Ingegneria del Software 2

Relazione di Fabiano Veglianti 0286057

Deliverable 1

Introduzione

L'obiettivo della deliverable 1 è quello di realizzare una carta di controllo per uno specifico attributo di un processo.

La carta di controllo è uno strumento statistico che permette di monitorare il valore di un attributo di un certo processo nel tempo, dunque tipicamente si pone sull'asse X il tempo e sull'asse Y il valore dell'attributo da monitorare.

La carta di controllo permette di determinare se il processo è stabile rispetto all'attributo di interesse, cioè se il valore che l'attributo assume nel tempo è contenuto in un certo intervallo, oppure se è instabile.

Variazioni anomale del valore dell'attributo osservati tramite la carta di controllo possono indicare che nel corrispondente periodo di tempo c'è stato un problema e quindi può suggerire di approfondire la cosa per indagare il problema e risolverlo. La carta di controllo può essere, però, usata anche per predire i valori che l'attributo del processo dovrebbe assumere e agire preventivamente in caso in cui i valori aspettati escono dai limiti tollerati.

Come già detto, la carta di controllo definisce un intervallo di valori dell'attributo all'interno del quale il processo è considerato stabile, si ammette un intervallo di valori validi perché le variazioni nel valore di un attributo possono dipendere anche da fattori inevitabili e incontrollabili che tipicamente generano piccole variazioni. Nel controllo di un processo siamo interessati a quei fattori che causano una grande variazione nel valore dell'attributo e soprattutto a fattori che sono controllabili!

Segue la realizzazione della carta di controllo dell'attributo *numero di bug fix* per il processo di sviluppo del progetto *Apache VCL*.

Progettazione

Per realizzare la carta di controllo si sono utilizzate informazioni prese da jira e dalla repository di github del progetto.

Da Jira sono stati raccolti i ticket relativi ai *Bug* con *status closed* o *resolved* e *resolution fixed*.

Da github sono stati raccolti tutti i commit effettuati sul branch master.

Per ogni ticket raccolto da jira si è cercato l'ultimo commit su github nel cui commento fosse contenuto l'ID del ticket considerato. Tipicamente ad ogni bug è associato un unico ticket, dunque un unico ID di un ticket, e di conseguenza l'ultimo commit che contiene nel suo commento l'ID del ticket si riferisce alla chiusura del bug, se il bug è stato chiuso ovviamente.

Fatto ciò, si è creato un insieme di date di commit considerando solo il mese e l'anno in cui è avvenuto l'ultimo commit per ogni ticket.

Per ogni data nell'insieme di date di commit è stato contato il numero di commit avvenuti in quello specifico mese-anno considerando solo i commit ultimi per qualche ticket.
Poi, fissata la prima e l'ultima data presente nell'insieme, si è riempito l'insieme delle date con tutte le date (intese come mese-anno) che non erano già presenti.

Infine, si è creato il file .csv con colonne data e numero di bug fix.

Implementazione

1. Il primo passo effettuato è stato quello di reperire da Jira i ticket relativi ai bug con le caratteristiche descritte nel paragrafo *Progettazione*, dunque utilizzando l'API rest di jira sono stati presi gli ID dei ticket e sono stati messi in un ArrayList<String>.
2. Il secondo passo è stato prendere da Github la lista dei commit sul branch master, questo passaggio è stato fatto utilizzando la libreria *JGit*. Tramite JGit si può ottenere una copia locale della repository facendo clone o checkout ed inoltre si può lavorare su questa repository. Dunque, dopo aver creato una copia locale della repository si ottiene la lista dei commit effettuati sul branch master di quella repository.
3. Il terzo passo è stato creare un HashMap<String, Date> che mette in corrispondenza una chiave=TicketID ad una data=dataDelCommit. Dunque, scorrendo la lista dei ticket, scorrendo la lista dei commit, per ogni commit che contiene nel messaggio l'ID del ticket è stata salvata la coppia <TicketID, DataCommit> nell' HashMap se l'elemento con la stessa chiave non era già

presente oppure se l'elemento con la stessa chiave era già presente ma la data del vecchio elemento era precedente alla data del commit attualmente considerato.

4. Il quarto passo è stato considerare solo i valori, dunque le date, della mappa creata al passo precedente e a partire da questi è stata creata un'ArrayList<String>. Per ogni data è data creata una stringa con formato "yyyy-MM" che è stata messa nella lista.
   Si noti che la lista contiene date ripetute, perché ad esempio la data 2015/08/22 e 2015/08/12 corrispondono entrambe alla stringa "2015-08", ma l'ArrayList ammette duplicati!
5. Il quinto passo è stato creare un insieme ordinato di date a partire dall'ArrayList creata al passo precedente. Si noti in questo caso che un insieme non ammette duplicati, inoltre l'ordinamento lessicografico delle stringhe ottenute da date applicando il formato "yyyy-MM" è uguale all'ordinamento cronologico delle date da cui derivano.
6. Il sesto passo è stato aggiungere a questo insieme le date mancanti. L'insieme contiene tutte e sole le date relative a qualche commit che ha effettuato un bugfix, ora se per esempio abbiamo un bug fix il mese 2015-08 e uno il mese 2015-10, ma nessuno nel mese 2015-09 allora l'elemento "2015-09" sarà assente dall'insieme, tuttavia per poter avere dati corretti, è necessario considerare anche i mesi in cui non sono stati effettuati bug fix, il sesto passo risolve proprio questo problema andando a considerare una ad una le date presenti nell'insieme e aggiungere la successiva, se la successiva non è presente.
7. Il settimo passo è stato creare e riempire un ArrayList<Integer>, per ogni elemento nell'insieme ottenuto al sesto passo, è stato calcolato il numero di occorrenze di quell'elemento nell'ArrayList ottenuto al quarto passo. Le date aggiunte manualmente al sesto passo avranno 0 occorrenze, come è giusto che sia; le altre date invece avranno almeno un'occorrenza, ma possono averne anche di più perché l'ArrayList (ottenuta al passo 4) contiene date duplicate. Questa ArrayList contiene il numero di commit per bug fix effettuati in ogni mese considerato.
8. Ci troviamo a questo punto in possesso dell'insieme ottenuto al sesto passo e l'ArrayList creata al settimo passo, queste due liste di elementi hanno stessa lunghezza per come la seconda è stata costruita. Dunque, queste due liste sono state utilizzate per creare un file .csv di nome *processControlChart.csv* che contiene per ogni riga anno-mese e numero di bug fix corrispondenti.

9. Infine, utilizzando Microsoft Excel è stato creato il process control chart. Il process control chart è stato realizzato calcolando il numero medio di bug fix in ogni mese, l'upper limit e il lower limit, questi calcolati come segue:

$$UL = E + 3\sigma$$

$$LL = \max\{0; \mathrm{E} - 3\sigma\}$$

A partire da questi valori, il process control chart è stato realizzato come grafico a linee: sull'asse X ci sono le date, sull'asse Y il numero di bug fix. Per il numero di bug fix è stato utilizzato il tipo di grafico *Linee con indicatori*, mentre per la media, l'upper limit e il lower limit è stato utilizzato il tipo di grafico *Linee*.

Risultati

I tickets presi in considerazione sono 434, di questi solo 388 hanno una corrispondenza nei 2625 commit effettuati sul branch master della repository del progetto, dunque i dati ottenuti si riferiscono a 388 tickets.

Il periodo temporale preso in considerazione va da Gennaio 2009 ad Agosto 2019, si è optato per considerare come ultima data la data dell'ultimo commit perché risale a più di un anno fa ed è possibile che i dati su Jira non siano stati più aggiornati.

Il process control chart ottenuto è il seguente:

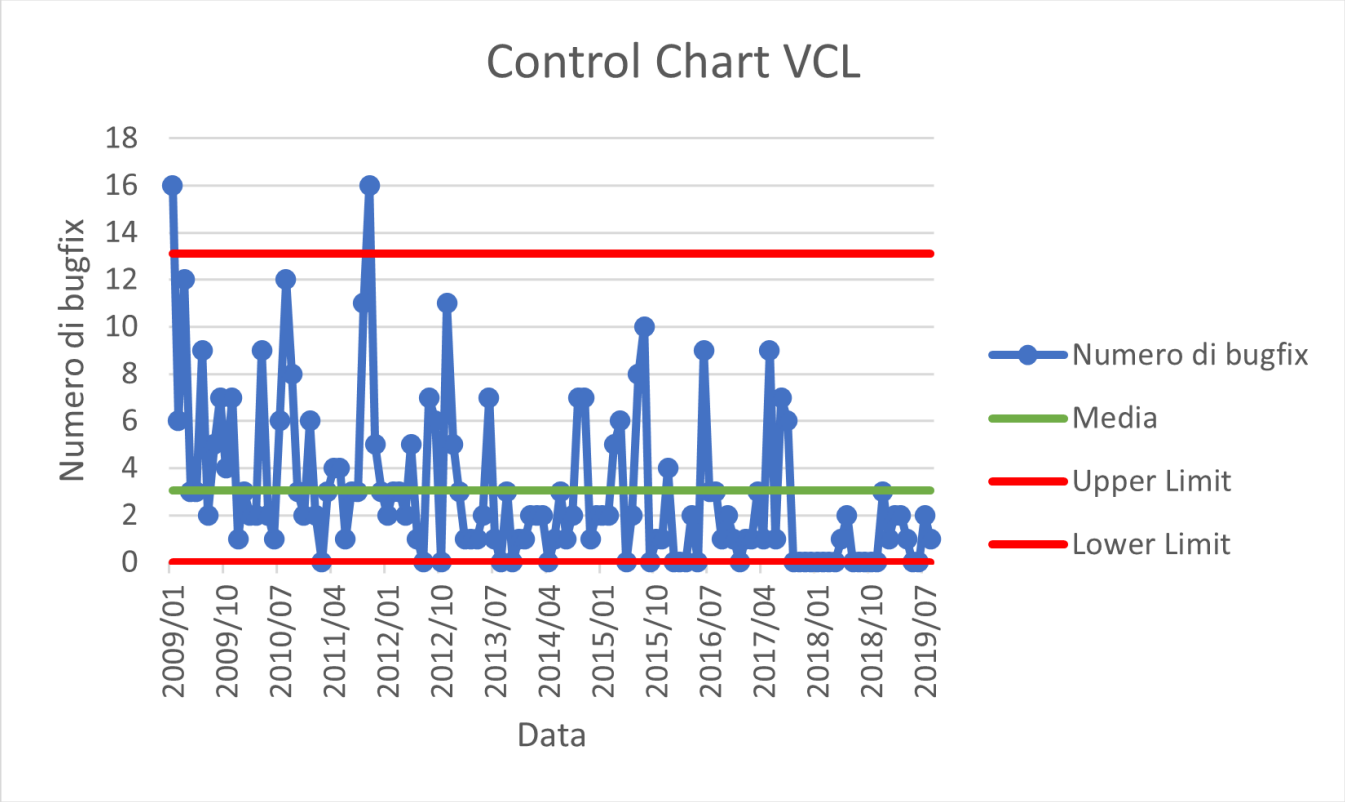

Figura 1: Process Control Chart per il numero di bug fix relativo al progetto Apache VCL

Per quanto riguarda il lower limit si è considerato ragionevole mettere un valore minimo a 0, perché è un valore con unità di misura “numero di bug fix” e non avrebbe senso considerare un numero di bug fix negativo.

Dall’osservazione del process control chart si può dedurre che il processo non può essere considerato stabile per l’attributo bug fix, infatti non tutti i punti sono all’interno dell’intervallo definito da [LL, UL]; in particolare nei mesi 2009/01 e 2011/09 il numero di bug fix è pari a 16 ed è superiore all’upper limit.

Per il caso 2009/01, siccome non si hanno dati precedenti a quella data, è difficile motivare il perché ci siano così tanti bug fix.

Per il caso 2011/09 invece, si può notare che si mantiene un andamento crescente già presente dal mese precedente, da ciò e guardando l’andamento generale della curva si può supporre che nei mesi 2011/08 e 2011/09 ci sia stato uno spostamento delle risorse verso il debugging, perciò il numero di bug fix in quel periodo è alto. Questa supposizione è fatta considerando l’andamento generale della curva e cioè il fatto che la varianza è alta, dunque è come se si alternassero periodi di basso impiego delle risorse nel debugging, con conseguente accumulo di bug e basso numero di bug fix, a periodi di alto impiego di risorse nel debugging, con conseguente aumento del numero di bug fix. Questa considerazione è comunque solo una considerazione fatta su dati incompleti, in quanto per dire ciò con maggiore ragionevolezza si dovrebbe almeno osservare che nelle fasi di “basso impiego delle risorse nel debugging” ci sia un alto impiego delle risorse in qualche altra parte o si debba sapere che le risorse impiegate nel progetto non sono costanti nel tempo.

Deliverable 2

Introduzione

L’obiettivo della deliverable 2 è quello di effettuare uno studio empirico finalizzato valutare l’efficacia di diverse tecniche di feature selection e di balancing in relazione all’accuratezza di diversi classificatori per la predizione delle difettosità delle classi, in particolare l’obiettivo è usare

- 2 progetti
- Walk forward come evaluation technique.
- No selection / Best first come tecniche di feature selection.
- No sampling / Oversampling / Undersampling / SMOTE come tecniche di balancing.
- RandomForest / NaiveBayes / IBk come classificatori

e trovare quale tecniche aumentano l'accuratezza per quali classificatori e quali dataset.

Le metriche di accuratezza prese in considerazione sono:

- Precision
- Recall
- Area Under ROC (AUC)
- Kappa

La predizione delle difettosità delle classi è un aspetto dell'ingegneria del software che permette di aumentare l'efficienza del processo di sviluppo. Infatti, poter determinare quale classi hanno un'alta probabilità di presentare dei difetti, permette di ridurre il costo (sia in termini economici che di tempo) necessario al testing del codice concentrando le risorse disponibili per il testing su alcune classi piuttosto che su altre. In realtà la predizione della difettosità non è un concetto che si limita alla programmazione a oggetti e dunque alle singole classi, ma nel corso della discussione tratteremo solo questo aspetto.

Per la realizzazione della deliverable sono stati considerati i progetti *apache/BOOKKEEPER e apache/SYNCOPE*.

Progettazione

Per effettuare l'analisi dell'affidabilità dei vari classificatori considerando le diverse tipologie di preprocessing applicate ai dati, si è diviso il lavoro in due parti: nella prima si realizza un file .csv che funge da dataset per il modello e nel secondo si calcola l'affidabilità applicando il modello stesso.

Per quanto riguarda la realizzazione del dataset si è attuato un approccio top-down riassumibile nei seguenti punti:

1. Si effettua il *clone* della repository del progetto considerato in locale.
2. Si ottengono da Jira le varie releases del progetto.
3. Si ottengono da Jira le informazioni sui tickets relativi ai bugfix.
4. Si ottengono da Github le varie releases del progetto.
5. Si ottengono da Github le informazioni sui commit effettuati.
6. Per ogni ticket si trova l'ultimo commit effettuato riguardo quel ticket.
7. Si calcola IV, OV e FV per ogni ticket: questi dati possono essere presenti in Jira e quindi ottenuti al punto 3, oppure possono essere calcolati applicando la variante *Moving_Window* di *Proportion*.
8. Si ottengono da Github le classi (file .java) presenti al rilascio di ogni release.
9. Per ogni classe in ogni release si calcolano le varie metriche e la bugginess.

10. Si scrive il file .csv.

Per quanto riguarda, invece, il computo delle performances dei vari classificatori considerando le varie alternative di preprocessing indicate, si è proceduto sempre con un approccio top-down riassumibile nei seguenti punti:

1. Si carica il dataset.
2. Per ogni classificatore, per ogni metodo di feature selection, per ogni metodo di resampling, per ogni iterazione di walk forward, si calcolano le metriche di accuratezza indicate nella consegna per il caso specifico.
3. Si scrivono tutti i risultati ottenuti in un file .csv.

Implementazione

Per la trattazione dell'implementazione si entrerà nel dettaglio di ogni punto definito nel paragrafo Progettazione, dunque si farà riferimento a quei punti.

Per quanto riguarda l'interazione con Jira, questa è avvenuta attraverso l'API di Jira stesso, mentre per quanto riguarda l'interazione con GitHub si è utilizzato la libreria *JGit*.

La parte che riguarda la valutazione delle performances è stata realizzata usando l'api Java del software *Weka*.

Partendo dalla creazione del dataset l'implementazione di ogni punto è avvenuta come segue:

1. La copia locale della repository su github è ottenuta tramite *checkout* o *clone*. Se è la prima volta che si esegue il codice, allora la copia locale della repository non è presente e dunque si procederà con il *clone* della repository, mentre se la copia locale della repository è presente si procede con il comando *checkout* per effettuare l'update dei files.
2. Per l'ottenimento delle release da Jira, semplicemente si è utilizzato l'api di Jira. In particolare, di tutte le releases restituite si sono mantenute solo le releases che sono state effettivamente rilasciate e per le quali è disponibile sul sito la data di rilascio. Il risultato del punto 2 è una oggetto List<ReleaseJira> che rappresenta una lista di oggetti *ReleaseJira*, ognuno dei quali mantiene un ID incrementale, il nome e la data di rilascio di una specifica release.
3. Per l'ottenimento dei dati relativi ai tickets si è utilizzata l'api di Jira, le informazioni tenute per ogni tickets sono: l'ID, la data di apertura e di chiusura, la lista delle versioni affette e la lista delle versioni in cui è stato effettuato il fix. I dati raccolti in questa fase sono mantenuti in due oggetti List, uno che tratta

esclusivamente i dati relativi ai tickets di Jira, l'altro che fa da ponte tra questi e i dati che successivamente prenderemo da github.
Prima di passare al punto 4, i dati appena raccolti vengono controllati e puliti:

- La prima operazione di pulizia riguarda la lista di *fixed version* restituita da jira.
  Sotto la voce *fixVersions* Jira, per ogni ticket, restituisce una lista di versioni, in alcuni casi però questa lista è vuota oppure contiene più di un elemento.
  Quando la lista è vuota si impone come unica fixed version, la prima release successiva alla data di chiusura del ticket, se esiste. Quando la lista contiene più di un elemento (occasione che potrebbe occorrere quando il bug viene fixato in rami paralleli dell'albero di lavoro) viene considerata come unica fixed version la più vecchia versione in questa lista.
- La seconda operazione di pulizia riguarda l'eliminazione dei dati che non portano alcuna informazione utile.
  Per ogni ticket, si considera OV la prima release successiva alla data di creazione del ticket.
  A partire dalla OV e dalla FV, si fanno diverse considerazioni per cui i dati vengono mantenuti quando OV < FV oppure IV < OV == FV, negli altri casi i tickets vengono eliminati (dunque non vengono considerati nel resto del programma).
  Si considera IV la versione più vecchia tra le versioni affette restituite da Jira, quando ce n'è almeno una.
  Dunque vengono non considerati i tickets per cui:
  i. IV == OV == FV, in questo caso infatti il bug non è esistito in nessuna release perché l'intervallo di versioni da considerare affette è l'intervallo [IV, FV), che in questo caso specifico è [FV, FV), cioè l'insieme vuoto.
  ii. OV == FV e non si conosce l'IV, in questo caso infatti si dovrebbe applicare proportion, ma a partire da $IV = FV - (FV - OV) * P$ per ogni valore di P si avrebbe $IV = FV - 0 * P = FV$ e dunque si ricade nel caso precedente.
  iii. IV > OV == FV, infatti in questo caso l'IV ottenuta da Jira non è coerente con il resto dei dati e dunque ricadiamo nel caso in cui OV == FV e non si conosce l'IV.

Prima di effettuare quanto appena descritto, nel caso in cui OV>FV si pone OV = FV, infatti, dopo aver imposto ciò possiamo trovarci in una delle seguenti tre situazioni:

1. IV non è presente su Jira, ma allora ci troviamo nel caso in cui OV==FV e l'IV non è conosciuta, dunque si scarta il ticket.
2. IV ≥ OV==FV, ma allora siamo nel caso (i) o (iii) di prima e dunque si scarta il ticket.
3. IV è presente su Jira e IV < OV==FV e questo caso lo vogliamo mantenere perché da Jira sappiamo sia IV che FV, sono coerenti tra di loro e sappiamo che le versioni affette da questo bug sono quelle nell'intervallo [IV, FV).
   La scelta di porre OV = FV, affliggerà proportion in un modo che sarà discusso in seguito, nel punto 7.

4. Tramite la libreria *JGit* si ottengono da Github la lista delle releases del progetto.
   I nomi delle releases ottenute sono modificati in modo da mettere in relazione ogni release presa da Jira con una presa da Github e le releases che non hanno corrispondenza sono eliminate. Questo è fatto perché in entrambi i progetti considerati le releases prese da Jira sono un sottoinsieme delle releases presenti su Github.
5. I commits di un progetto sono ottenuti da Github.
   La libreria *JGit* permette di ottenere tutti i commits effettuati tra due releases; ordinando le releases, i commits considerati sono tutti i commits tra ogni release e la precedente.
   Il risultato di questo passo è un oggetto List<GitCommit>, dunque una lista. Gli oggetti nella lista sono oggetti *GitCommit* che mantengono per ogni commit il proprio identificativo all'interno del progetto, l'identificativo del *commit parent,* se esiste, la data in cui il commit è stato effettuato ed infine il commento del commit.
6. Considerando l'id dei tickets, risultato del passo 3, e i commenti dei commits ottenuti al passo 5, cerchiamo per l'id di ogni ticket i commits che contengono quell'id nel messaggio.
   Potrebbero esistere più commits che soddisfano questo requisito, quindi manteniamo soltanto l'ultimo (in ordine cronologico).
   Può capitare che per qualche ticket non esista alcun commit che ne contenga l'id nel commento, in questo caso quel ticket è ignorato.
7. A partire dalle informazioni ottenute ai passi 2 e 3, viene applicato *proportion moving_window* per trovare la lista di *affected versions* di ogni bug. La

discussione di proportion moving_window richiede di poter distinguere tra la proporzione P_bug tra (FV-IV) e (FV-OV) per uno specifico bug e P_medio come la media delle P_bug relative all'ultimo 1% dei bug fixati.
Innanzi tutto c'è da dire che siccome studi empirici dimostrano che il 51% dei difetti non ha AV su Jira o ha AV non affidabile, mantenere la media delle P_bug nella finestra (cioè P_medio) aggiornata significherebbe farlo mediamente 1 difetto su 2 e "sprecare" memoria mentre non mantenerla significherebbe calcolarla mediamente 1 volta su 2, ma non sprecare memoria, dunque si preferisce non mantenere P_medio, ma ricalcolarlo ogni volta.
Per quanto riguarda invece i bug considerati, abbiamo detto che dopo la fase di pulizia rimangono solo i bug con OV<FV e con IV<OV==FV.
Per quanto riguarda la prima categoria, il calcolo di P_bug non presenta alcun problema, mentre per quanto riguarda la seconda categoria P_bug è impossibile da calcolare a causa di uno 0 a denominatore, dunque si sceglie arbitrariamente P_bug = 1.
Ora, considerando che i bug su cui applichiamo proportion sono quelli per cui l'IV non è conosciuta e cioè, a causa della pulizia, quelli che hanno OV<FV e l'IV non è conosciuta, sarebbe più corretto usare per questi bug una P_media calcolata sui soli bug della stessa categoria, cioè quelli che hanno OV<FV e l'IV è conosciuta; dunque l'uso arbitrario di P_bug = 1 nel caso IV<OV==FV causa l'utilizzo di un valore P_medio minore rispetto a quello che andrebbe utilizzato.
Ne consegue che l'implementazione di *proportion moving_window* realizzata tende ad avere meno FP e più FN rispetto alla versione standard, dunque maggiore precision e minore recall.

8. Per ogni release ottenute da Github al passo 4, si cercano i files .java posseduti.
   Ogni release corrisponde ad uno specifico commit ed è possibile attraverso la libreria *JGit* risalire ai files presenti nel momento del commit.
   Ogni file è un oggetto *ClassProject* che mantiene i dati di una classe.
   I dati settati a questo punto sono la dimensione della classe e un token che attesta che la classe è realmente presente nella release; questo token, apparentemente bizzarro, è necessario per attuare un meccanismo spiegato nell'ultima considerazione riguardo i risultati, spiegata nell'apposita sezione.
9. Il processo di calcolo delle varie metriche per ogni classe di ogni release è un processo laborioso, si riassumono quindi i punti salienti.
   Le metriche considerate sono divise in due tipologie: "per release" e "storiche"; quelle per release sono calcolate considerando tutte le revisioni che separano una release dalla release precedente, mentre quelle storiche sono calcolate

considerando tutte le revisioni dall'inizio del progetto alla release corrente. Le metriche considerate sono:

- size – storica
  Sono state ignorare le righe vuote, e i commenti.
- locTouched – per release
  Righe aggiunte + Righe rimosse
- numRevisions – storica
- numBugFixed – storica
- locAdded – per release
- maxLocAdded – per release
- avgLocAdded – per release
- churn – per release
  Righe aggiunte - righe rimosse
- maxChurn – per release
- avgChurn – per release
- age – storico
  Età in settimane dalla data di inserimento della classe alla data della release.

Per ogni release si considerano i commits tra la release e quella precedente, per ogni commit si considera la differenza tra il commit e il suo *parent*. Il risultato della differenza tra due commit è una lista di oggetti di tipo *DiffEntry* della libreria *JGit*; ogni oggetto *DiffEntry* rappresenta una differenza tra i due commit e può riguardare l'aggiunta (ADD), la rimozione (DELETE), la modifica (MODIFY), la duplicazione (COPY) o lo spostamento/rinominazione (RENAME) di una classe.

Tutti i casi sono stati presi in considerazione:

- L'aggiunta è stata presa in considerazione perché se si fossero considerati per ogni release solo i files presenti al momento del rilascio della release si sarebbero persi dei dati, in particolare s'immagini che tra la release 2 e 3 esistano 200 commit e che il file a.java viene introdotto nel primo commit dei 200 commit, viene modificato e poi viene rinominato in b.java al 160-esimo commit, quello che succede è che nella lista dei files presenti nella release ottenuta da Github il file a.java non esiste (perché al momento del rilascio il file a.java effettivamente non esiste) e in più le metriche relative al file b.java non considererebbero i cambiamenti effettuati nei primi 160 commit che separano le due release.

- La rimozione è stata presa in considerazione perché le classi eliminate non vengono considerate nelle release successive (a meno che non vengano aggiunte di nuovo tramite un nuovo ADD).
- La duplicazione e lo spostamento sono entrambe considerate per mantenere le misure delle classi quanto più corrette possibili.
  Per quanto riguarda la rinominazione lo scopo è stato già anticipato nell'esempio riguardo l'aggiunta: gestire la rinominazione del file a.java in b.java permette di preservare i dati relativi ai primi 160 commits che impattano la classe b.java quando possedeva un nome diverso.
  Un altro caso in cui la rinominazione è necessaria è quando consideriamo i dati storici, perché, per esempio, facendo riferimento alle classi a.java e b.java, l'età della classe b.java è considerata non dal momento della rinominazione, ma dal momento in cui a.java è stata inserita.
  Le motivazioni dietro la duplicazione sono analoghe.
  La differenza maggiore tra duplicazione e rinominazione è che la classe che viene rinominata è considerata rimossa.
- La modifica di una classe permette di aggiornare la maggior parte delle metriche prese in considerazione per la realizzazione del dataset.

Per quanto riguarda la bugginess di una classe partiamo dal fatto che siamo in possesso di una lista di bugs, per ogni bug possediamo il commit che ha effettuato il fix e la lista delle versioni affette. Per ogni bug si considera il commit che ha effettuato il fix e si considerano le differenze tra questo commit e il rispettivo parent: per il calcolo della bugginess si considerano solo i cambiamenti di tipo MODIFY, dunque se una classe è eliminata per fixare un bug, quella classe non si considera buggata. Per ogni cambiamento di tipo MODIFY, si considera il nome della classe che l'ha subito e un insieme di nomi (per ora vuoto) e si cerca la classe nell'ultima release affetta.
Se la classe viene trovata si setta la bugginess a *true*, si aumenta il numero di bug per quella classe e si salvano nell'insieme di nomi tutti i nomi che quella classe ha assunto tra la release considerata e la precedente. Per ogni nome salvato (e dunque ogni oggetto *ClassProject* diverso) si setta la *bugginess* a *true*.
Si noti che l'insieme dei nomi non è resettato ogni volta che si considera una nuova affected version per uno stesso bug, ma anzi è aggiornato con eventuali nuovi nomi man mano che si retrocede tra le affected versions. Terminata l'analisi dell'ultima release, si passa alla penultima e si ripete l'intero procedimento.

10. La scrittura del file *.csv* avviene attraverso la classe *FileWriter* della libreria io.
    Il file .csv possiede le colonne
    - versionID
    - versionName
    - filename
    - size
    - locTouched
    - numRevision
    - numBugFix
    - locAdded
    - maxLocAdded
    - avgLocAdded
    - churn
    - maxChurn
    - avgChurn
    - age
    - buggy

Per quanto riguarda la parte di valutazione delle performances, abbiamo invece:

1. Caricamento del dataset in memoria: il dataset è salvato su disco in un file .csv, si utilizza quindi l'apposita classe della libreria *Weka* per caricare un file .csv.
2. Per ogni classificatore, per ogni metodo di feature selection, per ogni metodo di resampling, per ogni iterazione di walk forward, si calcolano le misure della performance indicate nella consegna per il caso specifico.
   Questo punto è stato interamente realizzato tramite l'api Java del software Weka.
   Il risultato di questa fase è una lista di oggetti *Result* che mantengono le varie metriche di accuratezza di ogni caso considerato.
   Le metriche calcolate sono: *TP*, *FP*, *TN*, *FN*, *precision*, *recall*, *auc*, *kappa*.
   Per quanto riguarda i classificatori utilizzati, i metodi di feature selection e i metodi di resampling utilizzati, questi sono quelli definiti negli obiettivi della deliverable.
   Cose in più o particolari che sono state fatte sono: fare preventivamente feature selection eliminando le features *VersionID, VersionName* e *filename* e non usare la versione standard di smote, ma specificare delle opzioni particolari.
   L'utilizzo di default della classe SMOTE prevede che la percentuale delle istanze smote da creare sia del 100%, cioè raddoppia la cardinalità dell'insieme della

classe minoritaria, dunque è stato necessario specificare un' opzione per avere un dataset perfettamente bilanciato dopo aver applicato smote. L'opzione passata è -P, *parameter.*
Il parametro è calcolato come (#classe_magg - #classe_min)/#classe_min

3. La scrittura del file *.csv* avviene attraverso la classe *FileWriter* della libreria io.
   Il file .csv possiede le colonne
   - Progetto
   - #Training Release
   - %Training
   - %Buggy in training
   - %Buggy in testing
   - Classifier
   - Balancing
   - Feature selection
   - TP
   - FP
   - TN
   - FN
   - Precision
   - Recall
   - Area Under ROC
   - Kappa

Risultati

La discussione dei risultati sarà trattata inizialmente per progetto, i grafici mostrati sono stati generati tramite la libreria *matplotlib* di Python.

Per ogni progetto si discuteranno prima singolarmente classificatore, metodo di feature selection e metodo di balancing, poi si discuterà del preprocessing migliore indipendentemente dal classificatore utilizzato e infine si offrirà una panoramica completa dei risultati ottenuti per quel progetto.

Le metriche di accuratezza considerate sono Precision, Recall, Area Under ROC e Kappa.

Tutte le conclusioni discusse sono tratte dall'osservazione dei boxplot e delle mediane delle distribuzioni delle metriche di accuratezza considerate.

Bookkeeper

Il dataset relativo al progetto Bookkeeper è costruito considerando 11 delle 36 releases presenti su github e contiene i dati relativi a 160 dei 429 bugs presi da Jira.

Iniziamo con l'osservare le performances dei diversi classificatori indipendentemente dal preprocessing dei dati effettuato. La figura2 mostra i boxplot delle metriche di accuratezza in funzione del classificatore utilizzato.

Dai risultati ottenuti Random Forest risulta essere il classificatore migliore da utilizzare indipendentemente dalla metrica di accuratezza che vogliamo massimizzare.
IBk mostra prestazioni leggermente inferiori rispetto a Random Forest su tutte le metriche di accuratezza; rispetto a Naive Bayes, IBk risulta essere migliore sempre, tranne quando l'obiettivo è massimizzare AUC, tuttavia il peggioramento in termini di Recall e Kappa di Naive Bayes rispetto agli altri classificatori è netto ed è dunque sconsigliato, alla luce dei dati ottentuti, utilizzare Naive Bayes.
La tabella che segue mostra le mediane delle distribuzioni delle metriche di accuratezza in funzione del classificatore utilizzato.

	Random Forest	Naive Bayes	IBk
Precision	0.81	0.78	0.80
Recall	0.97	0.62	0.92
Area Under ROC	0.95	0.94	0.90
Kappa	0.73	0.58	0.70

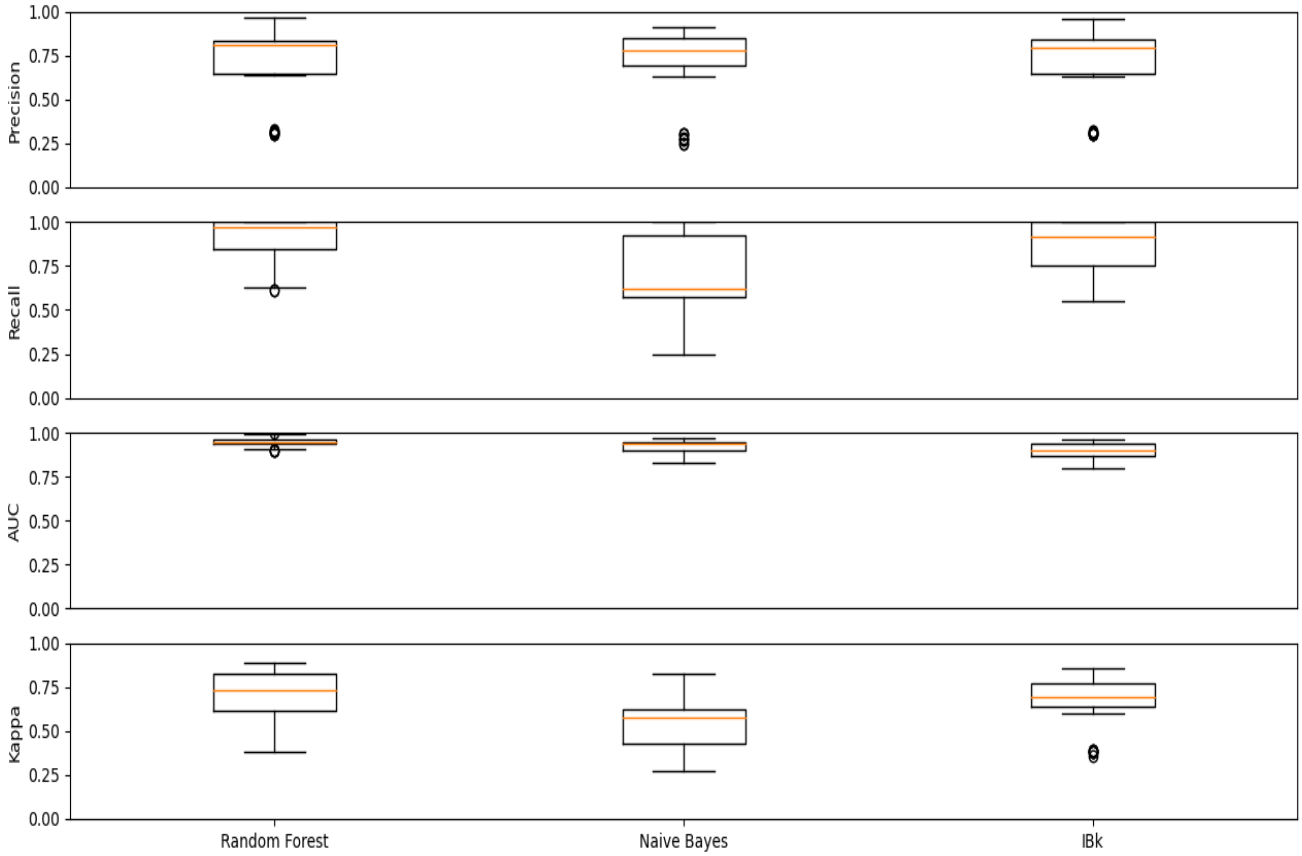

Figura 2: Boxplot classificatori (Bookkeeper)

Discutiamo ora come cambiano le performances al variare del metodo di feature selection utilizzato. La figura3 mostra i boxplot delle metriche di accuratezza in funzione del metodo di feature selection utilizzato.

Se l'obiettivo è massimizzare la Recall risulta essere nettamente vincente la scelta di effettuare feature selection. Riguardo le altre metriche, le differenze riscontrate tra le due alternative sono minime, comunque non effettuare feature selection sembra essere migliore se si vuole massimizzare la precision, altrimenti, secondo i dati ottenuti, risulta meglio applicare feature selection.

La tabella che segue mostra le mediane delle distribuzioni delle metriche di accuratezza in funzione del metodo di feature selection applicato.

	No Feature Selection	Best First
Precision	0.81	0.78
Recall	0.79	0.99
Area Under ROC	0.93	0.94
Kappa	0.64	0.68

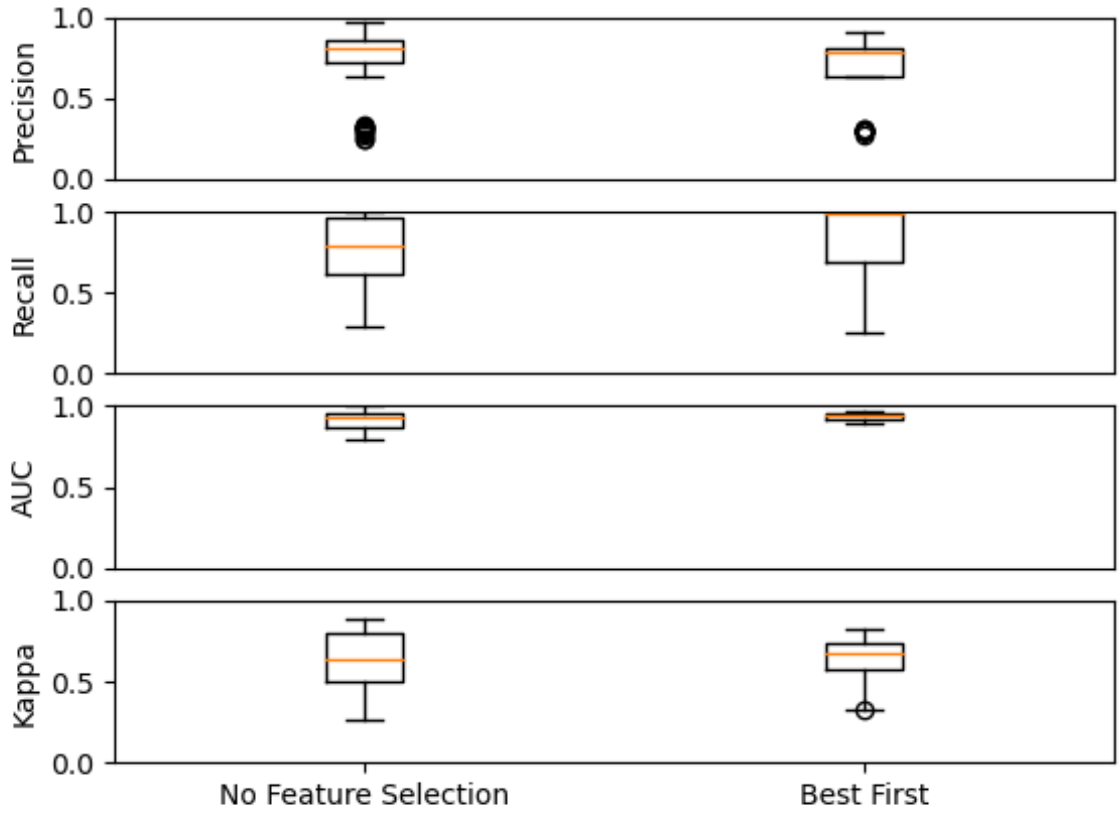

Figura 3: Boxplot metodi di feature selection (Bookkeeper)

Per quanto riguarda le tecniche di balancing del dataset, in riferimento alla figura 4 che mostra i boxplot delle distribuzioni delle metriche di accuratezza in funzione dei metodi di balancing, possiamo dire che: non bilanciare il dataset è la scelta migliore se si vuole massimizzare la Precision, tuttavia facendo ciò si minimizza la Recall; viceversa bilanciare il dataset tramite Undersampling massimizza la Recall, ma minimizza la Precision.

Oversampling e SMOTE rappresentano un compromesso quando non si vuole abbassare troppo né la Precision né la Recall; tra i due metodi, SMOTE risulta migliore su tutte le misure considerate.

NOTA: quando non si vuole abbassare troppo né la Precision né la Recall si lavora con l'intento di massimizzare la metrica di accuratezza F1 che però non è stata considerata in questa discussione; se si fosse considerata si sarebbe potuto osservare che, tra i metodi di bilanciamento del dataset considerati, SMOTE massimizza il valore di F1.

La tabella che segue mostra le mediane delle distribuzioni delle metriche di accuratezza in funzione del metodo di bilanciamento del dataset applicato.

	No Resample	Oversampling	Undersampling	SMOTE
Precision	0.81	0.79	0.76	0.80
Recall	0.80	0.84	0.94	0.88
AUC	0.94	0.94	0.94	0.94
Kappa	0.64	0.65	0.67	0.67

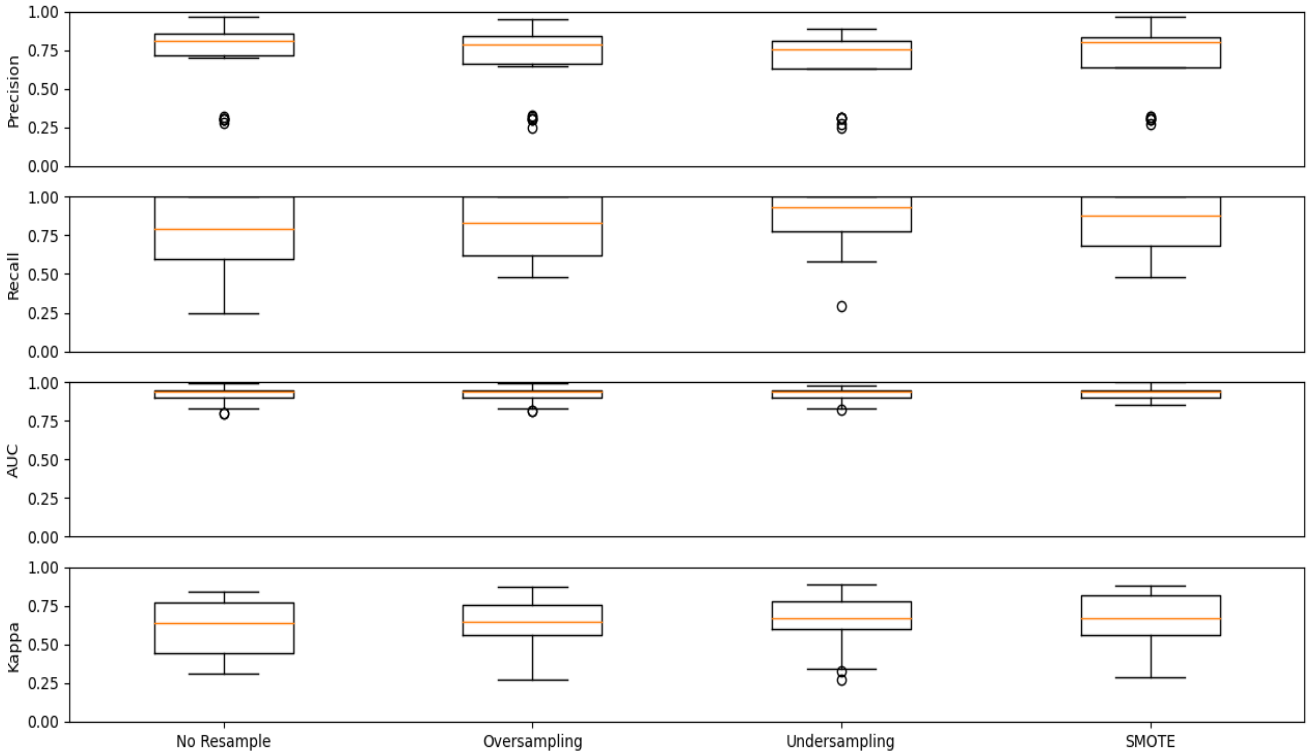

Figura 4: Boxplot metodi di balancing(Bookkeeper)

Applicare machine learning a dei dati può essere visto come una procedura composta da due fasi: la fase di preprocessing in cui i dati vengono manipolati e la fase di applicazione di un algoritmo di apprendimento automatico ai dati manipolati. La fase di preprocessing può comportare la manipolazione dei dati in molti modi, noi abbiamo trattato solo la feature selection ed il bilanciamento del dataset. Adesso vedremo quale preprocessing dei dati porta a valori delle metriche di accuratezza migliori per il progetto considerato, quindi Bookkeeper.

La figura 5 mostra i boxplot delle distribuzioni delle metriche di accuratezza in funzione del metodo di feature selection utilizzato e del metodo di balancing del dataset considerato.

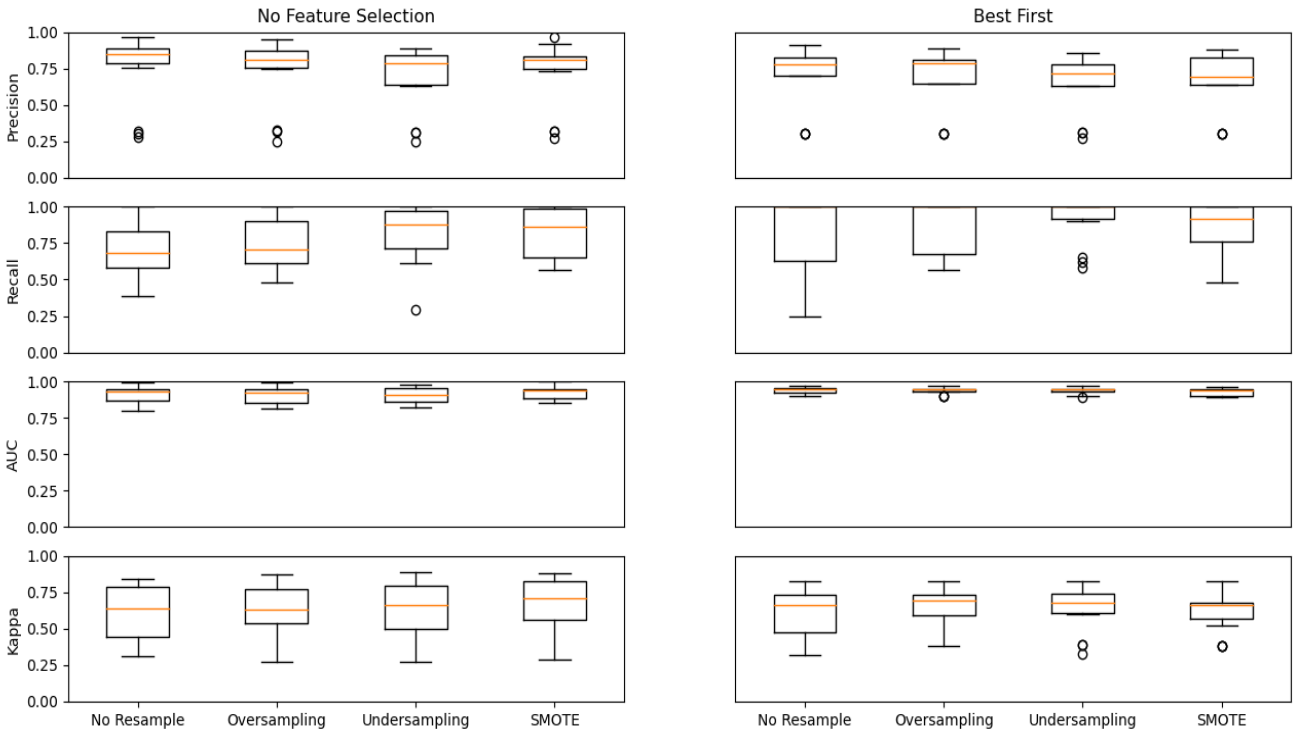

Figura 5: combinazione di metodi di balancing con metodi di feature selection (Bookkeeper)

Se l'obiettivo è massimizzare la Precision, i dati ci suggeriscono di adottare la combinazione "No Feature Selection"-"No Resample", mentre se l'obiettivo è massimizzare la Recall i dati ci suggeriscono di utilizzare la combinazione "Best_First"-"Undersampling", anche se questa combinazione presenta dei valori che sono outliers.

Per quanto riguarda la metrica AUC tutte le alternative di "Best First" hanno come mediana lo stesso valore che è di poco maggiore rispetto ai valori ottenuti senza applicare feature selection.

Infine, per quanto riguarda la metrica Kappa non c'è un'alternativa nettamente migliore delle altre, i dati mostrano che la combinazione "No Feature Selection"-"Smote" presenta la mediana maggiore (0.71) seguita dalla combinazione "Best-First"-"Undersampling" (0.69), con la combinazione "No Feature Selection"-"Undersampling" che possiede la mediana minore (0.63) tra tutte le alternative.

Quando si effettua machine learning tipicamente vogliamo massimizzare una certa metrica di accuratezza: per esempio se volessimo determinare l'infettività di una persona per metterla eventualmente in isolamento per 14giorni, potremmo voler massimizzare la Recall (accettiamo che qualcuno venga messo in isolamento inutilmente, ma evitiamo che vadano in giro persone infettive); bisogna allora tenere in mente che "a vincere" è tutto il processo di data mining e non le singole scelte migliori di ogni fase. A partire da ciò, la figura 6 mostra i boxplot delle distribuzioni delle metriche di accuratezza in funzione del metodo di balancing, del metodo di feature selection e del classificatore utilizzato.

Il colore blu indica Random Forest, il rosso indica Naive Bayes e il giallo indica IBk.

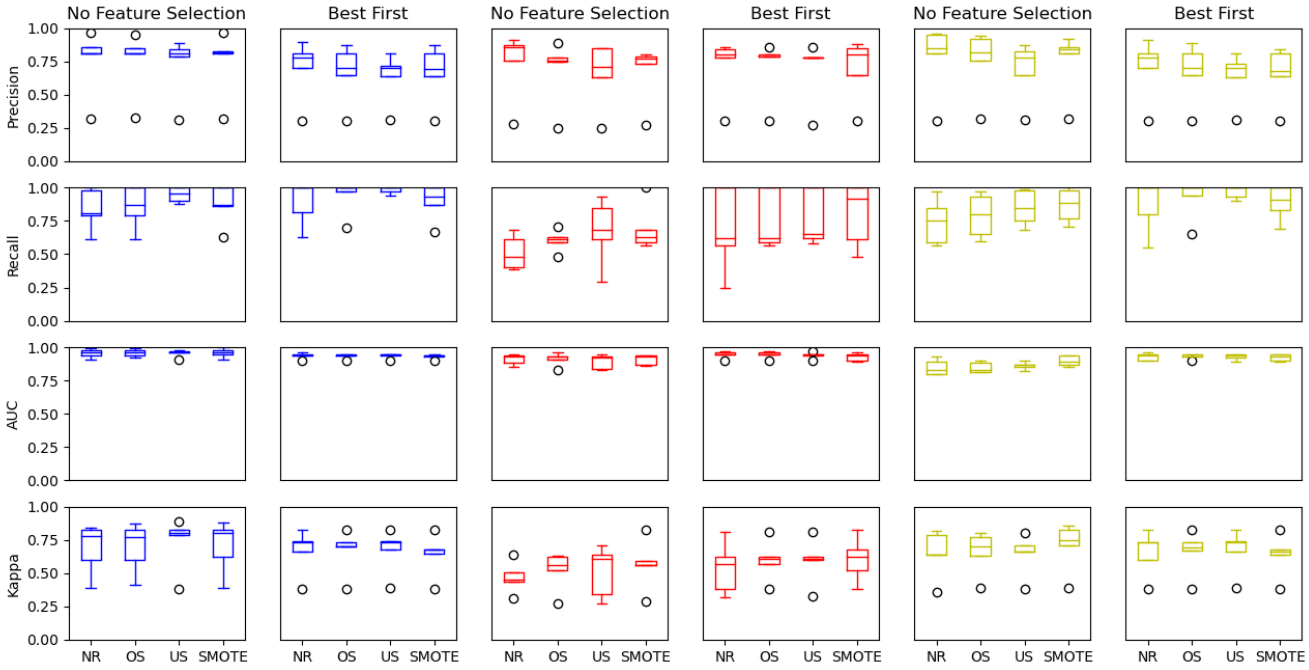

Figura 6: boxplot distribuzione delle metriche in funzione del metodo di feature selection, di balancing e del classificatore

Per massimizzare la Precision i dati ci suggeriscono di applicare la combinazione {Naive Bayes, No Feature Selection, No Resampling}, però questa combinazione minimizza la Recall tra tutte le alternative considerate.

Per massimizzare la Recall ci sono diverse alternative con mediana massima, cioè pari a 1, le alternative sono:

{Random Forest, Best First, No Resampling}
{Random Forest, Best First, Oversampling}
{Random Forest, Best First, Undersampling}
{IBk, Best First, No Resampling}

{IBk, Best First, Oversampling}
{IBk, Best First, Undersampling}

Per massimizzare AUC, i dati non ci forniscono una combinazione che sia nettamente migliore alle altre, tuttavia ci sconsigliano di utilizzare la combinazione {IBk, No Feature Selection, *} perché presenta valori di AUC inferiori a tutte le altre combinazioni che sono tra di loro equivalenti.

Infine, per massimizzare Kappa, i dati non ci forniscono una combinazione che sia nettamente superiore alle altre, tuttavia ci suggeriscono di utilizzare una combinazione del tipo {Random Forest, No Feature Selection, *} poiché le altre combinazioni mostrano risultati peggiori.

Syncope

Il dataset relativo al progetto Syncope è costruito considerando 63 delle 103 releases presenti su github e contiene i dati relativi a 622 dei 731 bugs presi da Jira.

Anche per Syncope iniziamo con l'osservare le performances dei diversi classificatori indipendentemente dal preprocessing dei dati effettuato. La figura 7 mostra i boxplot delle metriche di accuratezza in funzione del classificatore utilizzato.

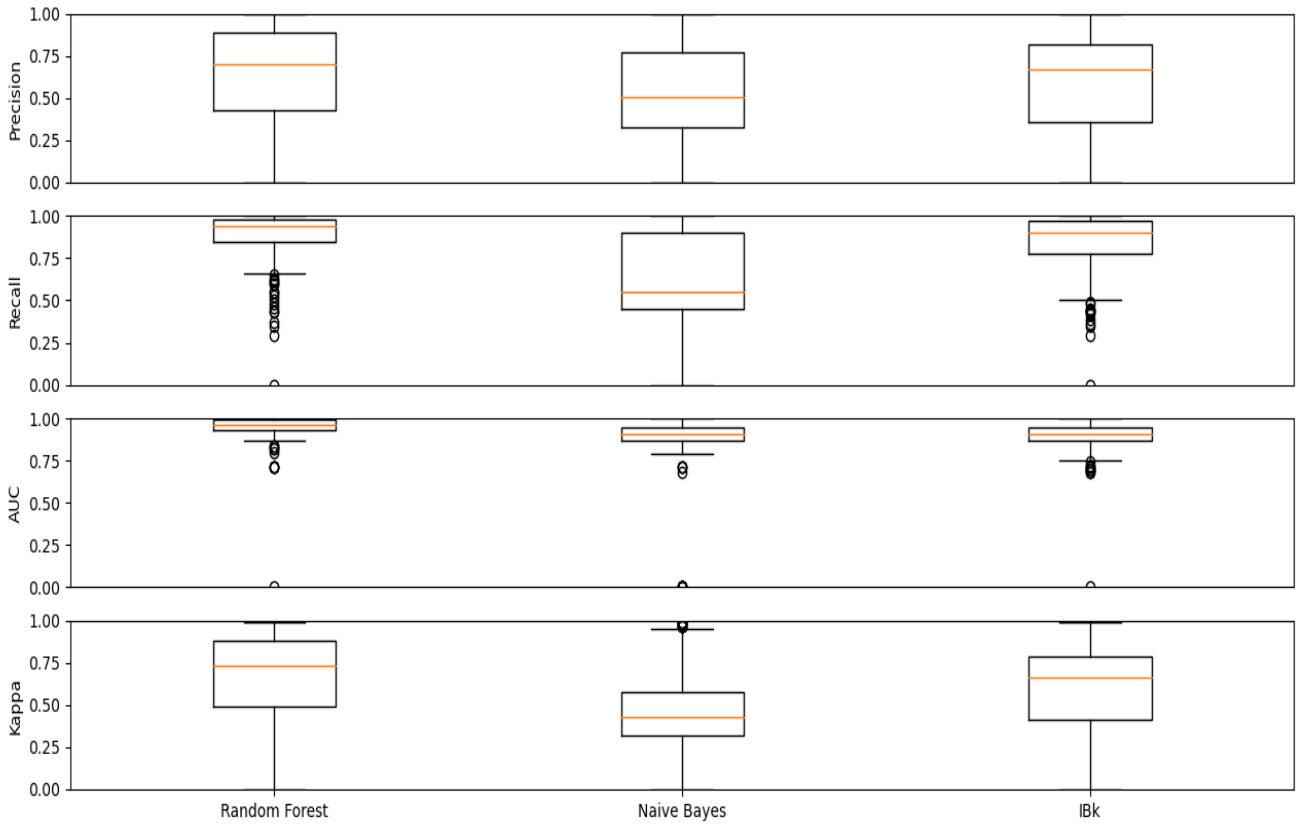

Figura 7: Boxplot classificatori (Syncope)

Dai risultati ottenuti Random Forest risulta essere il classificatore migliore indipendentemente dalla metrica di accuratezza che vogliamo massimizzare. Per quanto riguarda gli altri classificatori: IBk mostra prestazioni leggermente inferiori rispetto a Random Forest su tutte le metriche di accuratezza; Naive Bayes mostra prestazioni nettamente inferiori rispetto agli altri due classificatori su tutte le metriche di accuratezza tranne AUC.

La tabella seguente mostra le mediane delle distribuzioni delle metriche di accuratezza in funzione del classificatore utilizzato.

	Random Forest	Naive Bayes	IBk
Precision	0.70	0.51	0.67
Recall	0.94	0.55	0.90
Area Under ROC	0.96	0.91	0.91
Kappa	0.73	0.43	0.66

Discutiamo ora come cambiano le performances al variare del metodo di feature selection utilizzato. La figura 8 mostra i boxplot delle distribuzioni delle metriche di accuratezza in funzione del metodo di feature selection utilizzato.

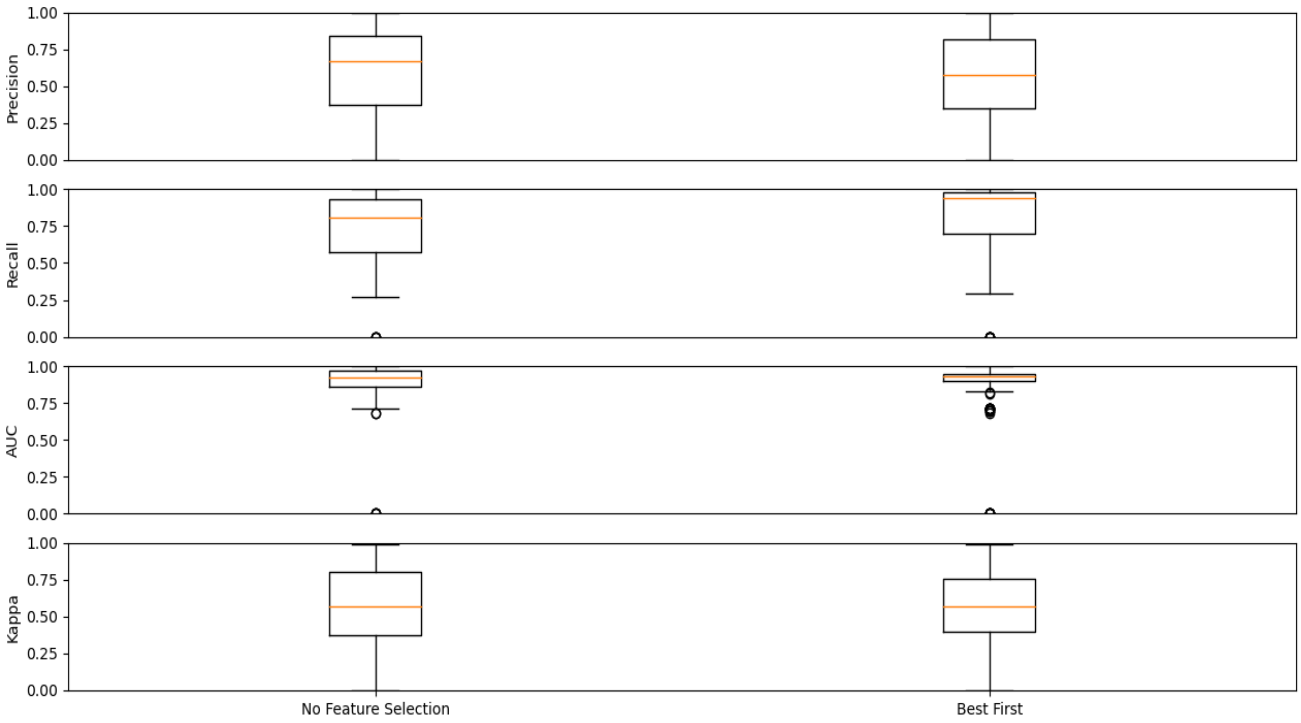

Figura 8: boxplot metodi di feature selection (Syncope)

Secondo i risultati ottenuti, se l'obiettivo è massimizzare la Precision allora è meglio non effettuare feature selection, sacrificando però la Recall, mentre se vogliamo massimizzare la Recall allora è meglio effettuare feature selection, sacrificando però la Precision. Per quanto riguarda invece AUC e Kappa, le due alternative, secondo i risultati ottenuti, sono equivalenti.

La tabella che segue mostra le mediane delle distribuzioni delle metriche di accuratezza in funzione del metodo di feature selection applicato.

	No Feature Selection	Best First
Precision	0.67	0.58
Recall	0.81	0.94
AUC	0.92	0.93
Kappa	0.57	0.57

Per quanto riguarda le tecniche di balancing del dataset, la figura 9 mostra i boxplot delle distribuzioni delle metriche di accuratezza in funzione dei metodi di balancing.

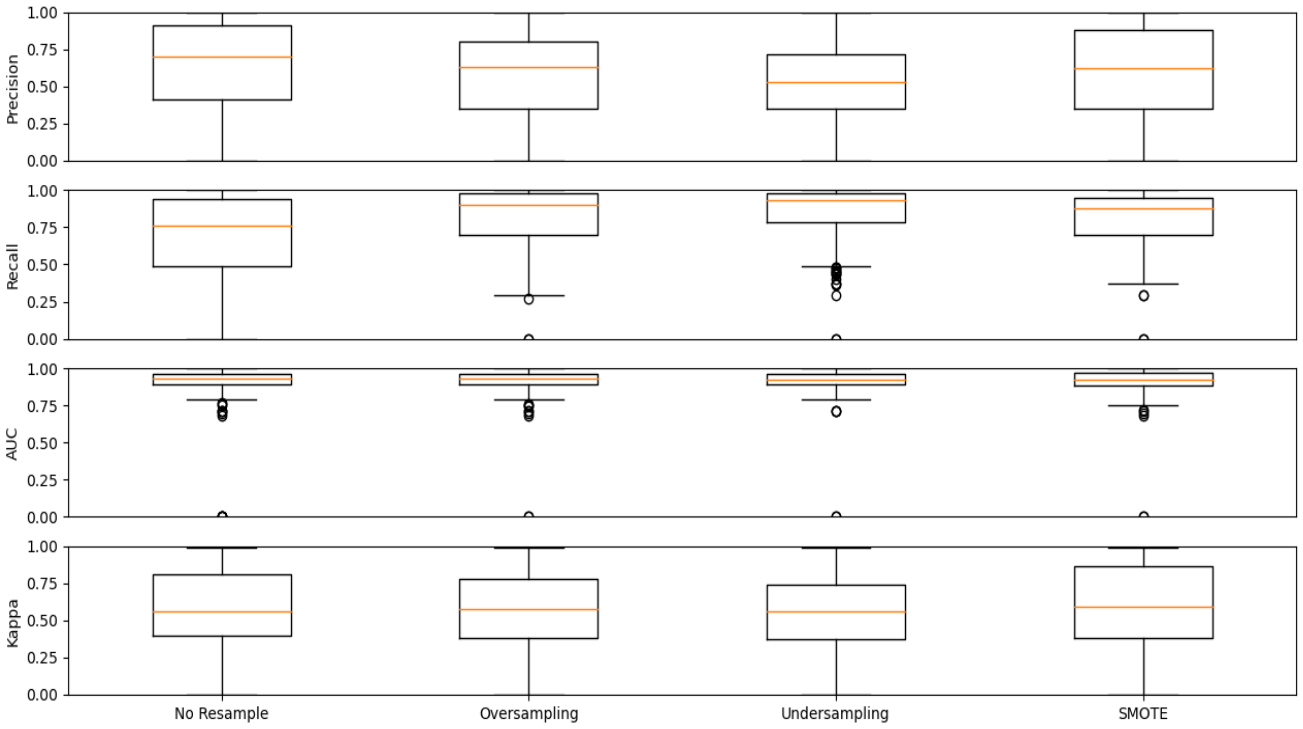

Figura 9: boxplot metodi di balancing (Syncope)

Se l'obiettivo è massimizzare la Precision, i dati ci suggeriscono di non effettuare il balancing del dataset, così facendo però andiamo contemporaneamente a minimizzare la Recall. Se l'obiettivo invece è massimizzare la Recall allora i dati ci suggeriscono di bilanciare il dataset effettuando Undersampling, però così facendo minimizziamo la Precision. Per quanto riguarda AUC tutte le alternative sembrano

essere equivalenti, mentre per quanto riguarda Kappa SMOTE sembra offrire prestazioni leggermente migliori.

Si noti che se non vogliamo massimizzare Precision o Recall a scapito dell'altra, allora dovremmo orientarci sui metodi di bilanciamento Oversampling e SMOTE i quali rappresentano un trade-off tra l'Undersampling (che massimizza la Recall e minimizza la Precision) e il "No Resample" (che massimizza la Precision e minimizza la Recall). Tra Oversampling e SMOTE sembra essere leggermente meglio optare per Oversampling.

La tabella che segue mostra le mediane delle distribuzioni delle metriche di accuratezza in funzione del metodo di bilanciamento del dataset applicato.

	No Resample	Oversampling	Undersampling	SMOTE
Precision	0.70	0.64	0.53	0.62
Recall	0.76	0.90	0.94	0.88
AUC	0.93	0.93	0.92	0.92
Kappa	0.56	0.58	0.56	0.60

Adesso discuteremo il preprocessing dei dati per il progetto Syncope, quindi nel nostro caso la combinazione di feature selection e balancing. La figura 10 mostra i boxplot delle distribuzioni delle metriche di accuratezza in funzione del metodo di feature selection e del metodo di balancing.

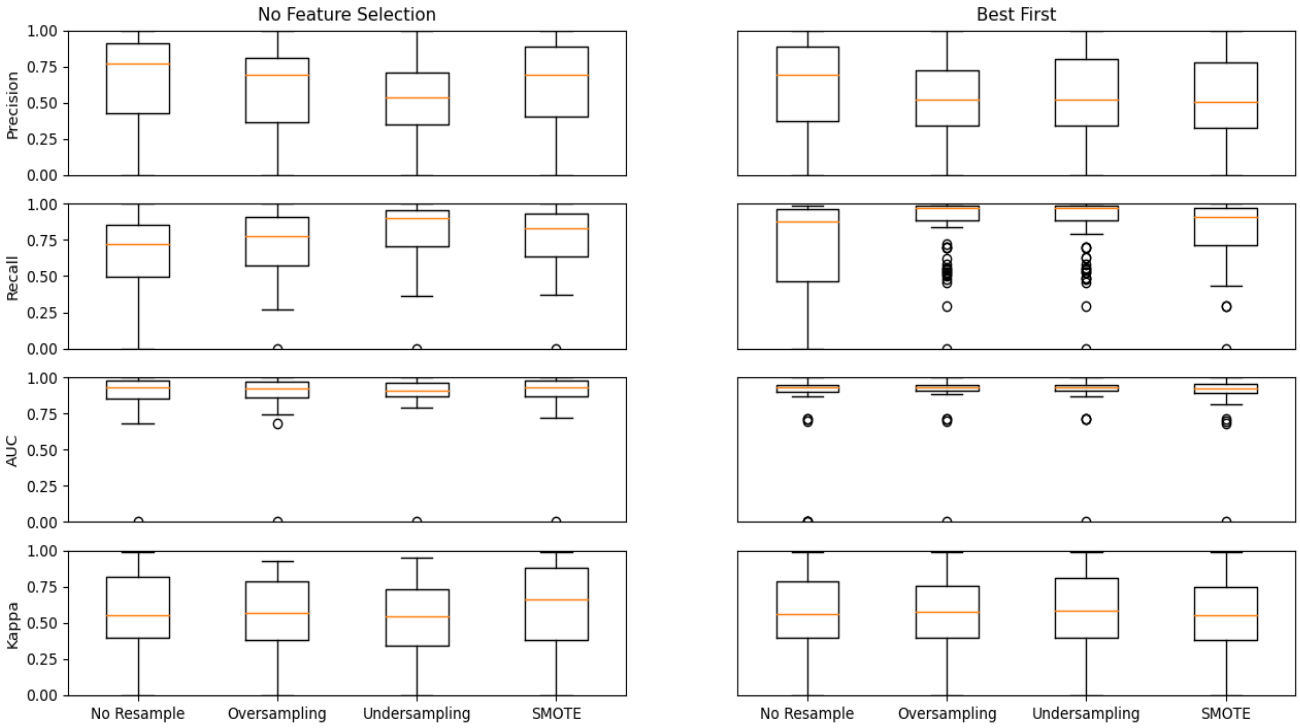

Figura 10: boxplot feature selection – balancing (Syncope)

Se l'obiettivo è massimizzare la Precision, i dati ci suggeriscono di adottare la combinazione "No Feature Selection"-"No Resample", mentre se l'obiettivo è massimizzare la Recall i dati ci suggeriscono di adottare la combinazione "Best First"-"Oversampling" o "Best First"-"Undersampling". Per quanto riguarda la metrica AUC, tutte le alternative sembrano equivalenti, mentre per quanto riguarda Kappa, i dati mostrano che la combinazione "No Feature Selection"-"Smote" è da preferire.

Discutiamo ora della distribuzione delle metriche di accuratezza in funzione del classificatore e del preprocessing applicato ai dati, dunque nel nostro caso in funzione del classificatore, del metodo di balancing e del metodo di feature selection. La figura 11 mostra i boxplot di queste distribuzioni.

Il colore blu indica Random Forest, il rosso indica Naive Bayes e il giallo indica IBk.

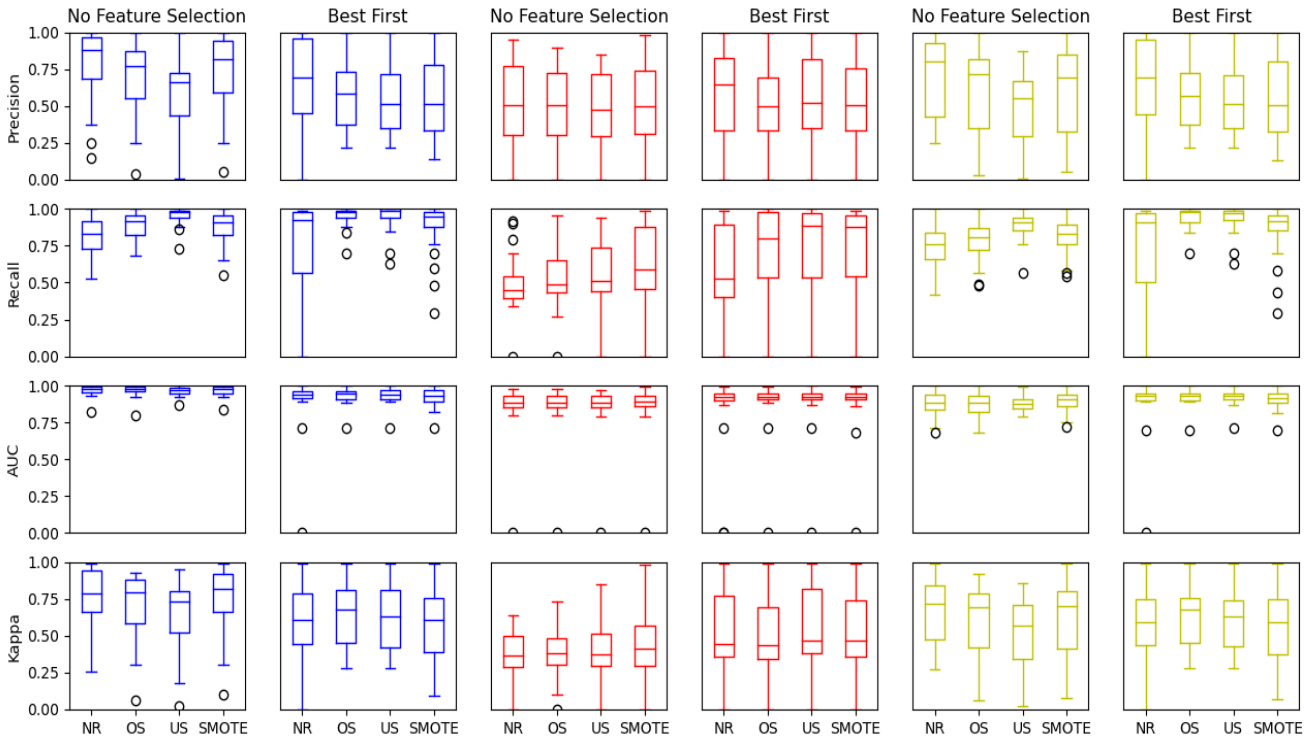

Figura 11: boxplot feature selection-balancing-classificatore

Per massimizzare la Precision i dati ci suggeriscono di applicare la combinazione {Random Forest, No Feature Selection, No Resampling}.

Per massimizzare la Recall non c'è una singola combinazione migliore delle altre, ma a pari merito abbiamo:

{Random Forest, No Feature Selection, Undersampling}
{Random Forest, Best First, Oversampling}
{Random Forest, Best First, Undersampling}
{IBk, Best First, Oversampling}
{IBk, Best First, Undersampling}

Per massimizzare AUC, i dati non ci consigliano una singola combinazione che sia migliore delle altre, ma ci consigliano di utilizzare un'alternativa della forma {Random Forest, No Feature Selection, *}.

Infine, per massimizzare Kappa, i dati ci suggeriscono di scegliere {Random Forest, No Feature Selection, Smote}, oppure sostituire a questa combinazione la scelta del metodo di balancing con "No Resample" o "Oversampling"; le altre alternative, particolarmente quelle che hanno Naive Bayes come classificatore, hanno distribuzioni di Kappa con mediane molto più basse della combinazione consigliata.

Riassunto e Confronto tra i due dataset

Confrontando i dataset possiamo osservare che:

- guardando solo al classificatore, in entrambi i casi risulta migliore Random Forest indipendentemente dalla metrica di accuratezza che si vuole massimizzare.
- guardando solo al metodo di feature selection, in entrambi i casi risulta migliore non effettuare feature selection se l'obiettivo è massimizzare la Precision, mentre risulta migliore effettuare feature selection se l'obiettivo è massimizzare la Recall. In entrambi i casi, su AUC e Kappa le due alternative sono quasi equivalenti.
- guardando solo al metodo di balancing, in entrambi i casi risulta migliore non effettuare balancing del dataset se vogliamo massimizzare la Precision a scapito della Recall e migliore Undersampling nel caso contrario. In entrambi i casi Oversampling e Smote rappresentano un trade-off tra i due estremi, ma in Bookkeeper Smote risulta migliore di Oversampling, mentre in Syncope succede il contrario.
- guardando a tutti i possibili modi in cui si può effettuare il preprocessing dei dati, dunque a tutte le combinazioni di feature selection e balancing considerate, possiamo dire che: in entrambi i casi per massimizzare la Precision risulta vincente la combinazione "No Feature Selection"-"No Resample", per quanto riguarda la Recall i risultati sono concordi sul fatto che la combinazione "Best First"-"Undersampling" possa essere un'ottima scelta, tuttavia i dati di Syncope suggeriscono che anche la combinazione "Best First"-"Oversampling"

permette di massimizzare la Recall. Per massimizzare AUC i dati di Bookkeper ci suggeriscono di utilizzare qualsiasi combinazione che abbia “Best First” come metodo di feature selection, mentre i dati di Syncope non evidenziano una combinazione migliore delle altre. Infine, per massimizzare Kappa, entrambi i dataset suggeriscono di utilizzare la combinazione “No Feature Selection”-“Smote”.

- guardando a tutti i possibili modi di fare machine learning considerati, per massimizzare la Precision i dati di Bookkeeper ci suggeriscono di utilizzare la combinazione {Naive Bayes, No Feature Selection, No Resampling}, mentre i dati di Syncope {Random Forest, No Feature Selection, No Resampling}. Per massimizzare la Recall, né i dati di Bookkeeper né quelli di Syncope suggeriscono un’unica combinazione migliore delle altre; tuttavia, si può osservare che l’intersezione delle combinazioni migliori suggerite da ciascun dataset non è vuota ed è composta dalle combinazioni: {Random Forest, Best First, Oversampling}, {Random Forest, Best First, Undersampling}, {IBk, Best First, Oversampling} e {IBk, Best First, Undersampling}; ciò potrebbe significare che queste combinazioni sono migliori indipendentemente da quale progetto apache si consideri (NB: il numero dei progetti analizzati è troppo basso per poter affermare questa cosa con un certo grado di sicurezza). Se l’obiettivo è massimizzare AUC, i dati di Bookkeeper non suggeriscono combinazioni migliori delle altre, mentre quelli di Syncope suggeriscono di utilizzare una combinazione della forma forma {Random Forest, No Feature Selection, *}. Infine, se l’obiettivo è massimizzare Kappa, i dati di Bookkeeper ci suggeriscono di utilizzare una combinazione della forma {Random Forest, No Feature Selection, *}; quelli di Syncope sono simili, ma non uguali, perché in Syncope la variante {Random Forest, No Feature Selection, Undersampling} non offre prestazioni peggiori delle altre varianti in {Random Forest, No Feature Selection, *}, che sono le migliori per questo progetto.

In conclusione, possiamo osservare che molte (ma non tutte) scelte ottime per un progetto lo sono anche per l’altro.

Considerazioni riguardo i risultati

Alcune considerazioni riguardo i risultati:

- È vero che in molti casi abbiamo osservato che le scelte ottime per un progetto lo sono anche per l’altro, tuttavia ciò non è sufficiente per poter dire con un certo grado di sicurezza che quelle scelte siano ottime per i progetti apache in generale, perché i progetti studiati sono solo 2; se avessimo studiato più progetti avremmo potuto trarre delle conclusioni in questo senso.

- I risultati mostrati sono stati ottenuti applicando quasi sempre le implementazioni standard dell'api Weka; tipicamente si effettua il tuning degli iperparametri, cioè fissato un metodo/modello si cercano quei parametri che massimizzano una certa metrica di accuratezza. Se avessimo effettuato il tuning degli iperparametri è probabile si sarebbe arrivati a dei risultati diversi.
- I risultati sono ottenuti partendo da un dataset non perfettamente generato: nella generazione del dataset le releases sono state ordinate in ordine cronologico, questo aspetto rappresenta una semplificazione in quanto non tiene conto di rami paralleli del grafo di sviluppo di un software. Per esempio, per il progetto Syncope se osservassimo un po' di releases in sequenza e il numero di classi contenute in ciascuna release vedremmo questa cosa:

  Release: 1.2.5 #Classes = 911
  Release: 1.2.6 #Classes = 912
  Release: 2.0.0-M1 #Classes = 1438
  Release: 1.2.7 #Classes = 914
  Release: 2.0.0-M2 #Classes = 1577
  Release: 1.2.8 #Classes = 915
  Release: 2.0.0-M3 #Classes = 1682
  Release: 2.0.0-M4 #Classes = 1711
  Release: 2.0.0.M5 #Classes = 1751
  Release: 2.0.0 #Classes = 1756
  Release: 1.2.9 #Classes = 916
  Release: 2.0.1 #Classes = 1758
  Release: 1.2.10 #Classes = 919
  Release: 2.0.2 #Classes = 1774
  Release: 2.0.3 #Classes = 1895

  è chiaro che la Release 1.2.9 segue la release 1.2.8 e non la release immediatamente precedente in ordine cronologico, cioè la 2.0.0, questo accade perché evidentemente le due release (1.2.9 e 2.0.0) appartengono a due rami paralleli del grafo di sviluppo. Questa cosa affligge il codice perché le revisioni per una release sono ottenute come le revisioni tra quella release e la release precedente; ciò è possibile tramite una particolare funzione della libreria *JGIT* che permette di ottenere i commits in un range ed il range specificato sono i commits relativi alle due releases che ci interessano. Ciò introduce un problema poi risolto: siccome le releases sono ordinate in ordine cronologico, quello che succede quando si elaborano le revisioni tra due releases che non sono realmente una precedente all'altra, ad esempio la 1.2.9 e la 2.0.0, è che molte classi modificate nei commits tra due release non

vengono trovate e dunque non è possibile tenere traccia delle modifiche a queste classi. Per risolvere questo problema e dunque per attenuare il problema che per semplicità si è considerato l'ordine cronologico delle release anziché considerare i rami di sviluppo separatamente, quello che si è fatto è stato (per ogni release) portare nella release successiva tutte le classi non eliminate della release attuale. In questo modo la release 1.2.9 non conterrà solo le 916 classi realmente presenti, ma anche tutte quelle classi non esplicitamente eliminate che erano presenti nella 2.0.0; quando poi si elaborano le differenze tra la 1.2.9 e la 2.0.1 il numero di classi modificate nei commits ma non trovate si riduce drasticamente (in molti casi si azzera) e quindi è possibile tenere traccia delle modifiche a queste classi. Si noti che per la produzione del dataset le classi che artificialmente sono state aggiunte alle release con questo meccanismo riparatorio non sono considerate, cioè relativamente alla release 1.2.9 nel dataset sono presenti 916 classi e non queste 916 classi più tutte quelle classi non esplicitamente eliminate che erano presenti nella 2.0.0, ciò è reso possibile dal token citato nel punto 8 dell'implementazione.

Links

Deliverable 1

- https://github.com/FabianoVeglianti/ISW2_deliverable1_0286057
- https://travis-ci.com/github/FabianoVeglianti/ISW2_deliverable1_0286057
- https://sonarcloud.io/dashboard?id=fabianoveglianti_isw2_deliverable1_0286057

Deliverable 2

- https://github.com/FabianoVeglianti/ISW2_deliverable2_0286057
- https://travis-ci.com/github/FabianoVeglianti/ISW2_deliverable2_0286057
- https://sonarcloud.io/dashboard?id=fabianoveglianti_isw2_deliverable2_0286057