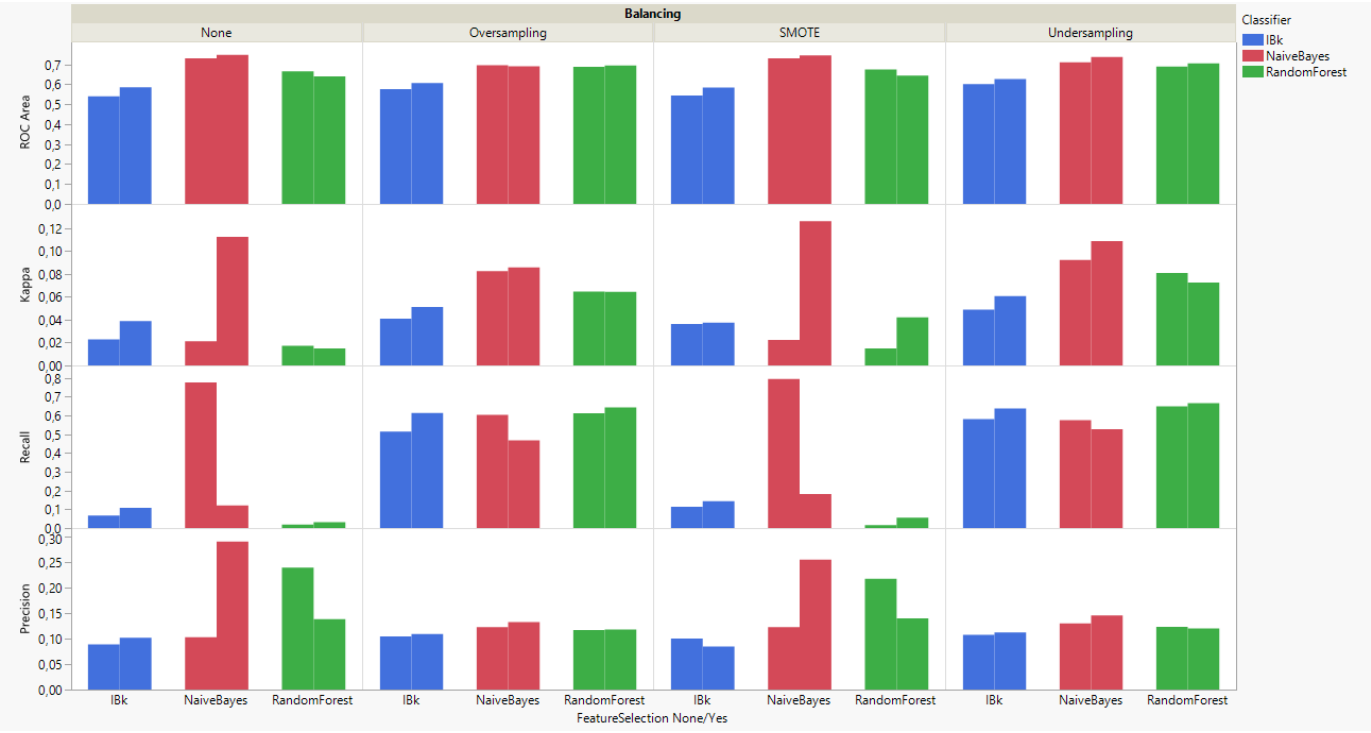

Fig. 8. Accuratezza dei classificatori in termini di Recall, Precision, AUC, e Kappa, rispetto Feature Selection e Sampling (Bookkeeper)

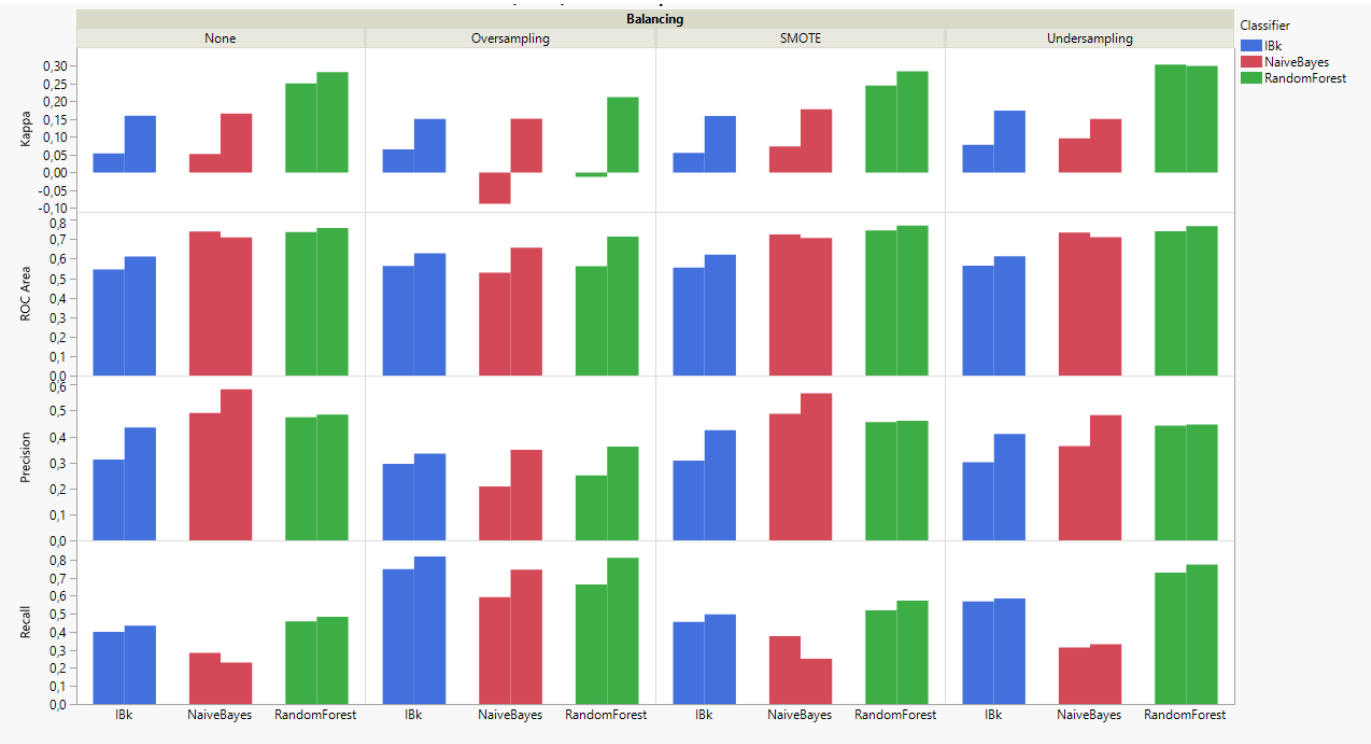

Fig. 9. Accuratezza dei classificatori in termini di Recall, Precision, AUC, e Kappa, rispetto Feature Selection e Sampling (Syncope)