Università degli studi di Roma Tor Vergata
Facoltà di Ingegneria Informatica

# Machine learning for software engineering
# Report

Anno Accademico
2019/2020

Giuseppe Lasco 0286045

# Sommario

# Deliverable 1

## Introduzione

Questa deliverable si pone l'obiettivo di produrre un *process control chart* al fine di valutare l'andamento e la stabilità di un attributo di progetto. Nel caso in esame si è analizzato il progetto open source *Apache Libcloud.* Attraverso il merging di informazioni provenienti da *Jira*, una piattaforma per il monitoraggio di ticket e bug, e GitHub, un servizio di hosting per progetti software, che implementa lo strumento di version control *Git,* è stata possibile la costruzione di un dataset relativo all'attributo *#fixed tickets,* che ha portato alla costruzione del *process control chart.* Il grafico derivante permette di tenere sotto controllo l'andamento di un progetto, di individuare oscillazioni e indagare sulle cause, al fine di raggiungere un livello di maturità maggiore.

## Progettazione

È stato sviluppato un programma *Java* che si snoda in più fasi:

- Recupero dei ticket con status *resolved* o *closed* con resolution *fixed* dalla piattaforma *Jira,* facendo uso delle API REST messe a disposizione dal servizio.
- Recupero dell'ultimo commit che ha come commento l'ID del ticket da *GitHub*, facendo uso della Command Line Interface messa a disposizione da Git.
- Recupero delle date di creazione e chiusura del progetto (in caso non fosse ancora attivo), coincidenti con il primo e l'ultimo commit effettuato.
- Estrazione della data (con cadenza mensile) di esecuzione del commit e aumento di un contatore relativo ad essa.
- Costruzione del dataset in formato csv

I dati sono stati gestiti con una *HashMap* la cui chiave corrisponde alla data e il valore al contatore di *fixed tickets.* Di fondamentale importanza è stata l'introduzione anche delle date in cui non sono stati rinvenuti dati, al fine di non alterare il significato statistico.

## Risultati e discussione

La generazione del *process control chart* è stata effettuata con *Excel.* Si è calcolata la media di ticket chiusi al mese (3,137097) e la deviazione standard derivate dalla serie e utilizzate per definire un *Lower Limit* e un *Upper Limit* come segue:

$$Lower\ Limit = \max\big((Mean - 3 * StdDev), 0\big) = 0$$
$$Upper\ Limit = \mathrm{Mean} + 3 * \mathrm{StdDev} = \ 16{,}79851482$$

Tipicamente il *Lower Limit* è definito semplicemente dalla formula $Lower\ Limit = \mathrm{Mean} - 3 * \mathrm{StdDev}$, ma nel caso analizzato non avrebbe avuto senso, poiché avrebbe riportato un valore negativo che non può essere assunto da una variabile positiva come *#fixed version.* Tra i ticket del progetto sono stati esclusi quelli che non riportano la data di risoluzione, una porzione considerevole; questo ha portato ha una diminuzione della precisione dell'analisi.

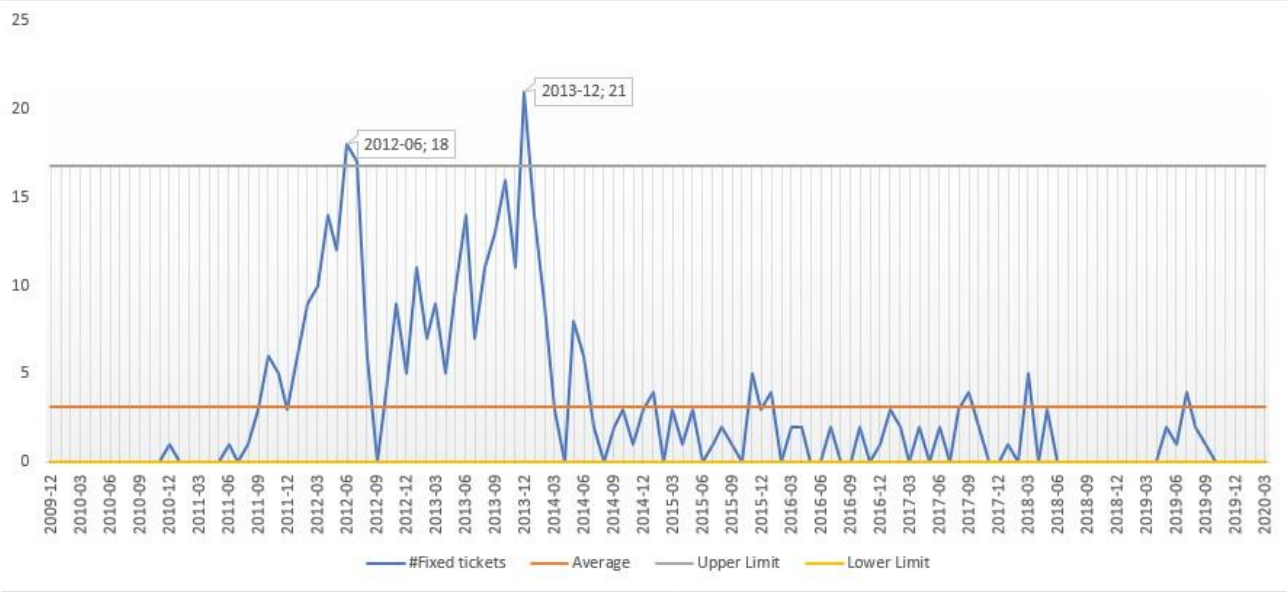


*Figura 1 Process Control Chart del progetto Libcloud relativo all'attributo Ticket Fixed*

La [Figura 1] mostra che nel complesso l'andamento dell'attributo risulta abbastanza stabile nel corso del tempo, tranne delle eccezioni in data 06/12 e 12/13 che introducono delle oscillazioni. Queste, però, si localizzano nella prima metà di vita del progetto, quindi del tutto naturali dato il basso livello di maturità iniziale che si innalza nel corso della seconda metà, in cui non si notano picchi, ma il processo sembra piuttosto stabile.
La corretta interpretazione del grafico porta un'organizzazione ad indagare sulle cause di instabilità e, quindi, alla risoluzione di problematiche, al miglioramento dell'attività di controllo e alla predizione, su base statistica, dei comportamenti futuri. Tutto ciò si riflette sui costi e sull'ottimizzazione al fine di raggiungere un alto livello di maturità.

### Link

- **https://github.com/giuseppelasco17/Deliverable1_Lasco**
- **https://sonarcloud.io/dashboard?id=giuseppelasco17_Deliverable1_Lasco**

## Deliverable 2

### Introduzione

Il fine ultimo di questa deliverable è quello di ottimizzare le attività di software testing, quindi di impiegare al meglio il budget. Tutto ciò si ottiene in termini statistici con l'utilizzo di tecniche di *Machine Learning.*
Il *Machine Learning* rappresenta uno strumento che permette, sulla base di conoscenze pregresse, con l'utilizzo di modelli, di ottenere previsioni sull'andamento di un certo processo. Alla base c'è il classificatore, ovvero un modello matematico che si addestra su un dataset di *training*, ovvero un insieme di osservazioni con delle caratteristiche di cui conosciamo gli esiti (*supervisionato*) o meno (*non supervisionato*), in seguito applicato ad un dataset di *testing* per verificarne la bontà in termini di alcune metriche (e.g. *precision*, *recall*, *f1 score*, …). Oltre a questo meccanismo di base vengono adottate tecniche che raffinano ulteriormente l'aderenza del modello al caso in esame, aumentando la bontà, ma facendo attenzione a non incorrere nell'*overfitting,* ovvero nell'adattamento eccessivo al training set, che porta ad un aumento degli errori di generalizzazione sul testing set.
Nel caso in esame, si sono raccolti dati relativi a classi *Java* di due progetti open source *Apache*, *Bookkeeper e Avro,* di cui si sono calcolate delle metriche e definita la *buggyness*, per poi essere utilizzati come base per l'addestramento di modelli di ML per poter definire, con una certa probabilità, la tendenza di una classe ad

essere buggata in una versione futura, ovvero sulla quale concentrare effort e budget nell'attività di software testing.

## Progettazione

L'obiettivo del caso di studio è stato raggiunto creando un programma *Java* che si compone di due parti: la costruzione del dataset e l'applicazione dei modelli di ML, al fine di avere una panoramica sulla bontà delle combinazioni di classificatori, tecniche di sampling (balancing), tecniche di feature selection, utilizzando Walk-forward come tecnica di holdout per la valutazione. È stato utilizzato Walk-forward per via delle dipendenze temporali esistenti tra le istanze presenti nel dataset

### Costruzione del dataset

La prima parte del programma si focalizza sulla creazione di un dataset che riporti una collezione di istanze composte da: versione, nome della classe, una serie di metriche e la difettosità della classe nella versione considerata. Tra le metriche calcolate si sono considerate le seguenti:

- **LOC_added**: # di linee di codice aggiunte;
- **MAX_LOC_added**: massimo di linee di codice aggiunte relative ad una revision appartenente ad una release;
- **AVG_LOC_added**: # medio di linee di codice aggiunte;
- **Churn**: # di linee di codice aggiunte meno quelle eliminate;
- **MAX_Churn**: massimo di linee di codice aggiunte meno quelle eliminate relative ad una revision appartenente ad una release;
- **AVG_Churn**: # medio di linee di codice aggiunte meno quelle eliminate;
- **NR**: # di revision di una release;
- **ChgSetSize**: # di classi di cui si è fatto un commit insieme a quella in questione;
- **MAX_ChgSet**: massimo valore raggiunto per la metrica *ChgSetSize* in relazione ad una certa revision di una versione;
- **AVG_ChgSet**: media di *ChgSetSize* sul numero di release.

L'analisi dei due progetti Apache è stata condotta su metà della durata di questi ultimi, poiché si è evinto, da studi statistici, che eliminando la seconda metà di un progetto dallo studio è possibile abbassare il numero di bug dormienti fino al 10%, diminuendo la distorsione introdotta da dati falsati nel dataset, nota come *snoring.*

Il programma si snoda nei seguenti passi:

- Individuazione della durata complessiva del progetto, catturando la data di primo e ultimo commit attraverso la Command Line Interface messa a disposizione da Git.
- Retrieve delle informazioni relative allo storico delle versioni e data di release usando le API REST dalla piattaforma *Jira.*
- Traduzione delle date in versioni e calcolo della release relativa alla metà della vita del progetto.
- Recupero dei ticket con status *resolved* o *closed* con resolution *fixed* da *Jira.*
- In relazione ad ogni ticket vengono individuati:
    - **IV**: *injection version*, ovvero la versione in cui il bug è stato introdotto. Ove tale informazione fosse risultata assente, è stata calcolata utilizzando il metodo *Proportion,* più avanti descritto.
    - **OV**: *opening version*, ovvero la versione relativa alla scoperta del bug e alla registrazione del ticket.
    - **FV**: *fixed version*, la versione in cui il difetto è stato risolto, ovvero il ticket è stato chiuso. Questo dato, ove assente in *Jira,* è stato estrapolato dalla versione riconducibile alla data di ultimo commit relativo a tale ticket.
- Individuazione della lista totale delle classi attraverso Git.

- Merging delle informazioni derivate dai ticket e quelle derivate da Git al fine di generare una lista di entry del dataset finale, dotate dell'attributo buggyness relativo ad ogni classe, per ogni versione, calcolato come positivo nell'intervallo [*IV, FV-1*] definito come AV, *affected versions*.
- Calcolo delle metriche facendo uso del comando *diff* di Git.
- Meging finale delle metriche calcolate e la lista generata al punto precedente.
- Scrittura del dataset in formato csv.

Sono state scartati tutti i ticket che hanno FV non valida o assente, ovvero relativa a una versione assente nel progetto, OV non valida, OV maggiore di FV oppure IV maggiore della versione relativa a metà della durata della vita del progetto.
Le informazioni relative alla data di introduzione di un certo difetto, il più delle volte, non sono disponibili, per cui è bene ricavare tali informazioni con un approccio statistico con l'introduzione di un metodo chiamato *Proportion.* Si è notato che esiste una proporzione tra il tempo che intercorre tra IV e OV e quello che intercorre tra OV e FV, ovvero esistono un certo numero di versioni che si susseguono tra l'introduzione di un difetto, la sua scoperta e la sua risoluzione che sono abbastanza stabili nel ciclo di vita di un progetto.
Il valore proporzionale P è stato calcolato utilizzando i ticket validi che possedevano i valori di Injected Version tramite la seguente formula:

$$\mathrm{P} = \frac{\mathrm{FV} - \mathrm{IV}}{\mathrm{FV} - \mathrm{OP}}$$

Nel caso di ticket validi, ma mancanti del dato relativo a IV è stata utilizzata la seguente formula per il calcolo:

$$\text{Predicted IV} = \mathrm{FV} - (\mathrm{FV} - \mathrm{OV}) * \mathrm{P}$$

Tra gli approcci per il calcolo di *P,* si è scelto quello incrementale, ovvero la proporzione viene calcolata man mano che l'analisi dei ticket va avanti, aggiornando di volta in volta *P* tramite una media incrementale.

## Applicazione dei modelli di Machine Learning

La seconda parte del programma si concentra sull'applicazione dei modelli di ML, sui dati precedentemente acquisiti, e sulla creazione di un dataset che riassume i risultati delle misurazioni in termini delle seguenti colonne:

- Nome del progetto
- Numero di release considerate nel training
- Percentuale di dati utilizzati nel training
- Percentuale di classi difettose nel dataset di training
- Percentuale di classi difettose nel dataset di testing
- Classificatore utilizzato
- Tecnica di balancing utilizzata
- Tecnica di feature selection utilizzata
- Numero di veri positivi (True Positive - TP)
- Numero di falsi positivi (False Positive - FP)
- Numero di veri negativi (True Negative - TN)
- Numero di falsi negativi (False Negative - FN)
- Metriche di accuratezza

Le metriche di accuratezza calcolate sono:

- Recall: $\frac{TP}{TP+FN}$ indica quanti positivi il modello è in grado di classificare.
- Precision: $\frac{TP}{TP+FP}$ indica quante volte il modello ha correttamente classificato una istanza come positiva.

- AUC: area under ROC curve, indica quanto il modello è in grado di distinguere le classi. La ROC curve permette di valutare uno stesso classificatore al variare delle soglie di decisione, mentre la comparazione delle AUC di vari classificatori, permette di trarre conclusioni in merito alla bontà globale di ciascuno di essi in confronto agli altri.
- Kappa: indica quanto il modello è migliore rispetto ad un classificatore dummy, con un valore che oscilla tra -1 e 1.

L'utilizzo dei modelli e delle varie tecniche è stato possibile grazie all'utilizzo di API *Java* messe a disposizione dalla libreria offerta dal software *Weka.*

## Risultati e discussione

Al fine di avere un'idea più chiara e ampia per poter fare considerazioni e valutazioni in merito a quale combinazione di tecniche e modelli fosse più aderente al dataset considerato, sono stati generati dei grafici riassuntivi che mettono in relazione le varie caratteristiche.
Si è scelto di generare un *boxplot* relativo a tutte le combinazioni di tecniche e classificatori, in modo da avere una panoramica ampia e riassuntiva. Inoltre, il *boxplot* permette di fare considerazioni non solo legate al valor medio ottenuto nei vari step del *walk-forward*, ma anche relativamente alla stabilità del valore ottenuto, osservando l'ampiezza del grafico e il posizionamento dei vari quantili principali della distribuzione.

### Bookkeeper

In [Figura 2] sono riportati i grafici relativi al progetto ***Bookkeeper***. Si può osservare che la combinazione migliore sembra essere quella che ha come classificatore *Ibk*, come tecnica di balancing *SMOTE* e nessuna tecnica di *feature selection*. Si nota che il *boxplot* relativo alla *recall* è molto schiacciato e ha una mediana molto spostata verso 1, cosa che è sicuramente positiva, ma questo indicatore risulta poco significativo, poiché la recall indica il numero di *TP* individuati, un valore piuttosto elevato può essere raggiunto da qualsiasi classificatore che classifica tutti gli elementi come positivi; valori elevati delle mediane relative *ROC area, kappa* e *precision*, invece, fanno pensare indubbiamente che questa combinazione sia preferibile rispetto alle altre, anche osservando il grafico molto raccolto e schiacciato verso l'alto della *precision*.
L'utilizzo di una tecnica di balancing migliora le prestazioni, il più delle volte, soprattutto in casi in cui c'è una grossa disparità tra le classi e il modello potrebbe avere poche istanze di una certa classe su cui addestrarsi, inoltre, *SMOTE* è una tecnica di *oversampling*, ovvero che incrementa il numero di istanze di classi minoritarie, che fa interpolazione tra quelle esistenti, ovvero introduce istanze completamente nuove con feature interpolate tra quelle esistenti. L'uso di una tecnica di *feature selection* è in grado di eliminare dal dataset, di solito utilizzando come discriminante un indice statistico, delle caratteristiche meno significative che possono in qualche modo impattare negativamente sull'addestramento del modello; tale operazione permette, inoltre, di diminuire la dimensione del dataset, quindi permettendo un minor dispendio di potenza di calcolo nell'addestramento del modello e permettendo una maggiore generalizzazione, contrastando problemi di *overfitting*. Nel caso in esame i risultati migliori sono stati raggiunti senza *feature selection*, questo fa pensare che tutte le caratteristiche fossero fondamentali per la classificazione. La combinazione migliore individuata non sembra un caso particolare, poiché in generale, con ognuno dei tre classificatori, la tecnica di *balancing SMOTE* e il mancato impiego di *feature selection* sembrano migliorare le prestazioni del modello. In relazione alla miglior combinazione individuata si è analizzata la composizione dei risultati rispetto a *TP, FP, TN* e *FN* [Figura 3], da cui si può notare come in media il classificatore commette molti più errori nella classificazione di negativi che di positivi, questo per la composizione sbilanciata del testing set. In [Figura 4] è riportato un grafico relativo alla percentuale media delle classi difettose nel training set rispetto agli step del *walk-forward.* Si nota come sia molto più presente la classe 'non defective', per cui è giustificato un incremento della bontà del modello impiegando tecniche di balancing dei dati.


*Figura 2 Boxplot risultati metriche Bookkeeper*

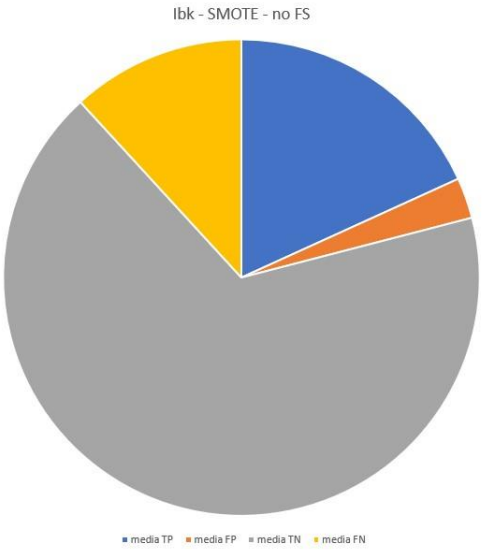


*Figura 3 Numero di veri positivi, falsi positivi, veri negativi e falsi negativi medi per la combinazione migliore*

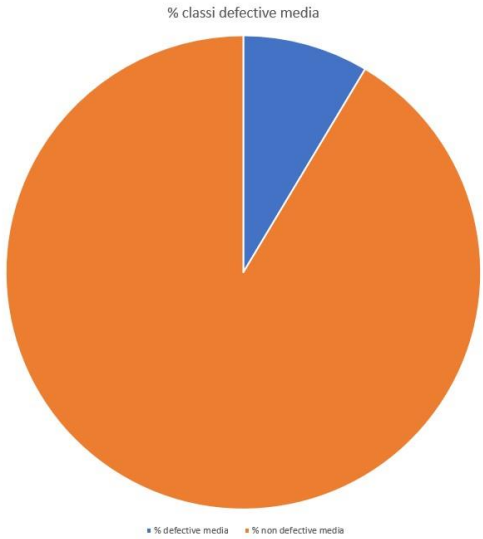


*Figura 4 Percentuale classi defective in media nel training set*

## AVRO

In [Figura 5] sono riportati i grafici relativi al progetto ***Avro***. Come nel caso precedente, anche in questo caso i risultati migliori sembrano essere quelli in cui si è scelto *SMOTE* come tecnica di *balancing* dei dati, infatti le metriche sono elevate, tranne nel caso di *kappa* in relazione al classificatore *Naive Bayes*. In questo caso la *feature selection* non impatta molto sui risultati, tranne che per la *recall* con il classificatore *Naive Bayes*, che risulta migliore con l'uso di *best first*. Per concludere, la soluzione migliore risulta essere *Random Forest* come classificatore, *SMOTE* come tecnica di *balancing* e *best first* come tecnica di *feature selection*. Come si nota dal grafico relativo alla composizione media del training set in [Figura 7], il numero di classi defective nel training set è molto bassa, per cui il modello potrebbe avere problemi nell'addestrarsi nell'individuazione della classe minoritaria; questo risulta coerente con l'osservazione precedente in cui i risultati migliori sono stati ottenuti utilizzando *SMOTE* come tecnica di balancing, ovvero una tecnica di oversampling delle classi minoritarie. La proporzione dei positivi e negativi classificati dal modello [Figura 6] risulta coerente con la composizione media del training set, condizionato dalla forte presenza di casi 'non defective' e influenzato dalla composizione del testing set, in linea con quella del training set poiché ricavati dallo stesso dataset.

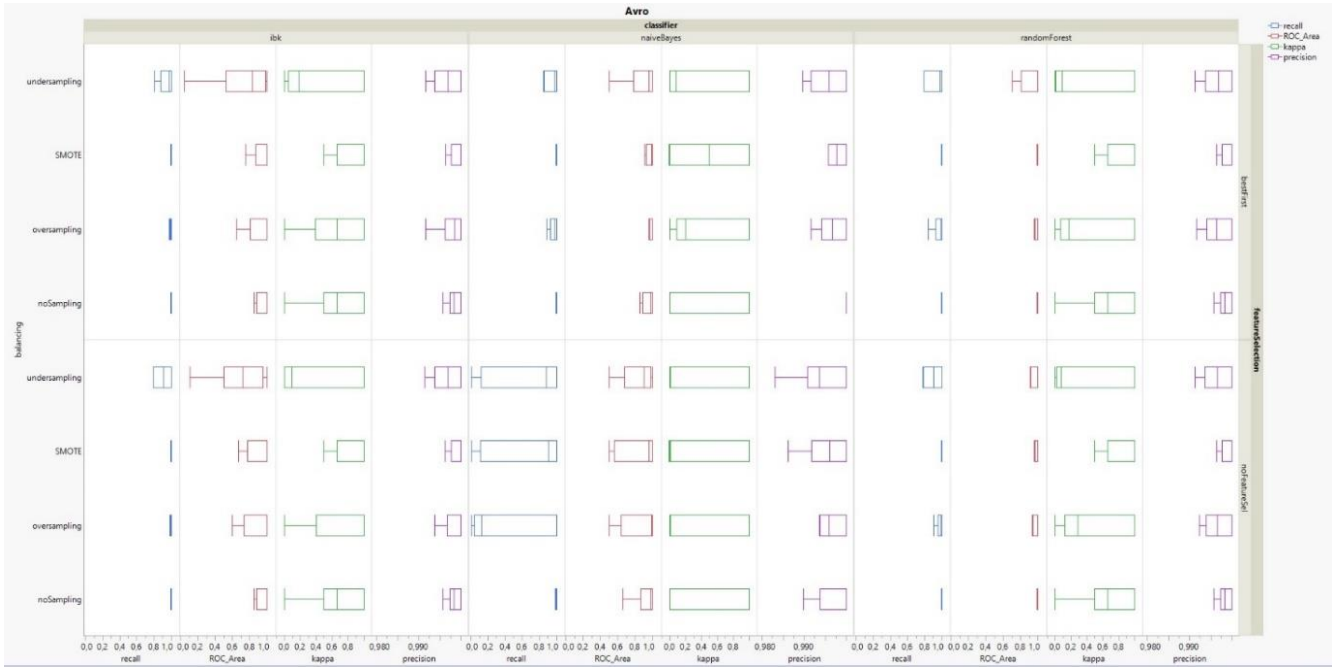


*Figura 5 Boxplot risultati metriche AVRO*

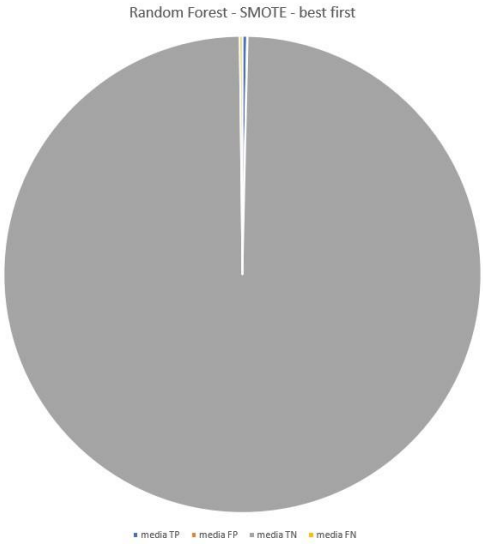


*Figura 6 Numero di veri positivi, falsi positivi, veri negativi e falsi negativi medi per la combinazione migliore*

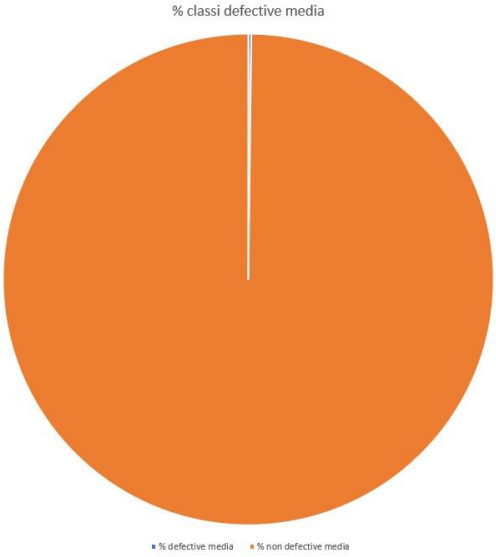


*Figura 7 Percentuale classi defective in media nel training set*

## Link

- **https://github.com/giuseppelasco17/Deliverable2_Lasco**
- **https://sonarcloud.io/dashboard?id=giuseppelasco17_Deliverable2_Lasco**