RELAZIONE PROGETTO ISW2 ANNO 2019/2020

Deliverable del Corso - Prof. Falessi

Autore:

Alessio Mazzola, Matricola: 0279323

Indice

Capitolo 1

Introduzione

Lo scopo dei Deliverable assegnati durante il corso è stato quello di analizzare due progetti open-source, in particolare BOOKKEEPER (in comune con i colleghi) e TAJO andando successivamente ad effettuare misurazioni e sfruttando i risultati ottenuti per l'utilizzo di Weka. i Deliverable sono:

- Deliverable 1
- Deliverable 2, divisa in due diverse milestone.

Per reperire le informazioni necessarie al fine di eseguire le misurazioni, sfruttiamo la suite offerta da JIRA, il framework GIT e la suite offerta da Weka.

Jira risulta essere una suite per il tracciamento delle segnalazioni (identificate attraverso il nominativo "Ticket") che consente il Bug Tracking e la gestione dei progetti in modalità agile. GIT è invece un software di version control, sfruttato dagli sviluppatori degli applicativi che permette a questi di lavorare sull'applicazione e tenere traccia delle modifiche riportate ad ogni nuovo commit. Weka è invece un software che fornisce strumenti per l'implementazione di algoritmi di Machine Learning.

L'importanza dell'analisi del Software può essere riassunta in quattro aspetti fondamentali:

- Quality Improvment: Cerca di rispondere alla domanda "Come prevenire Bug dell'applicazione nel futuro?"
- Product Improvment: Cerca di rispondere alla domanda "Come la continua integrazione del software impatta la produttività del team?"
- Process Improvment: Cerca di rispondere alla domanda "Come le pratiche di revisione del codice impattano la qualità del software?"
- Empirical Theroy Building: E' una teoria di Software Quality che punta a stimare la relazione tra Sforzo e Costo durante lo sviluppo dell'applicativo.

Capitolo 2

Deliverable 1

2.1 Scopo

Lo scopo del primo Deliverable è quello di andare a misurare la stabilità di un attributo del progetto considerato[1]. Nella slide del professore "Falessi Merging JIRA with GIT" è riportato l'algoritmo per la scelta dell'attributo da analizzare. Poiché la lettera M nell'alfabeto è associata al numero 11 (tenendo conto dell'alfabeto Italiano) allora ricaviamo :

$$11 \mod 3 = 2 \tag{2.1}$$

L'attributo considerato all'interno della milestone sarà il numero di Fixed New Feature. La necessità di eseguire un "Merge" tra le informazioni contenute all'interno di Jira e quelle contenute in Git deriva dal fatto che la sole informazioni di Git non risultano essere affidabile e, la maggior parte delle volte, non risultano essere esplicative dell'effettivo cambiamento effettuato. Come abbiamo visto durante il corso, inoltre, dovrebbe esserci una relazione 1-1 tra un ticket di Jira ed un Commit eseguito con Git, in modo tale da poter tenere traccia dei cambiamenti con il passare del tempo, ma questo solitamente non viene rispettato.

2.2 Progettazione

In questa sezione andiamo a presentare la deliverable nella sua totalità, andando a spiegare il contesto di utilizzo delle variabili ed il funzionamento dei metodi che hanno permesso di arrivare al risultato finale.

[1]Nella pagina web dei progetti su Jira, quello che è stato considerato tenendo conto dell'algoritmo è il progetto open-source MAHOUT

2.2.1 Classi

La deliverable1 presenta tre package principali, individuati da:

- engine : Package rappresentante il cuore del progetto, contiene infatti le classi principali adibite al reperimento dei ticket associati alle New Feature. Al suo interno troviamo le seguenti classi:
  - DonwloadCommit
  - RetrieveTicket
  - Searcher
- external : Questo package continene un'unica classe Runner che ha il compito di eseguire l'intero progetto.
- writer: Questo package contiene un'unica classe PropertiesWriter, necessaria al fine di scrivere sul file "config.properties" che sfruttiamo per avere riferimento delle posizioni dei file all'interno del progetto al fine di essere richiamati quando se ne ha bisogno.

–DownloadCommit.class–

Il metodo principale della classe prende il nome di getAllCommits(). Sfruttando la dipendenza di JGit permettendo di interfacciare Git con Java, va ad eseguire il clone del progetto considerato, se questo non dovesse già esistere nella directory specificata. Successivamente va a reperire tutti i commit del progetto e li scriviamo all'interno del file *commits.txt*. Prima di eseguire la clonazione del progetto, si verifica sempre se questo già esiste (cioè se già è stato clonato in precedenza), altrimenti viene eseguita la clonazione. Una volta che il progetto è stato clonato, sfruttando il framework JGit lo analizziamo e tentiamo di ricavare tutti i commit che sono stati eseguiti, andando a scriverli all'interno del file che sfruttiamo successivamente nelle altre classi per compiere le successive operazioni necessarie al retrieve dei Ticket.

–RetrieveTicket.class–

Questa classe ha il compito di interfacciarsi con il sistema Jira e di eseguire una query per andare a scaricare tutti i ticket di nostro interesse. Questo comportamento è modellato attraverso il metodo pubblico retrieve. Poiché abbiamo bisogno delle "New Feature", allora la query che andremo a specificare sarà la seguente:

```
String urlFixedNewFeature = "https://issues.apache.org/
    jira/rest/api/2/search?jql=project=%22" + proj + "%22
    AND%22issueType%22=%22New%20Feature%22AND(%22status
    %22=%22closed%22OR"+ "%22status%22=%22resolved%22)AND
    %22resolution%22=%22fixed%22&fields=key,resolutiondate,
    versions,created&startAt="+ i.toString() + "&maxResults
    =" + j.toString();
```

La variabile "proj" è identificativa del progetto preso in considerazione. La query risulta quindi essere *modulare*. Inoltre un altro aspetto importante sta nella definizione di "maxResults", questo perché altrimenti il sistema Jira restituirebbe solamente 50 risultati, mentre in questo modo li stiamo prendendo una quantità identificata dalla variabile "j".
Successivamente andiamo ad analizzare l'istanza JSONObject ricavata attraverso la query, andando ad estrapolare le informazioni a noi necessarie, cioè il nomi dei ticket che vengono ricavati attraverso la query. Infine queste informazioni vengono scritte all'interno del file "results.csv".

–Searcher.class–

Questa è la classe adibita al confronto tra i commit ricavati attraverso la clone del progetto tramite JGit ed i ticket scaricati attraverso la query eseguita sul sistema Jira. In particolare questa classe possiede al suo interno diversi metodi, tra cui:

- lastIssue(): Questo metodo restituisce in output il file *"NewCommits.txt"* che rappresenta una versione del file *"Commits.txt"* alla quale abbiamo eliminiato le parti superflue. In particolare otteniamo un risultato del tipo [Data,Ticket]. Poiché per la pratica dei commit non esiste un vero e proprio standard, alcuni di questi durante questo processo non saranno stati considerati.

- removeElements(): Questo metodo sfrutta l'output del metodo lastIssue(), quindi *"NewCommits.txt"*, e va ad eliminare ancora i dati superflui, in modo tale da avere ad ogni riga dispari la data del ticket e ad ogni riga pari il ticket stesso associato a quella data. Il file di output prende il nome di *"FinalCommits.txt*

- createTicketCSV(): Questo metodo crea una versione CSV di *"FinalCommits.txt* che prende il nome di *finRes.csv*

- removeSpacesCSV(); Questo metodo sfrutta il file *finRes.csv* e crea a sua volta un nuovo file chiamato *finRes2.csv*, che risulta essere un file "Normalizzato" al fine di essere comparato successivamente. Con Normalizzazione intendiamo l'eliminazione di elementi che non fanno parte del ticket stesso, come Spazi, Punti, ecc..

- finalResults(ArrayList<String[]> list): Questo metodo restituisce l'intersezione tra i Ticket presenti nel file *finRes2.csv* ed i ticket che abbiamo ricavato attraverso la query di Jira. In particolare questa intersezione risulta essere fondamentale poiché solamente attraverso JGit non possiamo sapere la natura del ticket considerato, mentre con l'intersezione della query di Jira, siamo consapevoli del fatto che i ticket "sopravvissuti" saranno quelli inerenti ad una New Feature. Questo output è scritto all'interno del file *risultatiFinali.csv*

- countOccurrences(Integer[] dates): Questo metodo restituisce infine il numero di ticket per una determinata combinazione mese/anno. Il file preso in considerazione è direttamente proveniente dal metodo finalResult(...) e viene scritto l'output all'interno del file *counterTicket.csv*.

2.3 Process Control Chart & Commenti

Presentiamo adesso il *Process Control Chart* ottenuto andando ad analizzare il file *counterTicket.csv*.

Dal grafico (Fig 2.1) notiamo come il numero di commit risulta essere compreso, per la maggior parte del tempo, tra il *Lower Limit* (impostato a 0[2]) e l'*Upper Limit*. Notiamo come nel periodo temporale che va dal 2009 al 2010 e nel periodo temporale che va dal 2011 al 2012, è stato superato l'*Upper Limit*, questo significa che, effettivamente, in quei periodi c'è stata una affluenza maggiore di commit inerenti le *New Feature*. Probabilmente questo fenomeno si è manifestato in quanto potrebbe essere stata presa in considerazione l'idea di pubblicare le New Feature solamente ad una nuova versione del progetto (Quindi ad una Major Release) piuttosto che eseguire aggionramenti man mano che queste New Feature venivano create.

Ci accorgiamo invece che per tutto il 2014 e parte del 2015, il numero di commit inerenti *New Feature* è pari a 0. Questo potrebbe significare che in quell'arco di tempo si è lavorato solamente sulla risoluzione di Bug dell'applicazione, piuttosto che all'aggiunta di nuove funzionalità.

Il risultato ottenuto è comunque da intendersi come "approssimativo", infatti non è da escludere che tutte le *New Feature* ricavante non rappresentino la totalità delle *New Feature* esistenti nel progetto. Possiamo fare questa affermazione in quanto non si ha un vero standard dei commenti quando viene effettuato un commit (Sia esso inerente ad una New Feature o ad altre problematiche). Avendo inoltre generato un sistema *"Modulare"*, cioè quindi applicabile a diversi progetti, cercando di adattarsi a questi ultimi, è

[2]Il valore di LCL è negativo, ma imponiamo a 0 il Lower Limit in quanto non è possibile avere un numero di commit negativo.

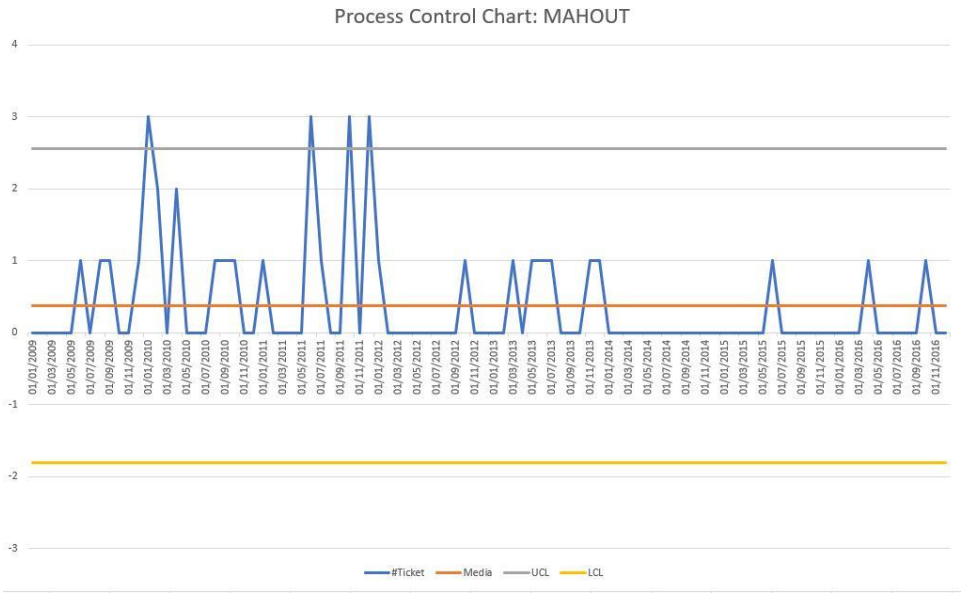

Figura 2.1: Process Control Chart
Di seguito vengono riportati anche i valori specifici:
Media: 0,375 Dev.Standard: 0,729095 UCL: 2,562285 LCL: -1,81229

normale che alcuni di questi commit siano stati scartati proprio perché non conformi agli standard presi come riferimento.
Bisogna infine considerare che molti commit non sono associati a nessun ticket, quindi questi saranno direttamente da scartare.

Capitolo 3

Deliverable 2 Milestone 1

3.1 Scopo

Lo scopo di questo deliverable è quello di generare come output un file CSV contenente le classi considerate del progetto, la versione di riferimento del progetto, le metriche calcolate su queste classi ed infine la determinazione del campo "Buggy" che identifica la difettività della classe nella specifica release del progetto. Per il calcolo di questo campo è stato utilizzato il metodo Proportion, questo poiché non è sempre possibile reperire l'IV da Jira, quindi l'intuizione ci ha portato a credere che ci sia una proporzione tra la versione nella quale si identifica un difetto e la versione in cui questo difetto viene effettivamente risolto. Per poter comprendere meglio il concetto risulta necessario definire la Fixed Version (FV), Introduction Version (IV) ed Opening Version (OV). In particolare possiamo dire:

- FV: Risulta essere la versione del progetto in cui il bug è stato definitivamente risolto, quindi in questa versione esiste un commit risolutivo associato al ticket generato precedentemente.
- IV: Risulta essere la "Più vecchia" versione tra le Affected Version, cioè tra le versioni del progetto che presentano quel determinato bug al loro interno. Con "Più vecchia" andiamo proprio ad intendere il significato temporale, in quanto la versione che introduce il bug sarà sicuramente più anziana rispetto ad altre versioni dove lo stesso bug esiste ma non è stato ancora risolto.
- OV: Risulta essere la versione in cui ci si è accorti della presenza del bug e si è creato un ticket su Jira. In generale andremo a prendere la data di questo ticket e la utilizzeremo come Opening Version.

Andiamo inoltre a definire il concetto di Affected Version (AV) come versione/i del progetto che risulta essere affetta da un determinato Bug. Ogni Bug può infatti colpire diverse versioni, quindi in generale l'Affected

Version non è unica. Poiché comunque non esiste un vero e proprio standard per queste situazioni, molte volte, durante la creazione del ticket su Jira, queste informazioni non vengono inserite, rendendo quindi il ticket inutilizzabile nell'esecuzione delle misurazioni.

3.2 Progettazione

In questa sezione andiamo a presentare i Package e le classi presenti all'intendo del progetto, andando a spiegare il contesto di utilizzo delle variabili ed il funzionamento dei metodi che hanno permesso di arrivare al risultato finale.

3.2.1 Classi

Il progetto presenta tre package principali, individuati da:

- engine : Questo package contiene al suo interno le classi necessarie all'esecuzione dei confronti con gli elementi ricavati da JGit e gli elementi ricavati attraverso le RestApi di Jiira. All'interno del Package sono presenti le seguenti classi:
  - CreateCSVPath
  - GetDefectedClasses
  - GitBlameWithJava
  - MetricsCalc
  - VersionDivisor
- restandgit: Questo Package contiene al suo interno le classi necessarie per interfacciarsi con Jira e con Git. In particolare per Jira sfrutteremo le RestApi messe a disposizione dal sistema stesso, mentre per Git andremo ad utilizzare la libreria JGit. All'interno del Package sono presenti le seguenti classi:
  - AssociationJiraGit
  - DownloadCommit
  - GetAffectedVersionFromJira
  - GetReleaseInfo
  - RetrieveTicketID
- writer: Questo package contiene al suo interno le classi necessarie alla scrittura dei file. In partciolare sono presenti le classi:

– OutputWriterMilestoneUnoDue: Questa classe è necessaria al fine di ricavare il CSV di output di questa milestone. Le colonne di questo CSV sono le seguenti:

["Version","Class","NFix","NRevision","NAuthor","LocAdded","LocDeleted" "LocTouched","MaxLockAdded","AverageLockAdded","Size","Age","Buggy"]

– PropertiesWriter: Questa classe è necessaria al fine di mantenere aggiornato il file *config.properties*[1] che sfruttiamo per avere riferimento delle posizioni dei file all'interno del progetto al fine di essere richiamati quando se ne ha bisogno.

–GetDefectedClasses.class–

Questa classe ha lo scopo di andare a determinare se una determinata classe può essere considerata "Buggy" o meno. Per eseguire questa misurazione viene inizialmente eseguito il controllo sui ticket disponibili ricavati da Jira dei quali si è riuscito a ricavare una associazione con i commit ottenuti tramite JGit. Se invece le infromazioni non dovessero essere disponibili viene sfruttato il metodo *Proportion*, in particolare questo metodo viene sfruttato quando non si hanno dati forniti da *Jira* per la classe presa in considerazione nella determinata release.Il valore di P risulta essere individuato dalla seguente formula:

$$P = \frac{FV - IV}{FV - OV} \tag{3.1}$$

Da questa sarà quindi possibile andare a calcolare il Predicted IV, che sarà individuato da:

$$IV = FV - (FV - OV) * P \tag{3.2}$$

Per il progetto, la variante di calcolo per il valore di P è *Increment*, cioè che P viene calcolato come la media tra i Fixed Defects nelle versioni precedenti del progetto.

Per la determinazione delle classi Buggy, viene quindi invocato il metodo *determinDefectiveWithProportion(double p)*, il quale ho il compito di andare a determinare se una classe è defective o meno per una release di riferimento sfruttando il metodo Proportion.

[1]Si ha un file properties per ogni progetto analizzato.

–MetricsCalc.class–

Questa classe ha lo scopo di andare a calcolare le metriche sulla totalità delle classi presenti all'interno del progetto preso in considerazione. Le metriche che sono state calcolate sono le seguenti:

- Size: Grandezza espressa in Linee di codice per una determinata Classe.
- LOC Touched: Somma sulle revisioni del progetto delle linee di codice aggiunte ed eliminate per una determinata classe
- NR: Numero di revisioni effettuate sulla classe
- NFix: Numero di BugFix effettuate sulla classe
- Nauth: Numero di Autori che hanno contribuito alla realizzazione della classe.
- LOC Added: Somma sulla revsioni delle linee di codice aggiunte per la classe considerata.
- *LOC Deleted*: Somma sulla revisioni delle linee di codice eliminate per la classe considerata. Questa è stata una metricha ulteriore aggiunta. La motivazione che ha spinto ad aggiungerla è la seguente: La metrica LOC Touched non è completamente esplicativa, in quanto potrebbe avere un valore elevato anche solo perché ci sono state molte linee di codice aggiunte e poche eliminate. Personalmente ritengo che le linee di codice eliminate siano quindi una metrica importante nella determinazione della difettività, in quanto più queste sono elevete, e maggiore potrebbe essere la possibilità che nella classe considerata sia stato eliminato qualcosa di non corretto (quindi un possibile Bug). Possiamo quindi pensare che nelle release precedenti, la classe che nella attuale release ha avuto questa eliminazione di linee di codice, potesse presentare con più probabilità una difettività. Questo risulta avere un senso se considerato assieme alle atre metriche.
- MAX LOC Added: Massimo sulle revisioni di linee di codice aggiunte per la classe considerata.
- AVG LOC Added: Media sulle revisioni di linee di codice aggiunte per la classe considerata.
- Age: Età della classe espressa in settimane rispetto alla versione del progetto considerata.

Capitolo 4

Deliverable 2 Milestone 2

4.1 Scopo

Nella seconda Milestone sviluppata sfruttiamo i risultati ottenuti dalla milestone precedente, quindi il file CSV contenente le classi dei progetti analizzati con le metriche da noi calcolate, e le sfruttiamo utilizzando il Framework *Weka* per andare ad eseguire delle misurazioni statistiche che ci permettano di comprendere la bontà del nostro output.

Ponendo la nostra attenzione inizialmente su *Weka*, questo risulta essere un framework adibito al Machine Learning e viene utilizzato per andare ad ottimizzare i *Criteri di Performance* utilizzando dati collezionati da esperienze passate, quindi sfruttando l'acquisizione di esperienza per arrivare a risultati sempre migliori. Nel Machine Learning sia la Statistica che l'Informatica posseggono due ruoli fondamentali in quanto:

- Statistica: Permette l'inferenza, cioè il procedimento di generalizzazione dei risultati ottenuti attraverso una rilevazione, andando ad analizzare un campione.
- Informatica: Offre strumenti ed *Algoritmi* efficienti per risolvere i problemi di ottimizzazione e per andare a rapprensentare e valutare il modello per l'inferenza.

4.2 Tecniche di Valutazione

La valutazione risulta essere la chiave per il successo, questa infatti permette di comprendere quanto effettivamente risulta essere predittivo il modello che abbiamo preso in considerazione. E' inoltre necessario tenere conto della presenza dell'*Errore* all'interno dei dati di *Training*, dovuto al fatto che, appunto, un set di dati non potrà mai essere "perfetto". Una soluzione semplice, da noi adottata, è quella di andare a dividere i dati ricavati

in *Training* e *Testing*. Oltre a questo, però, dobbiamo necessariamente utilizzare delle tecniche più sofisticate di analisi, altrimenti i nostri risultati risulterebbero essere limitati. Per questo scopo possiamo definire il *Test Set* come un insieme di instanze tra loro indipendenti che non hanno nessun ruolo nella formazione del *Classificatore*. Non dobbiamo inoltre dimenticarci dell'esistenza dell'*Error rate*, cioè la proporzione ottenuta dagli errori eseguiti sull'intero set di Istanze che sono state analizzate dal modello. In generale possiamo dire che si ha *Successo* quando l'istanza presa in considerazione è predetta correttamente, mentre si ha errore se il risultato della predizione non è corretto.

Una volta che la valutazione risulta essere terminata, i dati possono essere utilizzati per la costruzione del *Classificatore*. Generalmente più grande e più "adatto" sarà il *Training Set*, migliori saranno le sue predizioni. Lo stesso ragionamento può essere ripetuto con il *Testing Set*, il quale più è grande e maggiore sarà l'accuratezza dell'errore stimato.

In base alla procedura che viene utilizzata nella divisione dei dati in *Training Set* e *Testing Set*, si avranno risultati completamente differenti da parte dell'analisi del Classificatore.

Per il progetto realizzato, è stata utilizzata una tecnica di validazione di tipo *Time Series*. La caratteristica principale di queste tecniche sta nel fatto di preservare l'ordine temporale dei dati. E' stato possibile sfruttare questo tipo di validazione in quando il dataset ricavato dal Deliverable 2 Milestone 1 era ordinato temporalmente.
La tecnica di validazione utilizzata prende il nome di *Walk Forward*. Per l'utilizzo della tecnica il dataset viene diviso in più parti ordinate cronologicamente. Ad ogni esecuzione tutti i dati disponibili prima*Testing Set* vengono utilizzati come *Training Set*. L'accuratezza del modello è calcolata in base alla media dell'accuratezza calcolata durante ogni esecuzione dello stesso.

4.3 Sampling

Le tecniche di Sampling sono state introdotte al fine di bilanciare il Training Set ed il Testing Set per i classificatori. Prendendo in considerazione il dataset in nostro possesso, è normale aspettarsi uno sbilanciamento tra il numero di classi che sono difettive e quelle che non lo sono. A questo scopo vengono introdotte tre diversi metodi per eseguire il *Sampling*:

- UnderSampling: Con questa tecnica di sampling si va ad estrarre, dalla classe maggioritaria, un numero di istanze pari a quelle della classe minoritaria. Nel progetto, per l'applicazione dell'Undersampling abbiamo utilizzato l'oggetto Resample con le opzioni : -M, 1.0

- OverSampling: Con questa tecnica di sampling si va ad estrarre, dalla classe minoritaria, un numero di istanze fino a che non si raggiungerà la cardinalità della classe maggioritaria. Nel progetto, per l'applicazione dell'Oversampling abbiamo utilizzato l'oggetto Resample di Weka con le opzioni : -B, 1.0, -Z, 2**Percentuale_classe_maggioritaria*[1]
- SMOTE: La tecnica *SMOTE* risulta essere un raffinamento dell'Oversampling, in quanto l'estensione della classe minoritaria non avviene con dati ripetuti presi all'interno del dataset iniziale, ma andando a crearne di nuovi in maniera "Sintetica". Nel progetto, per l'utilizzo di SMOTE, è stato necessario importare una libreria aggiuntiva, poiché questa non era presente in Weka.

4.4 Metriche

Andiamo adesso ad elencare le metriche che sono state calcolate attraverso Weka ed il loro significato:

- %TruePositive (TP): Percentuale di istanze calcolate come positive e che realmente lo erano.
- %TrueNegative(TN): Percentuale di istanze calcolate come positive ma che in realtà dovevano essere negative.
- %FalsePositive(FP): Percentuale di istanze calcolate come negative e che realmente lo erano.
- %FalseNegative(FN): Percentuale di istanze calcolate come negative ma che in realtà dovevano essere positive.
- Precision[2]: Identifica il numero di volte che è stata correttamente classificata una istanza come positiva. Per calcolare questa metrica sfruttiamo la seguente formula:

$$\frac{TP}{TP + FP} \tag{4.1}$$

- Recall[3]: Identifica il numero di positivi che sono stati classificati in maniera corretta: Per calcolare questa metricha sfruttiamo la seguente formula:

$$\frac{TP}{TP + FN} \tag{4.2}$$

[1] In questo caso, la classe maggioritaria è rappresentata dalla percentuale di classi con attributo "Buggy" di valore "NO"

[2] Per maggiori informazioni: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall

[3] Per maggiori informazioni è possibile guardare lo stesso link di *Precision*

- AUC[4]: Rappresenta la capacità del modello di saper distinguere due classi. Nel nostro caso le classi sono rappresentate da "Vero/Falso" per la variabile Buggy, quindi maggiore risulta essere AUC e maggiore sarà la capacità del modello di predirre un Falso come tale o un vero come tale (ottenendo quindi un rate alto di TruePositive e True Nagative). Per la determinazione della metrica usiamo quindi TPR e TPF, che sono individuate da:

$$TPR = \frac{TP}{TP + FN}; FPR = \frac{FP}{FP + TN} \tag{4.3}$$

- Kappa: Numero di volte che il classificatore è stato più accurato rispetto ad un classificatore *Dummy*. Per calcolare questa metrica sfruttiamo la seguente formuila:

$$\frac{Accuracy - AccuracyDummy}{1 - AccuracyDummy} \tag{4.4}$$

  Il risultato della metrica Kappa è quindi compreso tra -1 ed 1, in generale più il risultato si avvicina ad 1 e più il nostro classificatore sarà migliore rispetto ad un classificatore *Dummy*.

4.5 Feature Selection

L'importanza della feature selection sta nella riduzione dei *Costi di Apprendimento*, andando quindi a ridurre il numero di attributi all'interno del dataset e contemporaneamente fornendo una migliore performance per l'apprendimento, andando a compararla con la performance ottenuta dal set originale di attributi. All'interno del progetto è stata utilizzata una tecnica di Feature Selection che sfrutta l'approccio *Wrapper*, cioè andando a selezionare un sottoinsieme di attributi utilizzando il classificatore come una *Black-Box*. La tecnica utilizzata nella deliverable prende il nome *Best First* e sfrutta l'approccio Backward per essere computata. *Backward Search* risulta essere una euristica Greedy che parte dalla totalità degli attributi e continua ad eliminare altri se si hanno peggioramenti nell'*Accuracy* non significativi. Il costo computazionale di questa euristica è individuato da:

$$O(n^2 * \frac{LearingTime}{TestingTime}) \tag{4.5}$$

4.6 Progettazione

In questa sezione andiamo a presentare i Package e le classi presenti all'intendo del progetto, andando a spiegare il contesto di utilizzo delle variabili

[4]Per maggiori informazioni: https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc

ed il funzionamento dei metodi che hanno permesso di arrivare al risultato finale.

4.6.1 Classi

Il modulo del progetto presenta un unico Package chiamato *engine* all'interno del quale sono presenti le seguenti classi:

- CSV2Arff: Questa classe sfrutta le API esposte da Weka per trasformare un file in formato CSV in un file in formato ARFF. L'utilizzo del file ARFF è necessario per la determinazione degli attributi del set, infatti se una divisione della release non presentasse delle classi "Buggy", allora l'attributo che vogliamo predirre con Weka sarebbe unitario, e quindi non avremmo potuto utilizzare i classificatori *Random Forest*, *Naive Bayes* e *IBK*.
- DivideCSVByRelease: Questa classe ha il compito di andare a dividere l'output della precedente milestone tenendo conto della release, in modo tale da poter sfruttare come tecnica di validazione *Walk Forward*.
- WekaEngine: Questa classe rappresenta il cuore della milestone, dove i metodi sfruttano i classificatori per andare a deteminare le metriche finali di nostro interesse. In particolare possiamo notare due metodi fondamentali:
  - private Object[] calculateNoFilter(...)
  - private Evaluation calculateWithFilter(...)
  - public List<String[]> walkForwardValidation(int numOfSteps)

–private Object[] calculateNoFilter(...)–

Questo metodo prende in input un classificatore, un istanza di training ed una istanza di testing ed un intero *mode* che identifica la tecnica di Sampling da dover utilizzare.

–private Evaluation calculateWithFilter(...)–

Questo metodo prende in input un classificatore, un istanza di training , una istanza di testing ed un Valutatore (*Evaluation*) e va ad eseguire nuovamente le misurazioni applicando come metodo di Feature Selection il *Best First*.

–public List<String[]> walkForwardValidation(int numOfSteps)–

Questo è infine il metodo che esegue la validazione di tipo *Walk Forward*. L'intero numOfSteps, determina il numero di iterazioni che dovranno essere eseguite. Questo metodo aggiunge i risultati ottenuti all'interno di una lista per ogni iterazione e alla fine restituisce un file CSV contenente tutte le misurazioni effettuate.

4.7 Grafici e Commenti

Passiamo ora all'analisi dei risultati ottenuti dalle misurazioni, andando prima ad introdurre i grafici ricavati dall'analisi dei due progetti (Tajo e Bookkeeper) e successivamente andando a rispondere alla domanda posta nella specifica del progetto.

4.7.1 Bookkeeper

Vogliamo rispondere alla domanda di quali tecniche di *Fature Selection* e *Balancing* aumentano l'accuratezza dei classificatori. Per rispondere andremo ad analizzare il grafico (Fig. 4.1). Focalizzandoci inizialmente sulle tecniche di Feature Selection, in generale, possiamo notare come con l'applicazione della tecnica di Feature Selection *Best First* si abbia una maggiore variabilità dei valori di *AUC* e di *Recall*, rispetto ai risultati ottenuti senza l'ausilio della stessa. Sotto l'ipotesi di *Best First* notiamo anche una diminuzione della variabilità del parametro *Kappa*.

Per quanto riguarda la variabile *Precision*, invece, dobbiamo eseguire una differenziazione sui classificatori utilizzati. Possiamo infatti vedere che per i calssificatori *IBK* e *RandomForest*, l'utilizzo della tecnica di Feature Selection Best First sembra farne diminuire la variabilità; considerando invece il Classificatore *NaiveBayes* sembra che il valore di precision rimanga invariato.

Andando invece a focalizzarci maggiormente sulle tecniche di *Balancing*, notiamo come *SMOTE* sembra essere la più appropriata per tutti e tre i classificatori, in quanto, anche avendo un valore di *Kappa* inferirore rispetto ad altre tecniche di balancing, permette di avere, in generale, una variabilità inferiore sui risultati.

Andando infine a considerare il Classificatore *Naive Bayes*, possiamo notare dei risultati simili per tutte le combinazioni delle tecniche di Balancing e di Feature Selection, con qualche piccola differenza dovuta alla variabilità dei Precision, Recall, AUC e Kappa.

Possiamo dire che l'utilizzo delle tecniche di Feature Selection ha avuto un impatto sulla varibilità dei risultati, mentre considerando le tecniche di Balancing, ed in particolare SMOTE, notiamo come ci sia un aumento generale della precisione sui classificatori.

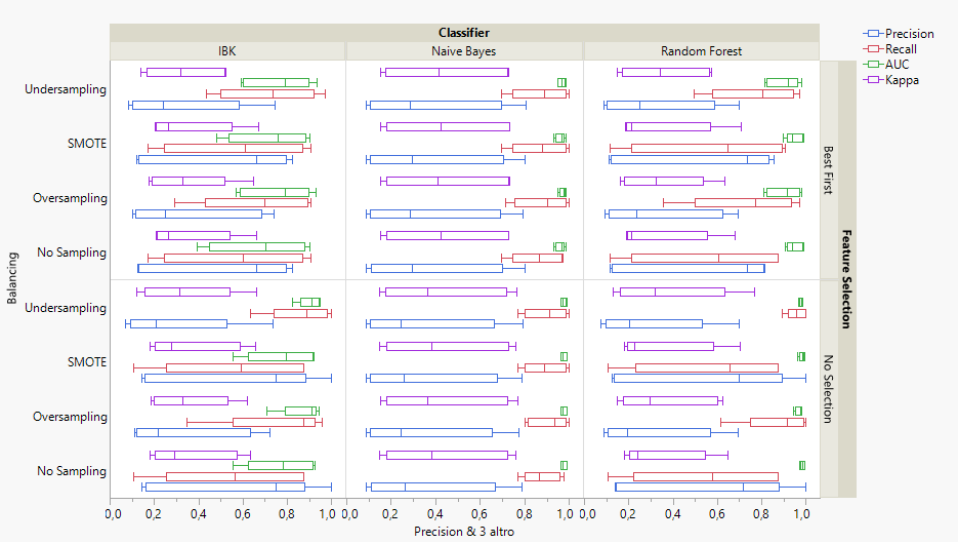

Figura 4.1: Bookkeeper: Value of Metrics over Balancing and Feature Selection for every Classificator

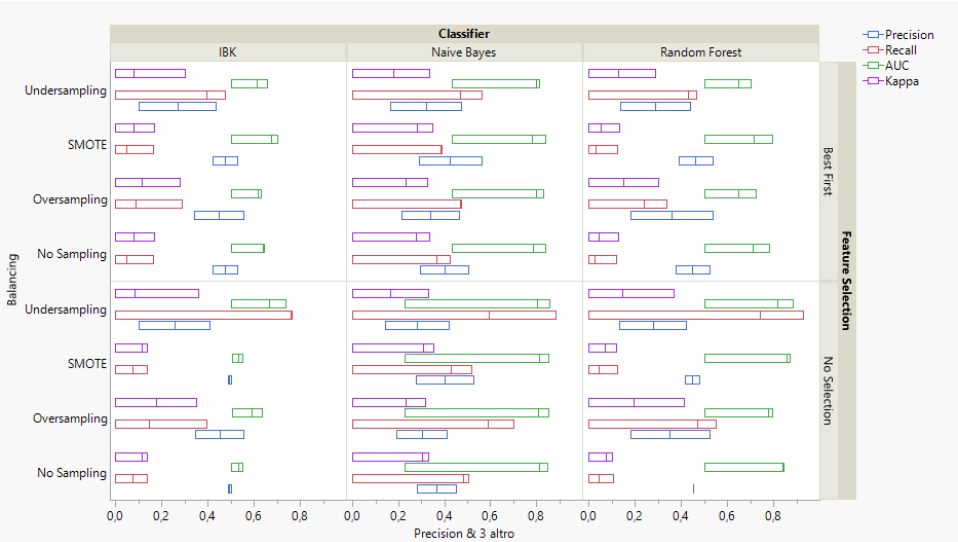

Figura 4.2: Tajo: Value of Metrics over Balancing and Feature Selection for every Classificator

Concludiamo, infine, affermando che, per il progetto open-source *Bookkeeper*, il classificatore *Naive Bayes*, tramite l'ausilio di *Best Firt* come tecnica di Feature Selection e sfruttando *SMOTE* come tecnica di Balancing, sembra essere la configurazione che permette di ottenere risultati migliori rispetto alle altre.

4.7.2 Tajo

Anche per quanto riguarda il progetto open-source *Tajo* siamo interessati a rispondere alla domanda posta all'interno della specifica del progetto. Andando ad analizzare il grafico (Fig 4.2), anche in questo caso possiamo inizialmente focalizzarci sulle tecniche di Feature Selection. In generale infatti notiamo come l'applicazione della tecnica di Feature Selection *Best First*, tenendo conto dei classificatori *Naive Bayes* e *Random Forest*, vada ad diminuire la variabilità dei valori di *AUC* e di *Recall*; Si ha invece un risultato "opposto" andando a considerare il classificatore *IBK*, dove possiamo notare che, attraverso le tecniche di balancing *SMOTE* e *No Sampling*, si ha una diminuzione della variabilità per quanto rigurda *AUC* e *Precision*.

Andando a focalizzare la nostra attenzione, invece, sulle tecniche di *Balancing*, notiamo come fra tutte la tecnica *SMOTE* sembra essere la quella più appropriata per tutti e tre i classificatori considerati, in quanto, seppur avendo una diminuzione del valore di *Kappa*, si riscontra una sostanziale diminuzione della variabilità dei risultati rispetto alle altre tecniche di balancing.

Anche per questo progetto open-source, come già riscontrato anche su *Bookkeeper*, andando infine a considerare il Classificatore *Naive Bayes*, possiamo notare dei risultati simili per tutte le combinazioni delle tecniche di Balancing e di Feature Selection.

L'utilizzo delle tecniche di Feature Selection ha avuto un impatto sulla varibilità dei risultati, mentre considerando le tecniche di Balancing, ed in particolare SMOTE, possiamo notare come ci sia un aumento generale della precisione sui classificatori presi in considerazione all'interno del progetto.

Concludiamo, infine, affermando che, per il progetto open-source *Tajo*, il classificatore *Naive Bayes*, tramite l'ausilio di *Best Firt* come tecnica di Feature Selection e sfruttando *SMOTE* come tecnica di Balancing, sembra essere la configurazione che permette di ottenere risultati migliori rispetto alle altre. Otteniamo un risultato molto simile a quanto visto per *Bookkeeper*.

Capitolo 5

Links

Riportiamo qui di seguito i link di GitHub, Travis-CI e SonarCloud del progetto:

5.1 Deliverable 1

- GitHub: https://github.com/GlaAndry/Milestone1-2
- Travis-CI: https://travis-ci.com/github/GlaAndry/Milestone1-2
- SonarCloud: https://sonarcloud.io/dashboard?id=GlaAndry_Milestone1-2

5.2 Deliverable 2

- GitHub: https://github.com/GlaAndry/Deliverable2
- Travis-CI: https://travis-ci.com/github/GlaAndry/Deliverable2
- SonarCloud: https://sonarcloud.io/dashboard?id=GlaAndry_Deliverable2