Deliverable 2

Gian Marco Falcone - 0300251

Indice

- Introduzione: Scopo e Strumenti utilizzati
- Dataset: Jira e Git
- Dataset: Proportion
- Dataset: Ultime considerazioni
- Weka: Contesto e Osservazioni
- Weka: Risultati e Analisi
- Weka: Feature Selection
- Weka: Sampling
- Weka: Cost sensitive classifier
- Conclusioni
- Links

Introduzione: Scopo

Si vuole eseguire uno studio empirico finalizzato a misurare l'effetto di tecniche di sampling, classificazioni sensibili al costo e feature selection, sull'accuratezza di modelli predittivi di localizzazione di bug nel codice.

Scopo del deliverable è quello di valutare come l'applicazione di diverse tecniche di feature selection, balancing o sensitivity aumenti o meno l'accuratezza dei classificatori in esame nel predire la difettosità di una certa classe in una release futura.

A tal fine le sperimentazioni hanno riguardato due progetti open-source di Apache:

- BookKeeper
- Avro

Introduzione: Strumenti utilizzati

Per lo sviluppo dell'applicativo si è fatto uso di:

- Linguaggio Java e Eclipse, per lo sviluppo del codice;
- Git e Jira, per il recupero delle informazioni necessarie nella costruzione del dataset;
- Weka, per l'utilizzo degli algoritmi di Machine Learning;
- SonarCloud, per l'analisi di qualità sul codice prodotto.

Dataset: Jira e Releases

Il primo passo è quello di costruire il dataset.

Si utilizzano le Jira REST API per ottenere le informazioni sulle release del progetto sotto sperimentazione, come mostrato in figura.

Si usano queste informazioni per creare una lista ordinata di classi contenente ognuna l'Id, il nome e la data di rilascio di una certa release.


```
"self": "https://issues.apache.org/jira/rest/api/2/version/12313858",
"id": "12313858",
"description": "The first stable version of the specification.",
"name": "1.0.0",
"archived": false,
"released": true,
"releaseDate": "2009-07-14",
"userReleaseDate": "14/Jul/09",
"projectId": 12310911
```

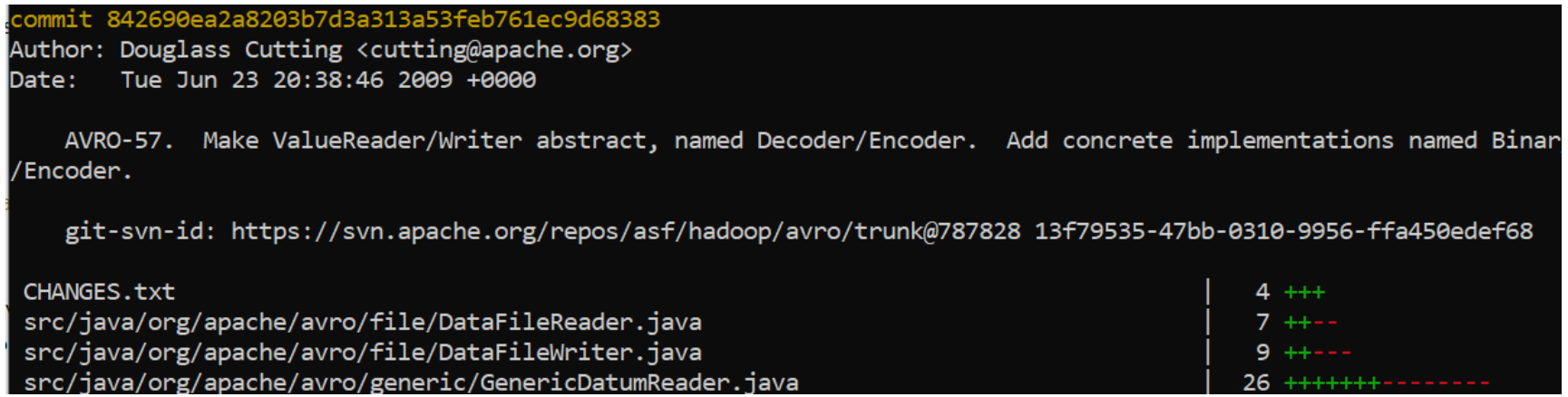
Dataset: Git

Si usa Git per creare una copia locale (*clone*) della repository remota di Apache.

Con l'uso del comando *log* di Git si mostrano tutti i commit in ordine cronologico.

Analizzando il risultato è possibile ottenere, per ogni release, le informazioni necessarie su ogni classe, tra cui:

- LOC touched
- LOC added
- Size
- Authors



Dataset: Gestione del Refactor del Codice

Si è notato che alcune classi hanno cambiato nome o path durante lo sviluppo dei progetti software, come è naturale aspettarsi. Questi cambiamenti però, presentando una struttura ben riconoscibile all'interno del log, sono stati gestiti in maniera corretta. Un esempio in cui ciò avviene è quello mostrato in figura.

Da notare che il cambiamento è racchiuso tra parentesi graffe e la presenza del simbolo '=>' tra il nome vecchio e quello nuovo della classe.

```
src/java/org/apache/avro/io/{BlockingValueWriter.java => BlockingBinaryEncoder.java}
src/java/org/apache/avro/io/DatumReader.java
src/java/org/apache/avro/io/DatumWriter.java
src/java/org/apache/avro/io/{ValueReader.java => Decoder.java}
```

Dataset: Jira e i Ticket

Sempre grazie all'utilizzo di Jira si ottengono i bug fixed per il progetto sotto osservazione, ossia quei ticket con valori:

- *project* = "nome_del_progetto"
- *issueType* = Bug
- *status* = closed OR resolved
- *resolution* = fixed

Per ogni ticket ottenuto si ricava la *chiave*, la data di apertura del ticket (*created*) e, se presenti, *fixVersions e versions*.

Si assume come *Injected Version* la prima in ordine cronologico tra le *versions e come Fixed Version l'ultima tra le fixVersions*, se ne sono presenti più di una.

Le restanti release in *versions* sono considerate *Affected Versions.*


```
"id": "13374670",
"self": "https://issues.apache.org/jira/rest/api/2/issue/13374670",
"key": "AVRO-3118",
"fields": {
  "fixVersions": [
    {
      "self": "https://issues.apache.org/jira/rest/api/2/version/12348321",
      "id": "12348321",
      "name": "1.11.0",
      "archived": false,
      "released": false
    }
  ],
  "versions": [
    {
      "self": "https://issues.apache.org/jira/rest/api/2/version/12349540",
      "id": "12349540",
      "description": "bugfix release",
      "name": "1.10.2",
      "archived": false,
      "released": true,
      "releaseDate": "2021-03-15"
    }
  ],
  "resolutiondate": "2021-05-28T21:42:36.000+0000",
  "created": "2021-04-23T13:49:26.000+0000"
}
```

Dataset: Integrazione tra Jira e Git

Effettuando un confronto incrociato con le informazioni sulle release ottenute in precedenza, si può valutare la *Opening Version* relativa ad ogni ticket.

Infine si usa Git per esaminare le classi modificate in quei commit che hanno, nel commento, l'Id relativo ad uno dei ticket bug fixed ottenuti in precedenza.

Per le classi così ottenute si valuta se queste debbano essere considerate difettose o meno, e, in caso affermativo, in quale release.

In particolare, sono considerate difettose in una determinate release se:

- sono relative ad un ticket il cui campo *affectedVersions* non è vuoto;
- una delle versioni in *affectedVersions* corrisponde alla release sotto osservazione;
- tale versione è diversa dalla *Fixed Version.*

Dataset: Considerazioni

In base a tale analisi è risultato che:

➢ Avro:

- 648 ticket su 827 presentano un campo *versions* valido, pari al 78%;
- Considerando tutte le release, 613 classi risultano essere difettose sulle 13313 totali.

➢ Bookkepper:

- 197 ticket su 435 presentano un campo *versions* valido, pari al 45%;
- Considerando tutte le release, 428 classi risultano essere difettose sulle 12112 totali.

I dati così ottenuti necessitano ancora di alcuni miglioramenti.

Dataset: Proportion

Per perfezionare il dataset, in quei casi in cui le classi difettose non possano essere correttamente calcolate per una mancanza delle informazioni necessarie, si può ricorrere alla tecnica di proportion.

Affinchè essa sia utilizzabile devono essere presenti *openingVersion* e *FixedVersion.*

*S*i procede quindi con lo stimare la presenza di ulteriori classi difettose in base alla formula:

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

Per entrambi i progetti si è proceduto con il calcolo di *P* in maniera incrementale, in base ai difetti valutati nelle versioni precedenti.

Ciò ha permesso di aumentare il numero di classi difettose totali a:

- 1326 per Avro, pari al 10% circa.
- 2556 per Bookkepper, pari al 21%.

Dataset: Ultime Considerazioni

Il lavoro fin qui eseguito ha permesso di creare, per ogni progetto sotto osservazione, un dataset utilizzabile da *weka*.

I dataset presentano la seguente struttura, da cui è possibile osservare le metriche utilizzate:

Version name	File name	Size	NR	NAuth	LOCTouch	LOCAdded	MAX_LOC_Added	AVG_LOC_Added	Churn	MAX_Churn	AVG_Churn	bugginess

C'è però ancora bisogno di un'ultima modifica. Poiché si è visto che molti difetti vengono scoperti solo molto tempo dopo la loro introduzione, l'andamento delle classi difettose sarà sicuramente descrescente nel tempo e il dataset biased, come è possibile notare dalle precedenti slides.

Per ovviare a questo problema e ridurre il numero di classi *snoring*, ossia quelle classi che presentano al loro interno dei difetti ma che ancora non sono stati rivelati, si procede scartando la seconda meta (50%) delle release nel dataset.

Si può ora procedere ad usare le Weka API sui dataset così costruiti.

Weka: Contesto

L'analisi tramite *weka* consiste nel valutare quale configurazione permette di aumentare l'accuratezza dei classificatori utilizzati, usando come tecnica di valutazione *walk forward.*

I classificatori da utilizzare sono:

- RandomForest
- NaiveBayes
- Ibk

Per ciascun classificatore si applicano come le seguenti configurazioni:

- Feature selection: No selection / Best First;
- Tecnica di sampling: No sampling / Oversampling / Undersampling / SMOTE;
- Cost sensitive classifier: No cost sensitive / Sensitive Threshold / Sensitive Learning.

Sono in totale richieste 72 run della tecnica *walk forward.*

Weka: Prime Osservazioni

Le singole iterazioni necessarie all'applicazione della tecnica *walk forward* sono salvate nei file csv contenenti i risultati ottenuti per ogni progetto.

Da notare che:

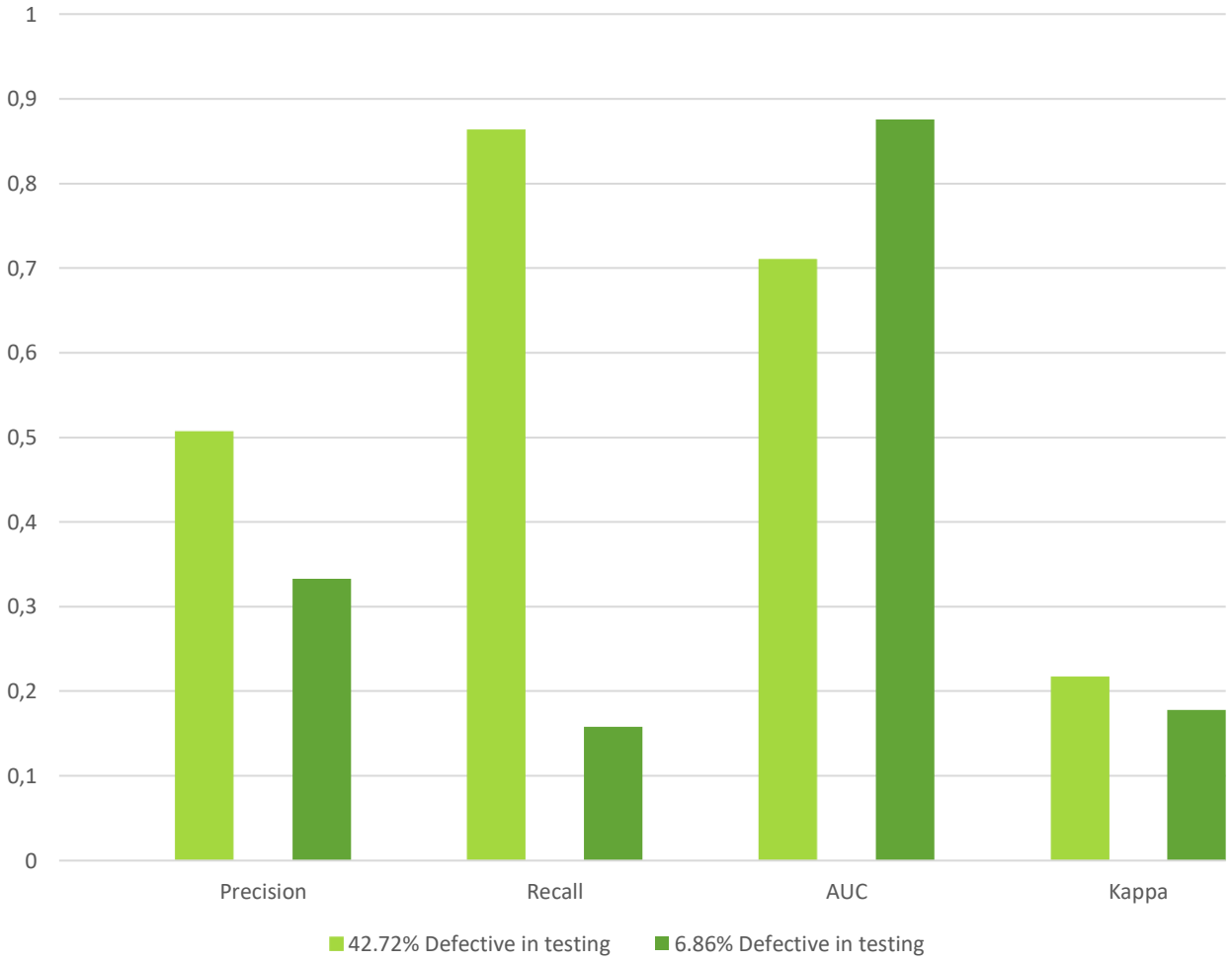
- i valori ottenuti per le metriche *precision* e *recall* dipendono fortemente dalla classe sotto analisi;
- i valori di *AUC* e *Kappa* sono indipendenti dalla scelta.

Ciò comporta che se le istanze da valutare sono quelle appartenenti alla classe prioritaria, il dataset risulta essere facile da stimare e presenta valori per le metriche molto alti.

Al contrario un dataset con poche classi positive è un dataset difficile da valutare.

Dalle successive analisi si potrà osservare come Bookkeepper, presentando una percentuale maggiore di classi positive, presenterà delle metriche generalmente migliori di Avro.

Weka: Prime Osservazioni



L'immagine mette a confronto i risultati ottenuti nel processo di *walk forward* per Avro, con la stessa scelta nelle tecniche applicate ma per due release di testing diverse e quindi diversa percentuale di classi positive in esse.

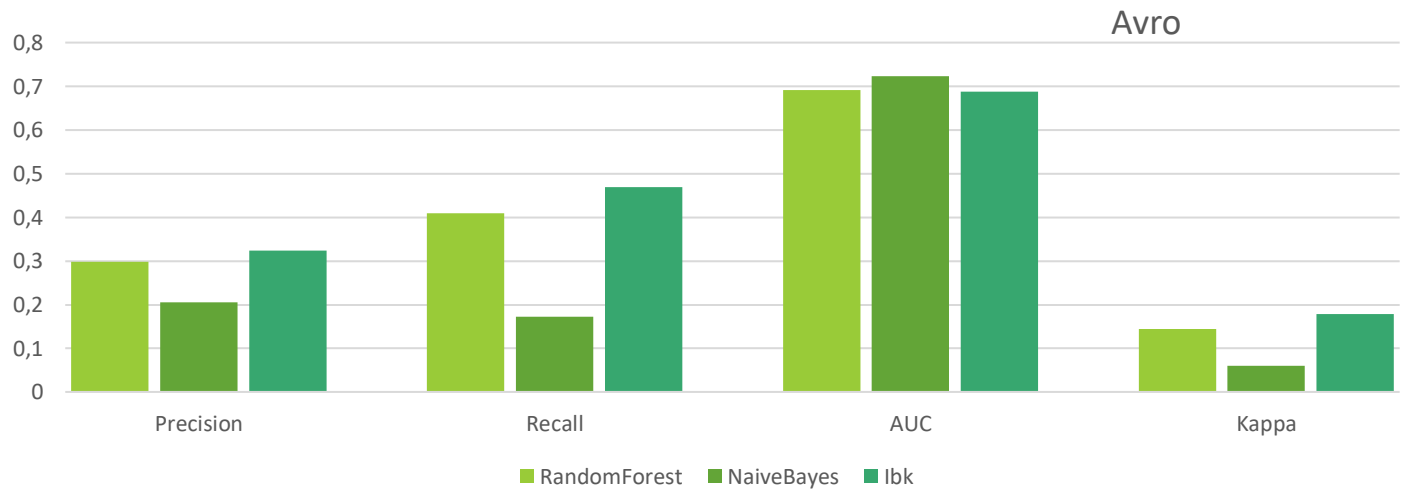
La release con più positivi presenta valori di *recall* e *precision* nettamente superiori, eppure, secondo la metrica *AUC*, non è lei quella ad essere stata stimata in maniera migliore.

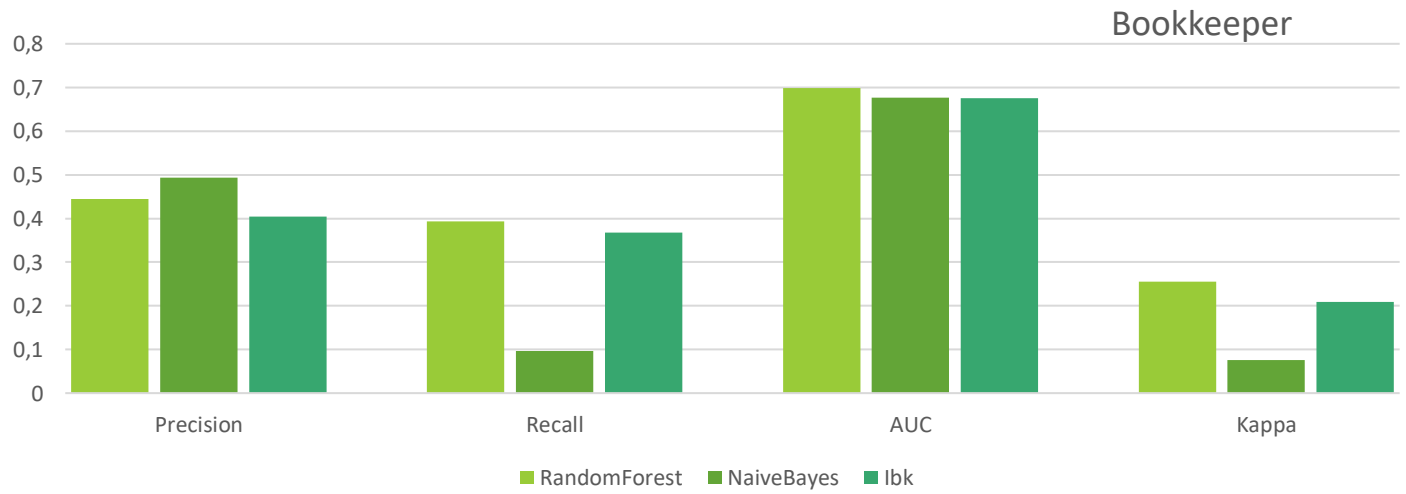
Weka: Risultati

Di seguito si analizza il comportamento dei classificatori selezionati.

Nei grafici si mostrano le medie delle metriche di accuratezza *recall, precision, AUC e Kappa*, riportate da ogni classificatore, supponendo:

- No Feature Selection
- No sampling
- No cost sensitive





Weka: Analisi dei Risultati

Dai risultati ottenuti nessun classificatore sembra essere, in assoluto, migliore degli altri, anche se NaiveBayes presenta dei valori veramente bassi di *precision* e *Kappa* per entrambi i progetti considerati.

Per tutti i classificatori e su entrambi i progetti la metrica *AUC* risulta sempre abbondantemente maggiore di 0.5 e la *Kappa* sempre positiva.

Questo dimostra che:

- i classificatori considerati riescano ad analizzare i dati meglio di un classificatore randomico;
- le metriche calcolate siano effettivamente in qualche modo correlate con la difettosità delle classi.

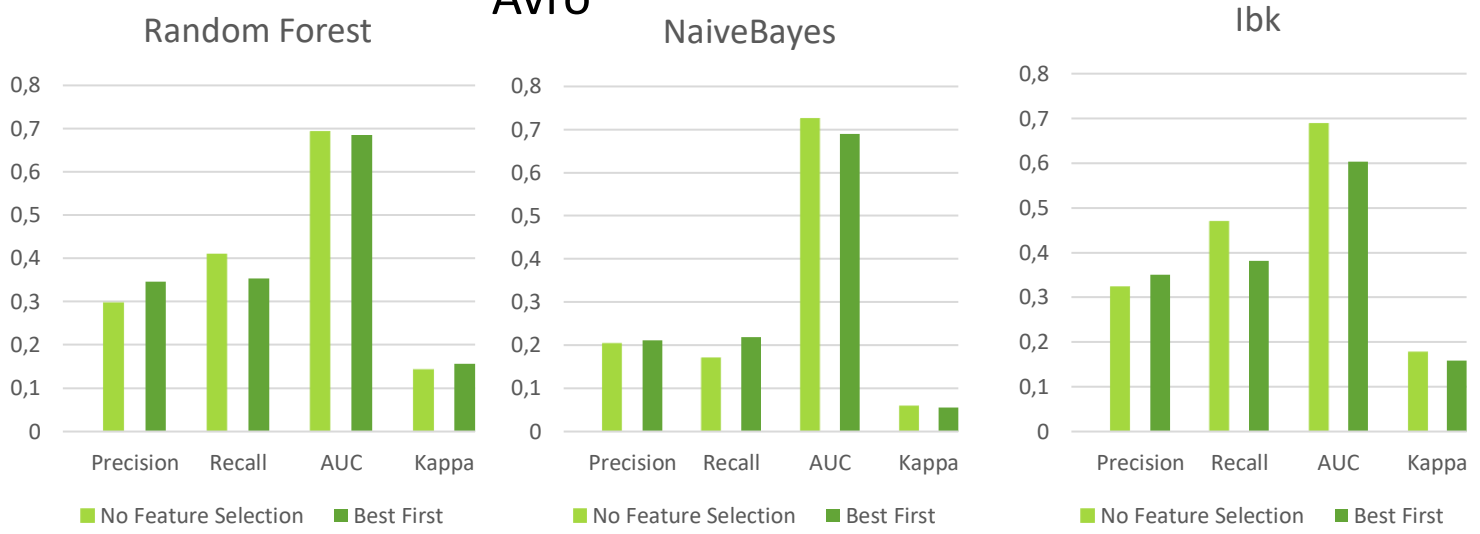
Di seguito proviamo a variare la combinazione delle tecniche applicate sul dataset per valutare come cambino le metriche di accuratezza per ogni classificatore.

Weka: Feature Selection

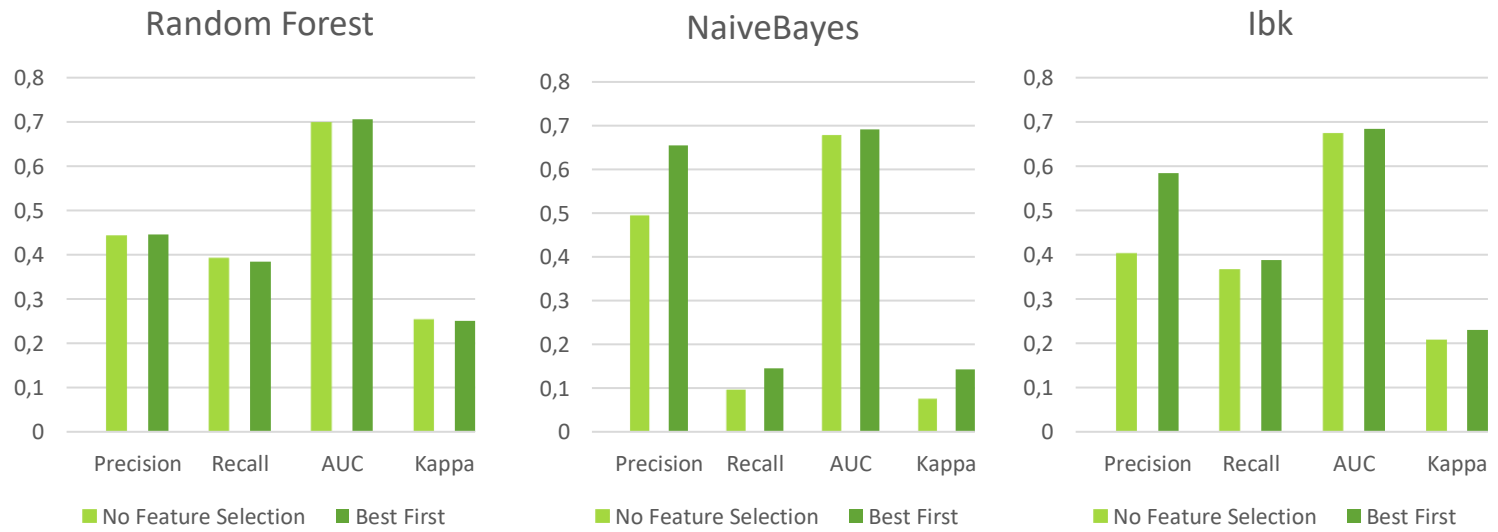
Si analizza il comportamento dei classificatori selezionati al variare della tecnica di feature selection:

- No Feature Selection
- Best First

Avro



Bookkeeper



Weka: Analisi Dell'Applicazione di Best First

Gli attributi che vengono mantenuti applicando Best First sono:

- *NR, LOCAdded* e *Churn* per Avro
- *Size, NR* e *AVG_Churn* per Bookkeeper

In generale l'applicazione della tecnica di feature selection Best First ha portato dei miglioramenti per la *precision*, notevole nel caso di Bookkeeper, e dei peggioramenti per la *recall* nella predizione delle classi difettose da parte dei classificatori.

I valori di *AUC* e *Kappa* risultano rimanere abbasanza stabili, eccetto per Ibk che vede peggiorare di molto i valori delle due metriche sul progetto Avro.

Avro in generale ha visto un peggioramento delle predizioni dopo l'applicazione di Best First, al contrario di Bookkeeper che invece ne ha beneficiato, anche se leggermente.

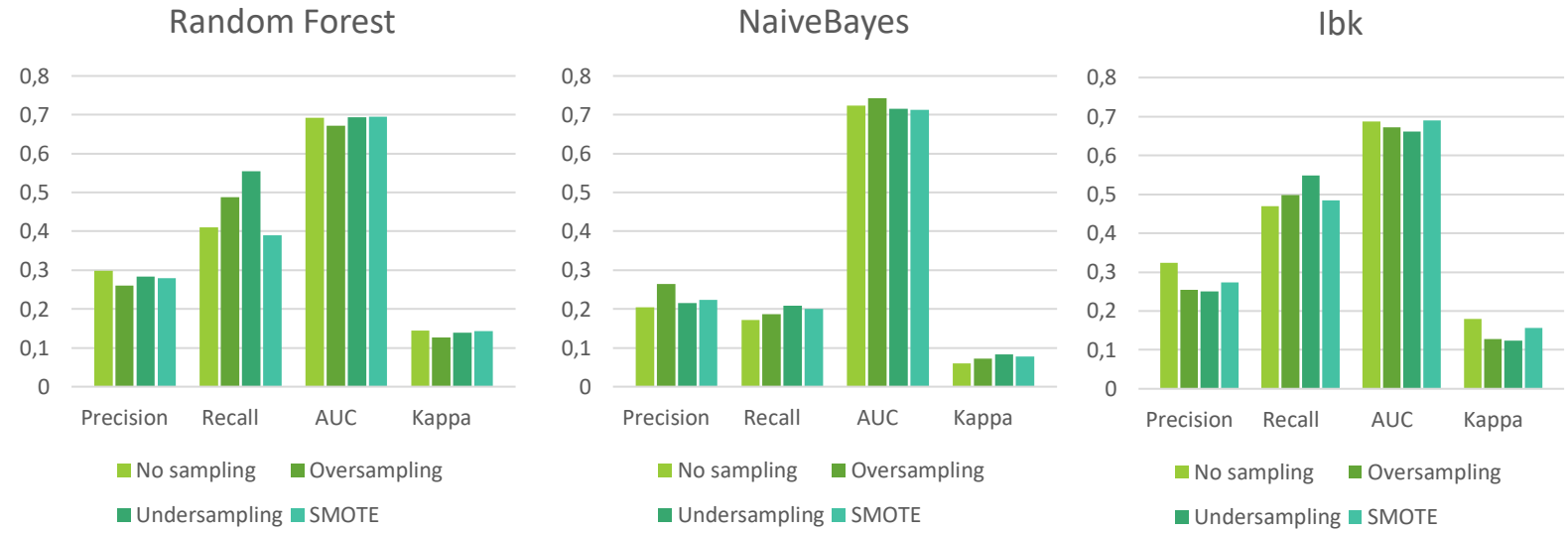
L'unico classificatore che sembra aver migliorato le sue analisi, su entrambi i progetti e per quasi tutte le metriche, è NaiveBayes.

Weka: Sampling

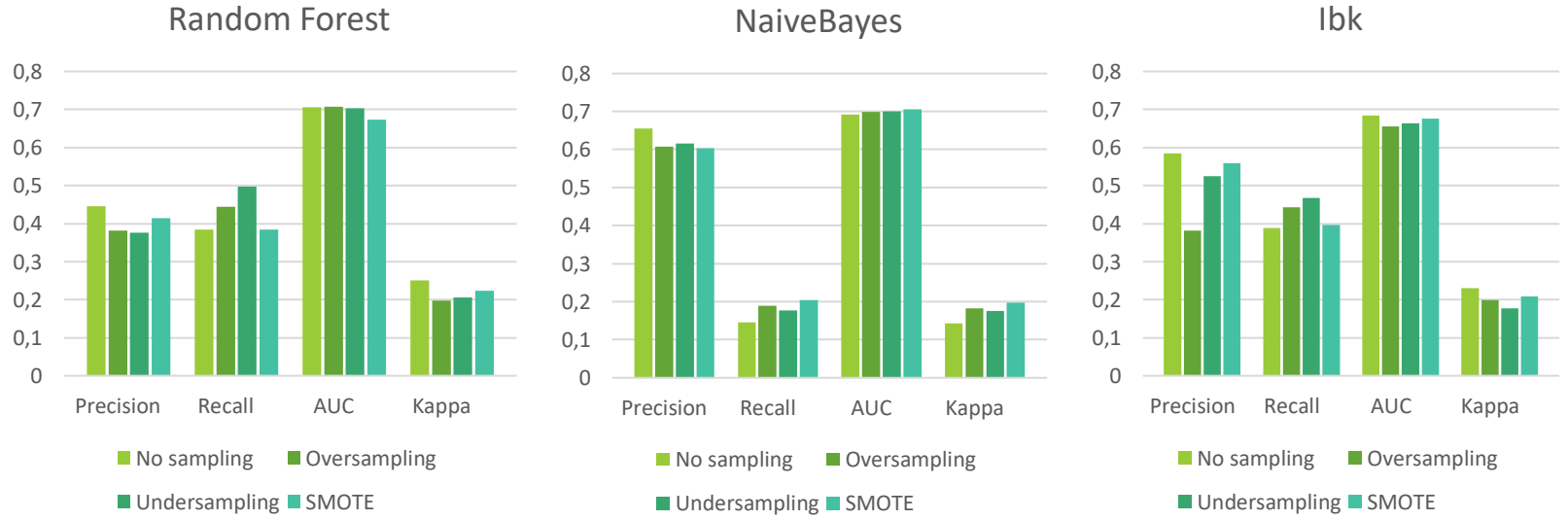
Si vogliono analizzare i benefici dati dal sampling al variare del classificatore.

- Per Avro si continuerà senza feature selection.
- Per Bookeepper si continuerà l'analisi utilizzando Best First, dati i miglioramenti ottenuti in precedenza.

Avro: no feature selection



Bookkeeper: Best First



Weka: Analisi Dell'utilizzo Del Sampling

L'applicazione di una qualsiasi tecnica di sampling, in generale, migliora la *recall* ma ha effetti negativi sulla *precision*. Ciò accade poiché le istanze che siamo interessati a stimare (le classi difettive) e cui si riferiscono le metriche riportate sono il campione minoritario.

In sintesi, quando le due classi (positive e negative) vengono portate, nel dataset di training, ad uno stesso numero di istanze, che sia per oversampling, undersampling o SMOTE, vi è un incremento nel numero di stime positive nel dataset di testing. Ciò comporta sia la presenza di un maggior numero di istanze correttamente valutate come positive (TP) ma anche un incremento degli errori commessi, ossia delle istanze FP.

Undersampling è la tecnica che dove questo fenomeno è più visibile.

AUC e *Kappa* invece hanno un lieve peggioramento dopo l'utilizzo del sampling, pur non mostrando variazioni significative.

Dal confronto non risulta esserci una tecnica di sampling migliore. La scelta tra esse dipende solamente dall'obiettivo della classificazione, ossia quale metrica si vuole massimizzare. SMOTE sembra essere quella più bilanciata, useremo questa tecnica per il resto della trattazione su Bookkeeper.

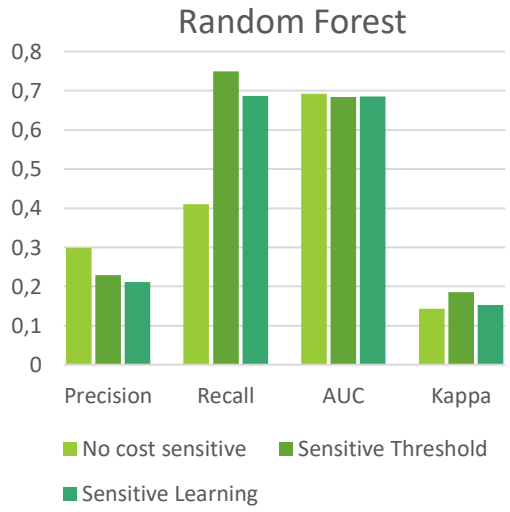
Weka:

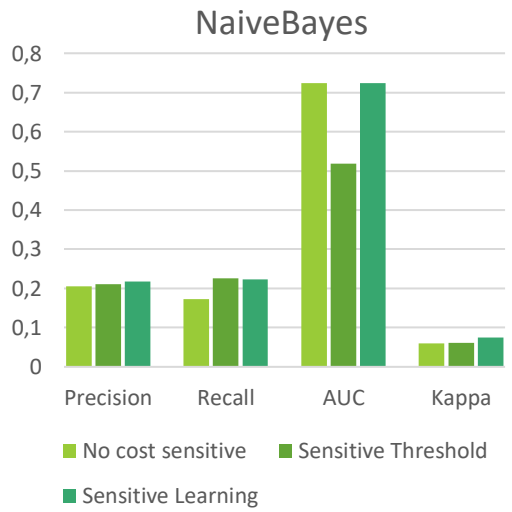
Cost Sensitive Classifier

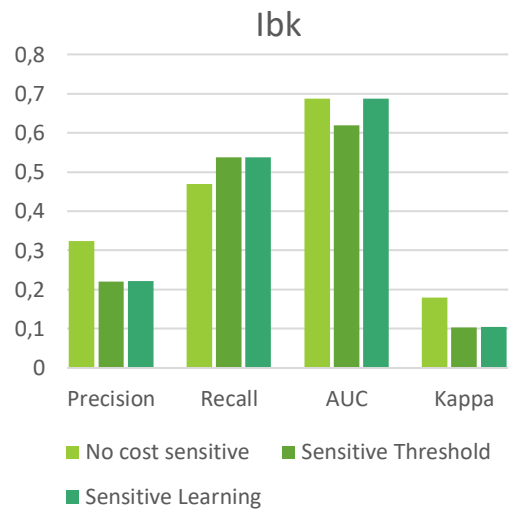
Si vuole analizzare il comportamento dei diversi classificatori al variare dei pesi assegnati ad ogni errore. In particolare:

- CFN = 10 * CFP
- Per Avro: no feature selection e no sampling.
- Per Bookeepper: Best First e SMOTE.

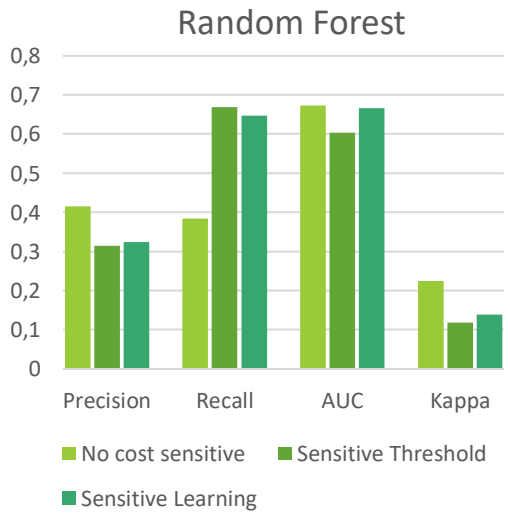
Avro: no feature selection, no sampling

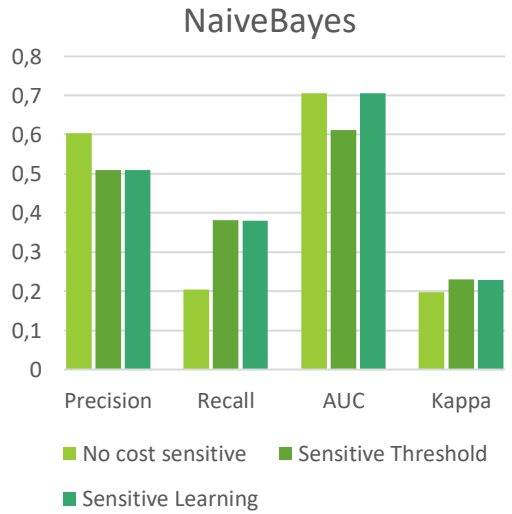


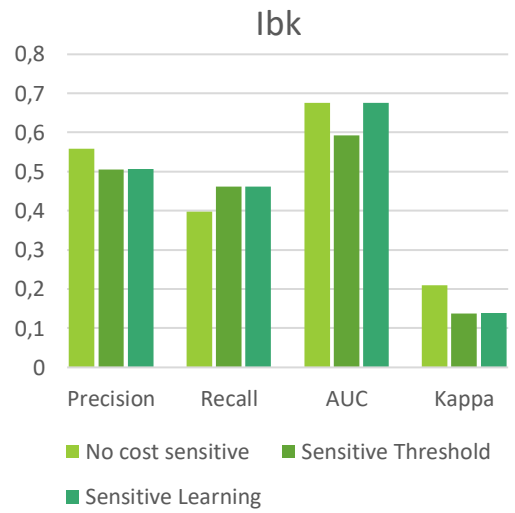




Bookkeeper, Best First, SMOTE







Weka: Cost Sensitive Classifier

Alcune volte potrebbe accadere che un tipo di errore sia più importante dell'altro. In questo contesto, l'applicazione di classificatori attenti ai costi, che sia tramite la tecnica di Sensitive Learning o di Sensitive Threshold, imposta costi maggiori ad eventuali errori di classificazione di istanze FN piuttosto che FP.

Ciò avviene in base ai valori assegnati alla matrice dei costi:

0	CFN
CFP	0

=

0	10
1	0

Come è possibile notare dai risultati ottenuti, applicare un cost sensitive classifier comporta un incremento nella *recall* a discapito del valore di *precision*. In generale anche i valori di *AUC* e *Kappa* tendono a diminuire: si fanno a fare più errori ma sono meno quelli che hanno un costo maggiore.

Tra le due tecniche quella che riesce a dare risultati migliori è sensitive learning, per la quale i valori di AUC rimangono perlopiù inalterati rispetto al caso in cui essa non venga applicata.

Sensitive Threshold invece peggiora il comportamento dei classificatori. Ciò è evidente per NaiveBayes applicato sul dataset di Avro: il valore di AUC raggiunge in questo caso un valore di poco superiore a 0,5, a significare che il classificatore si è comportato in maniera praticamente simile ad uno randomico.

Conclusioni

Dalle osservazioni effettuate possiamo concludere che:

- I tre classificatori sotto osservazione presentano comportamenti diversi a seconda del dataset cui sono applicati. Non si ha quasi mai un vincitore assoluto;
- La scelta di un classificatore piuttosto che di un altro dipende dall'obiettivo dello studio;
- Le metriche applicate di volta in volta, anche se presentando risultati diversi, modificano l'analisi dei classificatori in modo coerente (se una metrica migliora per un classificatore è molto probabile che migliori anche per un altro);
- Per i due dataset i valori di *AUC* e *Kappa* sono generalmente allineati per ogni classificatore, ad eccezione di NaiveBayes che su Avro ha un comportamento peggiore;
- Il dataset di Bookkeeper risulta avere in percentuale molte più classi positive e ciò è dimostrato dai valori in genere maggiori di *precision* e *recall* rispetto al dataset di Avro.

Links

Github

- https://github.com/ShockGiammy/Deliverable2

SonarCloud

- https://sonarcloud.io/dashboard?id=ShockGiammy_Deliverable2