Deliverable 2 – Valutazione Accuratezza Classificatori

DANILO DELL'ORCO 0300229

Roadmap

- Introduzione
- Progettazione – Costruzione del Dataset
  - Git Repository
  - Jira Project
  - Merging Git w/ Jira
  - Correzione dei Ticket
  - Metriche & Buggyness
- Progettazione – Analisi del Dataset
- Risultati Bookkeeper
- Risultati Syncope
- Conclusioni

Introduzione

- Il machine learning applicato ad un processo di sviluppo software permette di prevedere quali classi possono essere difettose
  - Ottimizzazione delle risorse impiegate nelle attività di testing
  - Prevenire la formazione di difetti, ponendo più attenzione su quelle classi che più probabilmente saranno difettose.
- La predizione viene effettuata tramite l'utilizzo dei *classificatori*.
  - Algoritmi di ML che utilizzano i dati presenti e passati per predirre i valori futuri
  - Devono essere molto accurati affinché possano effettuare una previsione il più simile possibile ai valori reali.
  - Diverse tecniche di balancing e feature selection potrebbero migliorare l'accuratezza dei classificatori utilizzati
- Lo scopo di questa attività è quello di effettuare uno studio finalizzato a valutare l'efficacia di diverse tecniche di feature selection, di balancing e di sensitivity in relazione all'accuratezza di diversi classificatori per la predizione delle difettosità delle classi:
  - 2 progetti (Bookkeeper e Syncope)
  - walk forward come tecnica di valutazione.
  - No selection / best first come feature selection.
  - No sampling / oversampling / undersampling / SMOTE come balancing
  - No cost sensitive / Sensitive Threshold / Sensitive Learning (CFN = 10 * CFP)
  - RandomForest / NaiveBayes / Ibk come classificatori.

Progettazione

- Si è sviluppato un programma Java che permette di eseguire la valutazione dell'accuratezza dei classificatori. Il programma è costituito da due componenti principali

1. Costruzione del dataset
   - Raccolta di tutti i dati necessari e generazione del dataset da fornire all'algoritmo di machine learning

2. Analisi del dataset
   - Suddivisione del dataset in *Training Set* e *Testing Set* secondo *walk worward*.
   - Training dei classificatori, applicando le tecniche di *cost sensitivity*, *feature selection* e *sampling*.
   - Valutazione della loro accuratezza.

- Tale programma è stato progettato in modo da poter poter analizzare progetti differenti
  - File *Parameters.Java* contiene tutti i parametri relativi ai progetti che si vogliono analizzare.
  - File *paths.config* contiene le cartelle in cui si vogliono clonare le repository

Progettazione – Costruzione Dataset

1. Si ottiene una copia locale della repository git considerata, tramite *clone* o *checkout*.
2. Si ottengono da Git tutte le release, e si mantengono soltanto quelle presenti anche su Jira.
3. Si ottengono da Git le informazioni su tutti commit effettuati.
4. Si ottengono da Jira le varie releases del progetto.
5. Si ottengono da Jira tutte le informazioni relative ai Ticket di tipo Bug Fix.
6. Si calcola *IV*, *OV* e *FV* per ogni ticket: questi dati possono essere presenti in Jira, oppure possono essere calcolati applicando l'approccio *Incremental* di Proportion.
7. Si mantengono soltanto i ticket che hanno un relativo commit associato su Git. Ad ogni commit di tipo Fix Bug viene associato il relativo ticket Jira.
8. Si ottengono da GitHub tutti i file presenti al rilascio di ogni release, ottenendo la lista delle *classi java*.
9. Per ogni classe individuata ed in ogni release, si calcolano *metriche* selezionate e la *buggyness*.
10. Si genera il dataset, scrivendo tutte le classi individuate e le relative metriche su un file .csv.

Git Repository

- Per gestire ed ottenere tutti i dati dalla repository GitHub, è stata utilizzata la libreria *JGit*. Si apre innanzitutto una copia locale della repository, tramite *clone()* o *checkout()*

- Si ottiene la lista di tutte le *release* sfruttando i *tag* presenti su Git

```
tagList = git.tagList().call();
RevWalk walk = new RevWalk(this.git.getRepository());

for (Ref tag : tagList) {
    String tagName = tag.getName();
    String releaseName = tagName.substring((releaseFilter + Parameters.TAG_FORMAT).length());
    ...
    RevCommit c = walk.parseCommit(tag.getObjectId());
    Date releaseDate = DateHandler.getDateFromEpoch(c.getCommitTime() * 1000L);
    String tagName = tag.getName();
    String releaseName = tagName.substring((releaseFilter + Parameters.TAG_FORMAT).length());
    GitRelease release = new GitRelease(this.git, c, releaseName, releaseDate);
    ...
```

  - Per ogni tag individuato si ricava il commit associato, e si istanzia un nuovo oggetto **GitRelease**

- Si ottiene la lista di tutti i *commit*, tramite il comando *log()* messo a disposizione da *JGit*.

```
LogCommand logCommand = this.git.log();
Iterable<RevCommit> logCommits = logCommand.call();
...
for (RevCommit c : logCommits) {
    Date date = DateHandler.getDateFromEpoch(c.getCommitTime() * 1000L);
    ObjectId parentID = null;
    ...
    GitCommit commit = new GitCommit(c.getId(), date, c.getFullMessage());
    this.commitList.add(commit);
}
```

  - Per ogni commit individuato viene istanziato un nuovo oggetto **GitCommit**

Jira Project

- Per individuare lo storico dei difetti è stato utilizzata la piattaforma ***Jira***, sfruttando le REST API che questa mette a disposizione.
- Si ottiene una lista di tutte le *release* presenti su *Jira*.

```
json = JsonHandler.readJsonFromUrl(this.url);
JSONArray releasesList = json.getJSONArray("versions");
    for (int i = 0; i < releasesList.length(); i++) {
        JSONObject tempRelease = releasesList.getJSONObject(i);
        if (tempRelease.has("releaseDate") && tempRelease.getBoolean(Parameters.RELEASED_JSON)) {
            LocalDate releaseDate = LocalDate.parse(tempRelease.getString("releaseDate"));
            JiraRelease release = new JiraRelease();
            release.setName(tempRelease.getString(Parameters.NAME_JSON));
            release.setReleaseDate(releaseDate);
            allRelease.add(release);
        }
    return allRelease;
```

- Vengono scartate le release che non hanno una data, o che non hanno il flag «Released»
- Per ogni release valida ottenuta da Jira, viene istanziato un oggetto JiraRelease

- Si ottiene una lista di tutti i *ticket* di tipo *Bug Fixed* relativi al progetto considerato.

```
JiraTicket t = new JiraTicket();
    t.setId(tempTicket.getString("id"));
    t.setName(tempTicket.getString("key"));
    t.setResolutionDate(DateHandler.stringToDate(fields.getString("resolutiondate")));
    t.setCreationDate(DateHandler.stringToDate(fields.getString("created")));
    t.setFixedVersions(fixedVersions);
    t.setAffectedVersions(versions);
```

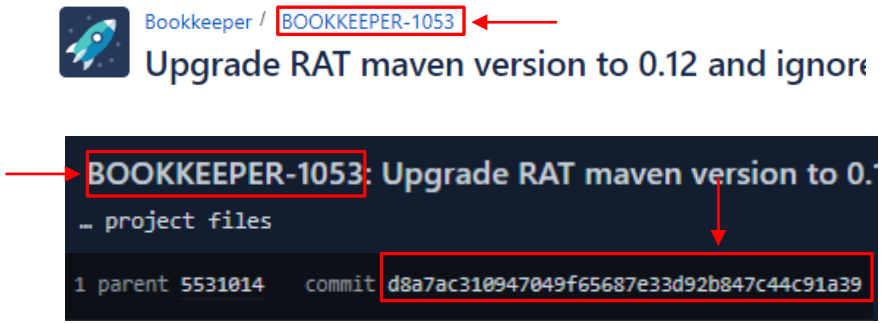
- Per ogni ticket trovato, viene istanziato un oggetto JiraTicket.
  - **OV**: *Opening Version*, ricavata tramite il campo *ResolutionDate*.
  - **FV**: *Fixed Versions*, ricavate tramite il campoi *FixedVersions*.
  - **IV**: *Injected Version*, ricavato come la più vecchia tra le *Affected Versions*.

Merging Jira w/ GitHub (1/2)

- Dopo aver ottenuto *release*, *ticket* e *commit*, si effettua un mapping tra i dati ricavati da *Jira* e quelli ricavati tramite *Git/GitHub*

- Si mantengono soltanto i commit che hanno un ticket associato, ed i ticket che hanno un relativo commit.
  - Si verifica se il *messaggio del commit* contiene *l'identificativo del ticket*
  - Si ottiene una lista contenente tutti i *GitCommit* di tipo *Bug Fixed*
  - Non vengono considerati i *ticket* che non hanno effettivamente risolto un problema tramite un *commit* su GitHub

```
Iterator<JiraTicket> iterator = tickets.iterator();
while (iterator.hasNext()) {
    JiraTicket t = iterator.next();
    for (GitCommit c : this.commitList) {
        if (c.hasTicketName(t.getName())) {
            filtered.add(c);
            c.setFixCommit(true);
            c.setTicket(t);
            founded = true;
        }
    }
    if (!founded) {
        iterator.remove();
    }
}
```



Mapping tra JiraTicket e GitCommit

Merging Jira w/ GitHub (2/2)

- Si mantengono soltanto le release presenti sia su *Jira* che su *GitHub*
  - Non è possibile identificare una *Affected Version* su una release se questa non è presente su Jira.
  - Non è possibile ottenere la lista delle classi se la release non esiste su GitHub.

```
for (JiraRelease jR : jiraReleases) {
    for (GitRelease gR : gitReleases) {
        if (gR.getName().equals(jR.getName())) {
            gR.fetchClassList();
            commonReleases.add(gR);
            jR.setReleaseDate(DateHandler.convertToLocalDate(gR.getDate()));
            this.commitList.add(gR.getCommit());
        }
}
```

- Per ogni release considerata, si cercano i files *.java* presenti nella release
  - Ogni release ha un relativo commit associato
  - Tramite *JGit* si ottengono tutti i file presenti al momento del commit
  - Per ogni classe individuata si istanzia un oggetto *ProjectClass*, che mantiene tutti i dati e le metriche della classe

```
while (treeWalk.next()) {
String classPath = treeWalk.getPathString();
    if (classPath.contains(Parameters.FILTER_FILE_TYPE)) {
        String className = PathHandler.getNameFromPath(classPath);
        ProjectClass projectClass = new ProjectClass(classPath,className,this);

        objectId = treeWalk.getObjectId(0);

    // Calcolo e setto la size della classe
        Metrics metrics = new Metrics();
        metrics.calculateSize(objectId, reader);
        projectClass.setMetrics(metrics);
        classList.add(projectClass);
    }
}
```

Correzione dei ticket (1/2)

- Prima di procedere con l'analisi della buggyness è necessario correggere i ticket con informazioni parziali, e scartare quelli con informazioni errate.
  - Il ticket contiene le informazioni inserite dallo sviluppatore, e non sono necessariamente corrette
  - Si vuole costruire un dataset il più accurato possibile, basato soltanto su informazioni coerenti e complete
  - Si effettuano diverse operazioni di *"pulizia"* dei ticket
- La **prima operazione di pulizia** riguarda la *lista di fixed versions* ottenuta tramite le API di Jira.
  - Nel campo *fixVersions* Jira restituisce una lista di versioni, in alcuni casi però questa lista è vuota oppure contiene più di un elemento.
  - **Lista vuota**: si ricava *FV* come la prima release successiva alla data di chiusura del ticket.
  - **Lista con più fixed version**: si considera come FV la più vecchia tra tali versioni.
- Se al termine di questa fase non si è riusciti ad individuare una *fixed version* il ticket viene scartato
  - Senza tale informazione non si può individuare la release in cui è stato risolto il bug, e quindi non si riesce ad ottenere la lista delle Affected Version

Correzione dei ticket (2/2)

- La **seconda operazione di pulizia** consiste nel considerare i soli ticket che presentano informazioni corrette riguardo **IV**, **OV** e **FV**

- Vengono rimossi dalla lista tutti i ticket in cui:
  1. **IV > OV = FV**. *Injected Version* non coerente, in quanto il bug sarebbe stato introdotto dopo averlo già fixato.
  2. **IV = OV = FV**. Non posso ricavare nessuna AV poiché in questa versione il bug è stato sia introdotto che risolto
  3. **OV = FV** e **IV** non disponibile. Anche applicando *Proportion* si ricadrebbe nella precedente casistica

- Al termine della seconda operazione di pulizia, restano soltanto i ticket in cui:
  - **IV < OV ≤ FV**: conoscendo IV, FV, e la lista di tutte le *release* Jira, possiamo facilmente *ricavare* la lista completa delle *Affected Version*
  - **OV < FV**: possiamo effettuare una *predizione* dell'*Injected Version* applicando il metodo *Proportion*

Proportion

- In alcuni ticket non viene specificata nessuna Injected Version, e di conseguenza non è possibile ricavare la lista delle Affected Versions.
- ***Proportion*** è una tecnica che permette di effettuare una stima dell'Injected Version di un ticket
  - L'idea alla base di tale metodo è che vi sia una certa costante di proporzionalità tra il numero di revisioni nell'intervallo [IV;FV] ed il numero di revisioni nell'intervallo [OV;FV]
  - Tale costante è definita come $P = \frac{FV - IV}{FV - OV}$ ed è costante per tutti i bug di un progetto
- La *costante P* viene calcolata utilizzando soltanto ticket validi raccolti da Jira
- Per ogni ticket T senza una Injected Version valida si effettua una predizione sfruttando la costante di proporzionalità P.
  - $IV(T)_{predicted} = FV(T) - P * (FV(T) - OV(T))$
- Vengono assegnate le *Affected Versions* ad ogni ticket T
  - $AV(T) = [IV(T)_{predicted}; FV(T))$

Metriche (1/2)

- Corretti tutti i ticket è possibile procedere con il calcolo delle metriche delle varie classi individuate
  - Si analizzano tutti i commit presenti nella lista ottenuta in precedenza
- Sono state prese in considerazione le seguenti metriche:
  1. **Size –** numero di linee di codice
  2. **locTouched –** numero di linee di codice modificate
  3. **locAdded –** numero di linee di codice aggiunte
  4. **maxLocAdded –** numero massimo di linee di codice aggiunte tra tutte le release
  5. **avgLocAdded –** media del numero di linee di codice aggiunte tra tutte le release
  6. **chgSetSize –** numero di file committed insieme alla classe
  7. **maxChgSetSize –** numero massimo di file committed insieme alla classe tra tutte le release
  8. **avgChgSetSize –** numero medio di file committed insieme alla classe tra tutte le release
  9. **numberRevisions –** numero di revisioni della classe nella release corrente
  10. **numberBugFixes –** numero di bug fixati sulla classe nella release corrente
  11. **nAuth –** numero di autori che hanno apportato modifiche alla classe
  12. **Age –** differenza in settimane tra la data della release e la data di creazione della classe

Metriche (2/2)

- Per analizzare le modifiche introdotte da ogni commit, si utilizza la classe *DiffFormatter* di *JGit*
  - *scan()* fornisce una lista di oggetti *DiffEntry*, che rappresentano una differenza tra il commit ed il suo parent
  - Iterando su tali oggetti vengono calcolate tutte le metriche relative alle classi toccate
    - **ADD:** aggiunta di una classe, permette di calcolare l'age
    - **DELETE:** rimozione di una classe
    - **MODIFY:** modifica di una classe, permette di calcolare le metriche di Loc
    - **COPY:** duplicazione di una classe
    - **RENAME:** rinominazione di una classe

```
GitDiff gitDiff;

DiffFormatter diffFormatter = new DiffFormatter(DisabledOutputStream.INSTANCE);
diffFormatter.setRepository(git.getRepository());
diffFormatter.setDiffComparator(RawTextComparator.DEFAULT);
ObjectId origin = commit.getParentID();
List<DiffEntry> diffEntries = diffFormatter.scan(origin, commit.getId());
```

- Per calcolare la **buggyness** si analizzano i *DiffEntry* relativi si soli commit di tipo *bug fix*
  - Se una classe viene *modificata* in un commit di tipo *bug fix*, vuol dire che la modifica introdotta ha risolto un *bug*, e quindi che la classe era precedentemente buggy.
  - Si accede alle *Affected Versions* riportate nel ticket associato al commit, e si imposta la classe come *buggy* in tutte le release contenute nella lista di *AV*

```
public void setBuggynessWithAV(GitCommit fixCommit,String pathClass) {
    JiraTicket fixTicket = fixCommit.getTicket();
    List<JiraRelease> affectedVersions = fixTicket.getAffectedVersions();

    for (JiraRelease av:affectedVersions) {
        GitRelease gitAv = getReleaseByName(av.getName());
        ProjectClass projClass = gitAv.getProjectClass(pathClass);
            if (projClass!=null) {
                projClass.setBuggy(true);
            }
        }
}
```

Analisi Dataset

- Terminata l'analisi dei commit viene generato il dataset, scrivendo tutte le classi e le relative metriche all'interno di un file *.csv*
  - **Bookkeeper**: 5201 classi / 967 buggy (18.59%)
  - **Syncope**: 28495 / 7142 buggy (25.06%)

- Per valutare le prestazioni del classificatore, sono state utilizzate le *API Java* messe a disposizione da **Weka**.

1. Si converte il dataset da *.csv* a *.arff*, e si carica all'interno di Weka.

```
CSVHandler.convertCSVtoARFF(Parameters.OUTPUT_PATH + projectName + Parameters.WEKA_CSV);
Instances dataset = CSVHandler.loadFileARFF(Parameters.OUTPUT_PATH + projectName + Parameters.DATASET_ARFF);
weka.setDataset(dataset);
```

2. Per ogni classificatore, per ogni metodo di feature selection, per ogni metodo di resampling, e per ogni tipologia di sensitivity si esegue una run di *walk forward*, e si calcolano le metriche di accuratezza specificate

```
for (String classifierName : this.classifiers) {
    for (String featureSelectionName : this.featureSelectionMethods) {
        for (String resamplingMethodName : this.resamplingMethods) {
            for (String costSensitiveMethod : this.costSensitiveMethods) {
                for (int i = 2; i < releasesNumber; i++) {
                    WekaMetrics result = new WekaMetrics(classifierName,featureSelectionName,
                resamplingMethodName, costSensitiveMethod);
                    Instances[] trainTest = splitTrainingTestSet(getDataset(), i);
                    runWalkForwardConfiguration(trainTest, result, i);
                }
```

   - Si calcolano i valori di *recall*, *precision*, *area under ROC* e *kappa*
   - Si esegue una run per ogni possibile combinazione di tecniche utilizzate
   - Complessivamente si eseguono **72 run** di walk forward

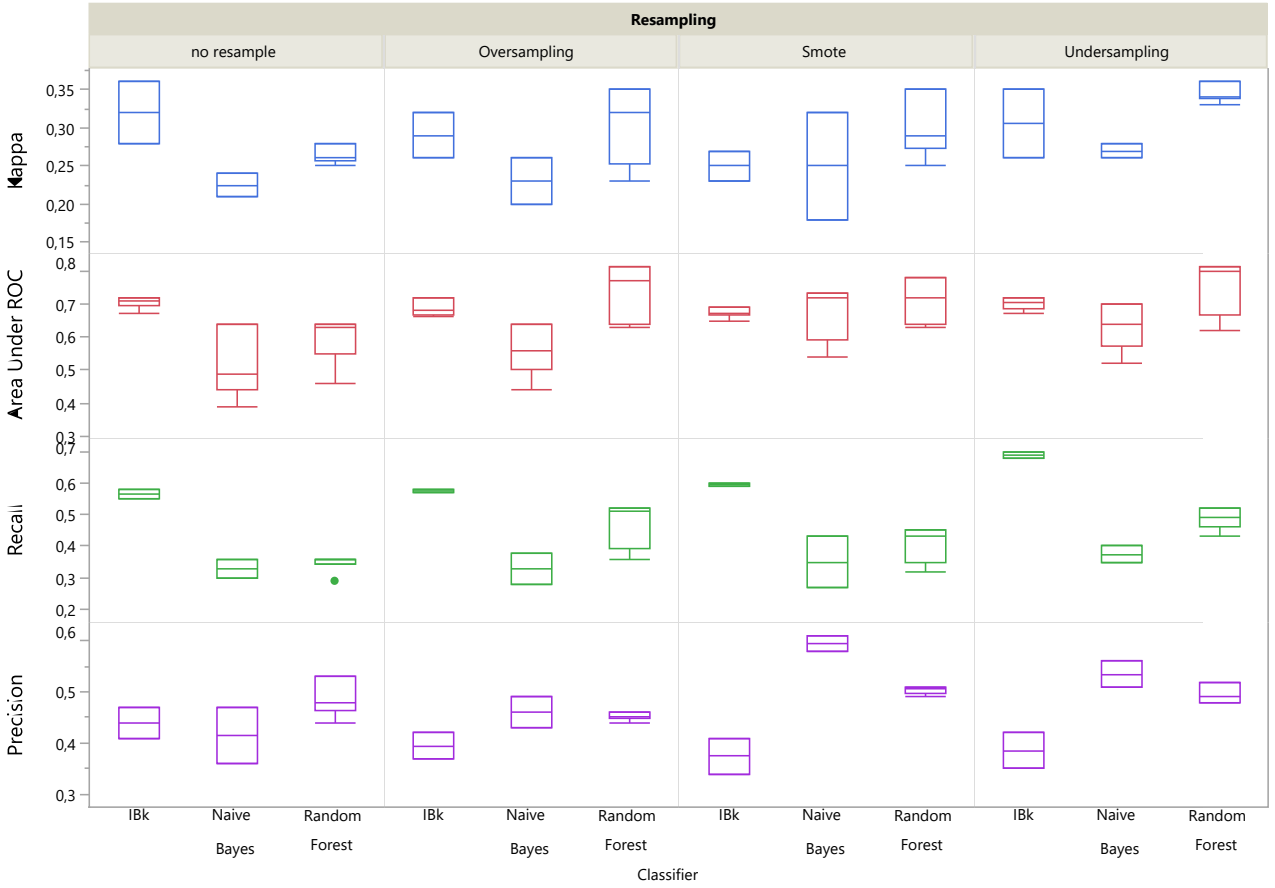
3. Si scrivono i risultati di ogni run all'interno di un file csv.
   - Permette di analizzare le prestazioni di ogni classificatore al variare delle configurazioni di sampling, sensitivity e feature selection

Bookkeeper – Feature Selection

Accuratezza dei classificatori al variare dei metodi di feature selection

- **Ibk** ha una maggiore accuratezza utilizzando *best first*
  - Best First: *precision* **0.43** - *recall* **0.61** - *AUC* **0.71** - *kappa* **0.37**
  - No Feature: *precision* 0.36 – *recall* 0.60 - *AUC* 0.67 - *kappa* 0.25
- **Naive Bayes** ha performance migliori *senza feature selection*
  - Best First: *precision* 0.47 - *recall* 0.30 - *AUC* 0.55 - *kappa* 0.21
  - No Feature: *precision* **0.53** - *recall* **0.39** - *AUC* **0.64** - *kappa* **0.28**
- **Random Forest** offre prestazioni *molto simili* a prescindere dall'utilizzo o meno di feature selection
  - Best First: *precision* 0.49 - *recall* 0.44 - *AUC* 0.71 - *kappa* 0.30
  - No feature: *precision* 0.48 - *recall* 0.42 - *AUC* 0.69 - *kappa* 0.31
- L'impatto della *feature selection* sull'accuratezza dipende dal classificatore considerato

Bookkeeper - Resampling



Accuratezza dei classificatori al variare dei metodi di resampling

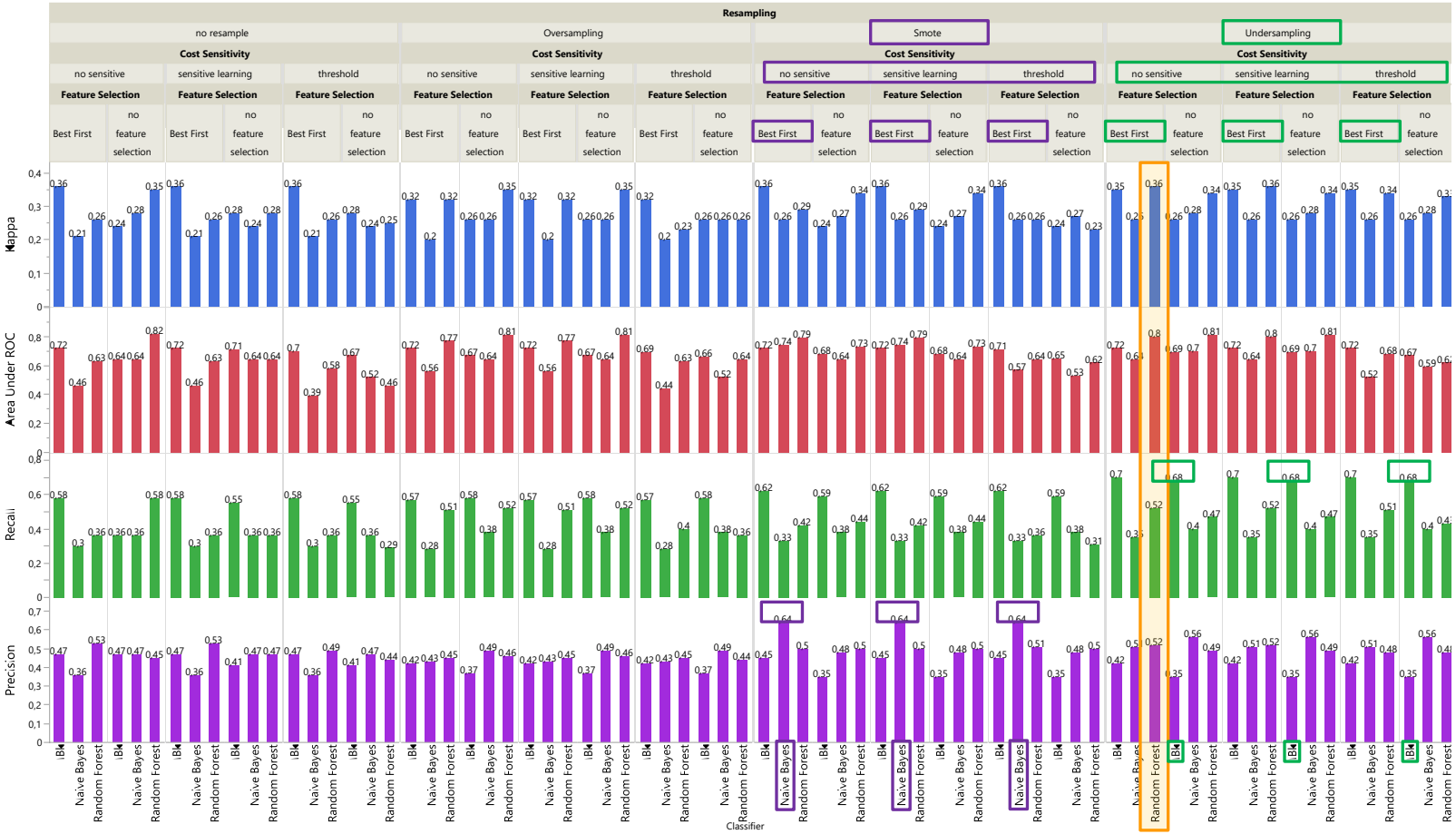
- **Ibk** ha generalmente un'accuratezza maggiore *senza balancing*
  - *precision* 0.44 - *recall* 0.56 - *AUC* 0.71 - *kappa* 0.32
  - *Undersampling* migliora notevolmente la *recall* da 0.56 a 0.69
- **Naive Bayes** ha performance migliori con *smote*
  - *precision* 0.595 - *recall* 0.35 - *AUC* 0.675 - *kappa* 0.25
  - *Undersampling* migliora leggermente la *recall* da 0.35 a 0.38
- **Random Forest** offre la maggiore accuratezza utilizzando la tecnica di *undersampling*
  - *precision* 0.497 - *recall* 0.487 - *AUC* 0.753 - *kappa* 0.345
- Per questo dataset *oversampling* non rappresenta mai una scelta vantaggiosa, in quanto non migliora nessuna metrica a prescindere dal classificatore considerato.
- A prescindere dal classificatore, la tecnica *undersampling* massimizza sempre la *recall*

Bookkeeper – Cost Sensitivity

Accuratezza dei classificatori al variare dei metodi di cost sensitivity

- I box plot in figura mostrano come tutti i classificatori abbiano una accuratezza simile, a prescindere dalla tecnica di cost sensitivity utilizzata
  - La cost senitivity sembra avere in media poco impatto sui risultati della classificazione
- L'unico classificatore che mostra risultati differenti è **Random Forest**, se si utilizza la tecnica *Sensitive Threshold*
  - Peggiorano *recall*, *AUC* e *kappa*
- *Random Forest / No Sensitive*
  - *precision* 0.49 - *recall* 0.48 - *AUC* 0.77 - *kappa* 0.33
- *Random Forest / Sensitive Learning*
  - *precision* 0.49 - *recall* 0.45 - *AUC* 0.75 - *kappa* 0.32
- *Random Forest / Sensitive Threshold*
  - *precision* 0.48 - *recall* 0.38 - *AUC* 0.61 - *kappa* 0.27

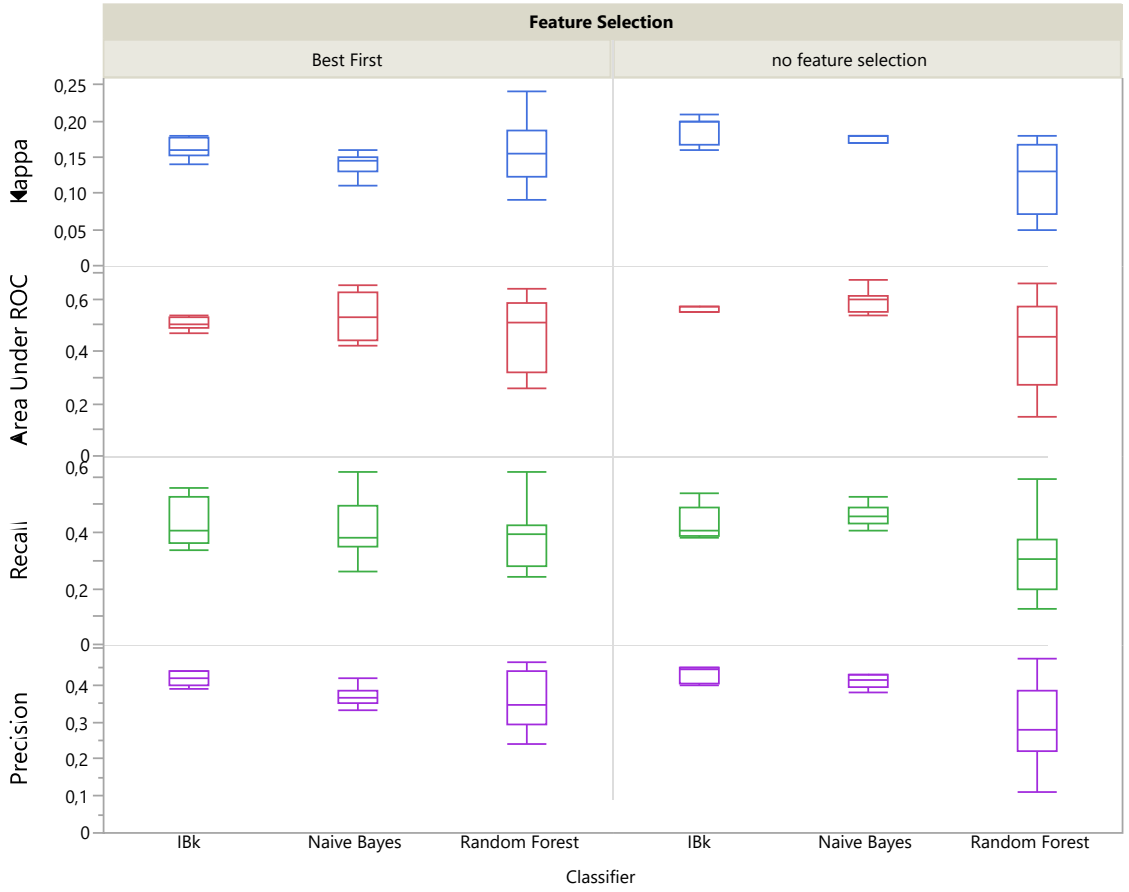
Bookkeeper – Risultati Complessivi



Accuratezza dei classificatori per ogni configurazione considerata

- Se l'obiettivo è quello di massimizzare *precision*, ***Naive Bayes*** offre le migliori prestazioni utilizzando *smote / best first*, a prescindere dalla politica di sensitivity
  - Precision 0.64
  - Tuttavia recall bassa di 0.33
- Se l'obiettivo è massimizzare *recall*, le prestazioni migliori sono offerte da ***IBK*** con *undersampling/best first* (*0.72*)
  - Si hanno buoni valori anche per le altre metriche
  - Precision 0.42 - AUC 0.69 - Kappa 0.35
- La migliore *accuratezza media* viene raggiunta utilizzando ***Random Forest*** con *undersampling / best first / no sensitive*
  - Precision 0.52 - Recall 0.52 - AUC 0.8 - Kappa 0.36
- A prescindere dalle tecniche utilizzate, *Random forest* offre sempre dei valori accettabili di *precision [0.46;0.53]*

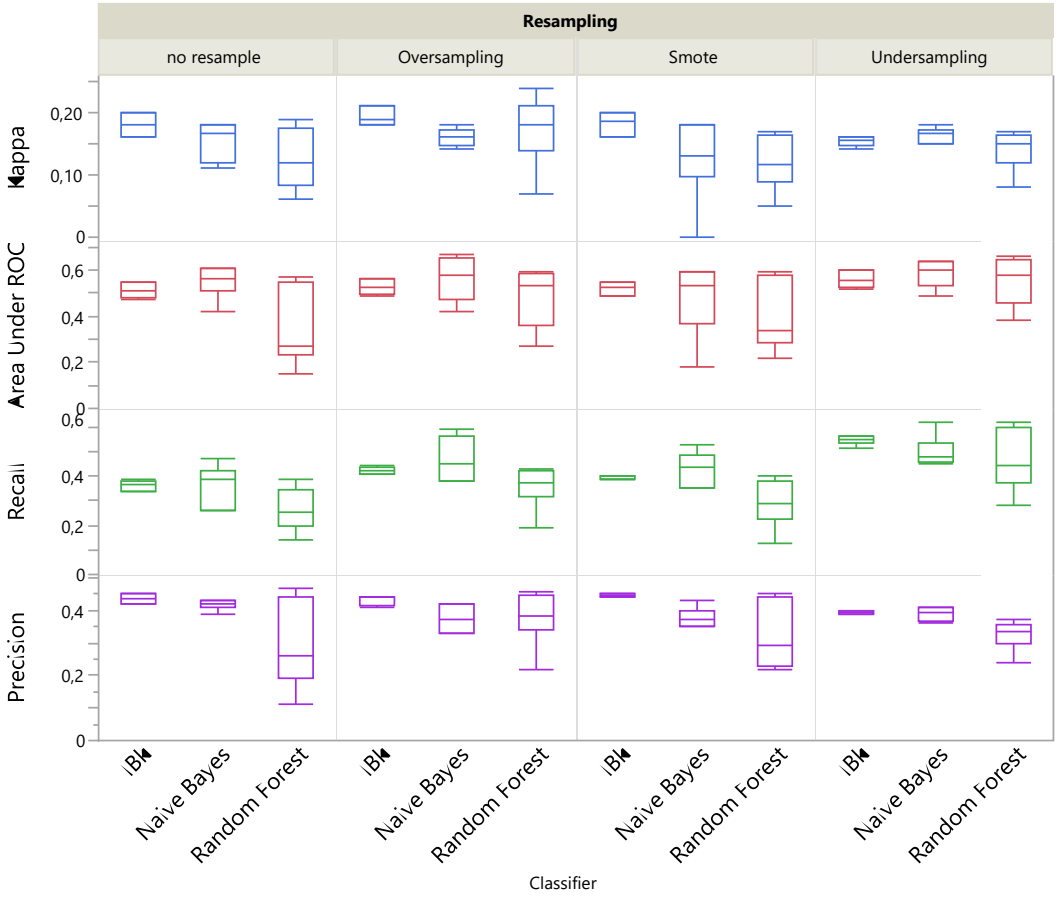
Syncope - Feature Selection



Accuratezza dei classificatori al variare della tecnica di feature selection

- **Ibk** ha una maggiore accuratezza senza l'utilizzo di feature selection, in termini di *area under ROC e kappa*
  - Best First: *precision* 0.42 – *recall* 0.43 - *AUC* **0.50** - *kappa* **0.16**
  - No Feature: *precision* 0.42 - *recall* 0.43 - *AUC* 0.56 - *kappa* 0.19
- **Naive Bayes** ha performance leggermente migliori *senza feature selection*
  - Best First: *precision* 0.37 - *recall* 0.41 - *AUC* 0.53 - *kappa* 0.15
  - No Feature: *precision* **0.41** - *recall* **0.47** - *AUC* **0.56** - *kappa* **0.16**
- **Random Forest** beneficia dell'utilizzo di best first come tecnica di feature selection
  - Best First: *precision* **0.36** - *recall* **0.38** - *AUC* **0.47** - *kappa* **0.16**
  - No feature: *precision* 0.30 - *recall* 0.30 - *AUC* 0.42 - *kappa* 0.12

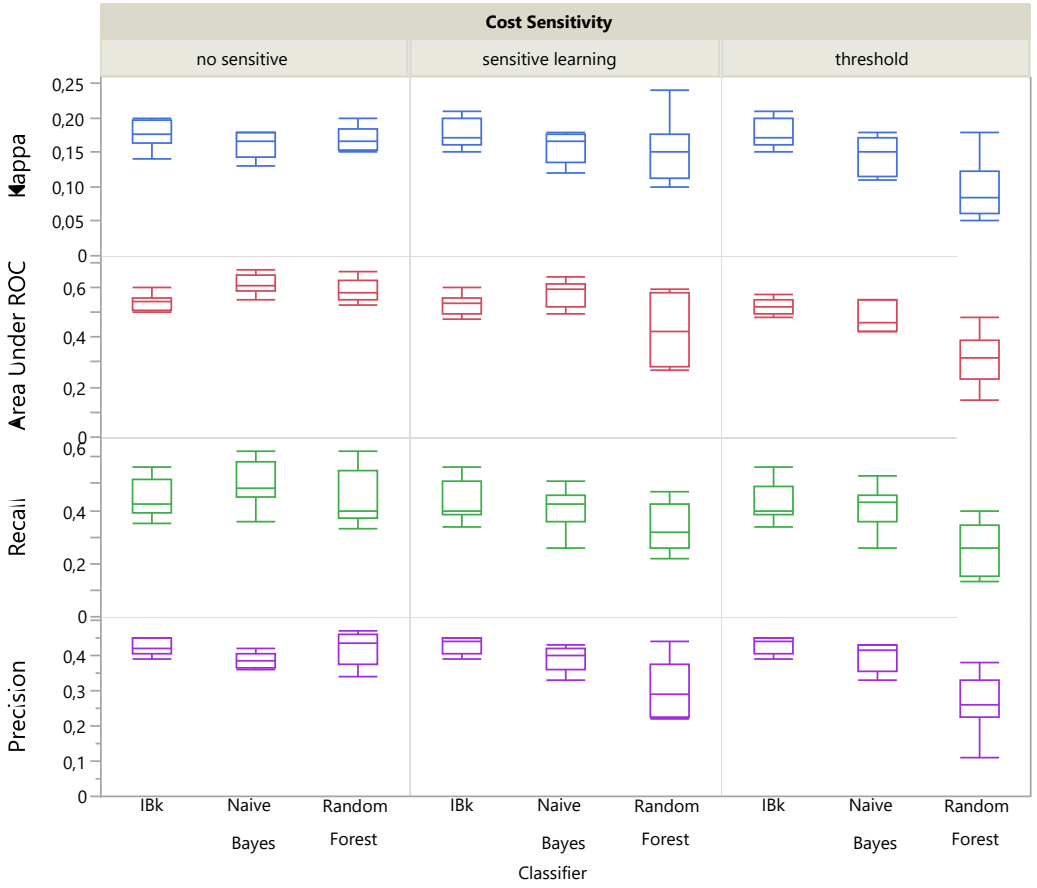
Syncope - Resampling



Accuratezza dei classificatori al variare della tecnica di Resampling

- **Ibk** offre i migliori livelli di *precision* e *kappa* con *smote*, mentre *undersampling* permette di massimizzare sia *recall* che *area under ROC*
  - Smote: *precision* **0.45** – *recall* 0.40 - *AUC* 0.52 - *kappa* **0.18**
  - Undersampling: *precision* 0.39 - *recall* **0.55** - *AUR* **0.56** - *kappa* 0.15
- **Naive Bayes** massimizza la *precision senza resampling*, mentre *undersampling* offre risultati migliori per le restanti metriche di accuratezza
  - No Resampling: *precision* **0.42** – *recall* 0.36 - *AUC* 0.55 - *kappa* 0.15
  - Undersampling: *precision* 0.39 - *recall* **0.5** - *AUC* **0.59** - *kappa* **0.16**
- **Random Forest** ha migliore *precision* utilizzando *oversampling*, mentre la *recall* migliora utilizzando *undersampling*
  - Oversampling: *precision* **0.38** – *recall* 0.36 - *AUC* 0.48 - *kappa* **0.17**
  - Undersampling: *precision* 0.32 - *recall* **0.46** - *AUC* **0.55** - *kappa* 0.14
- Ogni tecnica permette di massimizzare qualche metrica di accuratezza relativamente ad uno specifico classificatore
- In generale *undersampling* permette sempre di massimizzare la recall

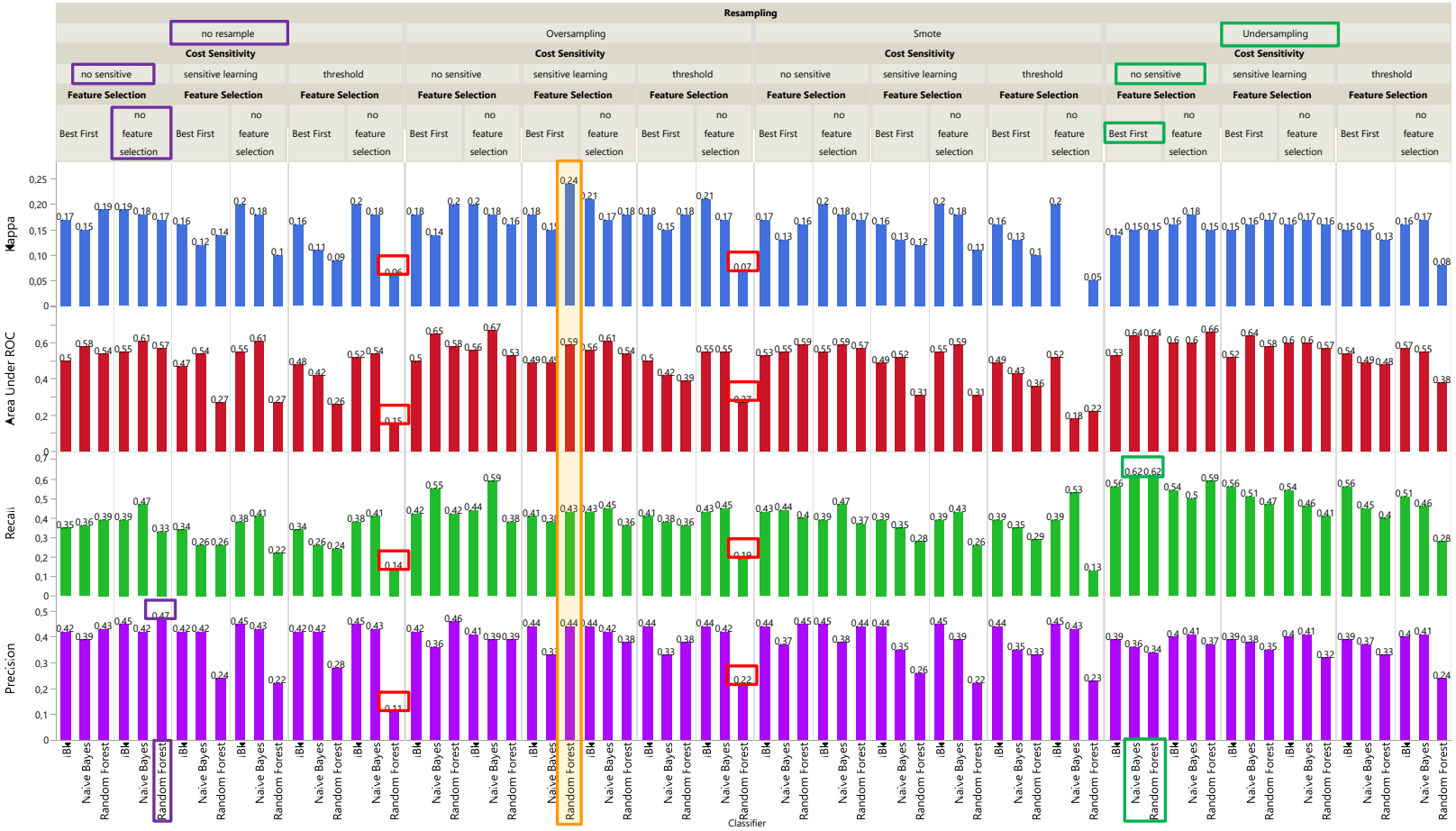
Syncope - Cost Sensitivity



Accuratezza dei classificatori al variare dell'approccio di cost sensitivity

- **Ibk** ha prestazioni molto simili per ognuna delle politiche considerate
  - *precision* 0.43 - *recall* 0.44 - *AUC* 0.53 - *kappa* 0.178
- **Naive Bayes** ha un'accuratezza maggiore *senza* l'utilizzo di *cost sensitive classifier*, soprattutto in termini di *recall* e *area under ROC*
  - No Sensitive: *precision* 0.38 – *recall* **0.50** - *AUC* **0.61** - *kappa* 0.16
  - Sensitive Threshold: *precision* 0.39 – *recall* 0.41 - *AUC* 0.44 - *kappa* 0.13
- **Random Forest** ottiene i risultati migliori risultati se non viene utilizzato *cost sensitive*
  - No Sensitive: *precision* **0.42** – *recall* **0.44** - *AUC* **0.58** - *kappa* **0.17**
  - Sensitive Learning: *precision* 0.30 – *recall* 0.33 - *AUC* 0.43 - *kappa* 0.15
- L'approccio migliore a livello generale è quello *senza cost sensitivity*, in quanto migliora le prestazioni di *Naive Bayes* e *Random Forest*, mantenendo inalterate quelle di *IBK*

Syncope – Risultati Complessivi



Accuratezza dei classificatori per ogni configurazione considerata

- Se l'obiettivo è quello di massimizzare *precision*, ***Random Forest*** offre le migliori prestazioni senza utilizzare nessuna tecnica per migliorare l'accuratezza
  - *Precision* 0.47
  - Tuttavia si ha una bassa *recall* di 0.33
- Se l'obiettivo è massimizzare *recall*, le prestazioni migliori sono offerte da ***Random Forest*** e ***Naive Bayes*** utilizzando *best first/ undersampling / no sensitive*
  - Recall 0.62
  - *Precision* 0.35 - *AUC* 0.64 - *Kappa* 0.15
- La migliore *accuratezza media* viene raggiunta utilizzando ***Random Forest*** con *oversampling / best first / sensitive learn*
  - *Precision* 0.44 - *Recall* 0.43 - *AUC* 0.59 - *Kappa* 0.24
- E' sconsigliato l'utilizzo della configurazione *Random Forest / Cost Sensitive / No Feature Selection*, in quanto si hanno metriche di accuratezza molto basse che oscillano tra 0.11 e 0.27

Conclusioni

- Per entrambi i dataset *undersampling* migliora i valori di *recall*, abbassando invece i valori di *precision*:
  - Vengono ridotte le istanze negative, per cui il classificatore tende a classificare più istanze come positive (maggiore recall)
  - Classificando più istanze come positive aumenta anche il numero di falsi positivi individuati (minore precision)
- Per entrambi i dataset, generalmente utilizzare *best first* come feature selection migliora l'accuratezza
  - Sono presenti diverse features che hanno una bassa correlazione con la variabile di interesse, e che portano quindi a delle classificazioni sbagliate
- Per entrambi i dataset si hanno valori di *kappa* sempre positivi
  - Tutti i classificatori, a prescindere dalla configurazione, operano sempre meglio di un classificatore dummy
- Per **Bookkeeper** non esiste un classificatore sempre migliore, in quanto ognuno permette di raggiungere dei requisiti specifici in base alle tecniche utilizzate
- Per **Syncope** il classificatore più adeguato risulta essere *Random Forest*, poiché offre migliori prestazioni rispetto a *IBK* e *Naive Bayes*, sia a livello generale, che nel massimizzare delle metriche specifiche.
- A parità di configurazioni, si ottengono dei valori di adeguatezza più elevati su *Bookkeeper* rispetto a *Syncope*.
  - Questo testimonia come le metriche di adeguatezza siano fortemente dipendenti dal dataset utilizzato

Links

- Repository GitHub: https://github.com/danilo-dellorco/deliverable2
- Travis CI: https://travis-ci.com/github/danilo-dellorco/deliverable2
- SonarCloud: https://sonarcloud.io/dashboard?id=danilo-dellorco_deliverable2