Deliverable 2

Studio Dell’ Accuratezza Nella Predizione

GABRIELE QUATRANA 0306403

Sommario

- Introduzione
- Progettazione
- Risultati
  - Bookkeeper
  - Tajo
- Conclusioni
- Riferimenti

Introduzione

- L'obiettivo di questo deliverable è quello di eseguire uno studio finalizzato a misurare l'effetto di tecniche di sampling, classificazioni sensibili al costo e feature selection sull'accuratezza di modelli predittivi per la localizzazione di bug nel codice di applicazioni open-source.
- In particolare sono stati presi in considerazione:
  - Due progetti: **Bookkeeper** e **Tajo**.
  - **Walk forward** come tecnica di validazione.
  - No selection / **Best first** come feature selection.
  - No sampling / **Oversampling** / **Undersampling** / **SMOTE** come balancing.
  - No cost sensitive / **Sensitive Threshold** / **Sensitive Learning** (CFN = 10 * CFP) come cost sensitive.
  - **RandomForest** / **NaiveBayes** / **Ibk** come classificatori.

Progettazione

- Per valutare l'accuratezza dei vari classificatori è stato sviluppato un software apposito in linguaggio Java.
- Il programma si occupa di:
  - Clonare il progetto da un repository **Git**.
  - Estrarre le release del progetto da **Jira**.
  - Estrarre i commit dal repository.
  - Estrarre da **Jira** i **Ticket** di tipo **Bug Fix** e selezionare solo quelli che hanno almeno un commit associato su **Git**.
  - Calcolare le **AV** per ogni **Ticket** (attraverso le informazioni di **Jira** oppure **Proportion**).
  - Rimuovere l'ultima metà delle release per avere un dataset meno rumoroso.
  - Estrarre le classi **Java** presenti in ogni release del progetto.
  - Calcolare per ogni classe le *metriche* considerate e la *bugginess*.
  - Generare il dataset scrivendo le informazioni ottenute su un file *CSV*.
  - Convertire il dataset in formato *ARFF*.
  - Valutare l'accuratezza dei vari classificatori attraverso il dataset generato utilizzando **Weka**.
  - Salvare i risultati ottenuti su un file *CSV*.

Progettazione – Git & Jira

- Per l'interazione con il progetto *Git* è stata utilizzata la libreria *Jgit*:
  - Il programma clona la repository tramite il comando *clone():*

```
private static void cloneProject(String projName) throws GitAPIException {
    if (!Files.exists(Paths.get(repoDir))) {
        String url = "https://github.com/apache/" + projName.toLowerCase();
        Git git = Git.cloneRepository().setURI(url).setDirectory(new File(repoDir)).call();
        git.close();
    }
}
```

- Per ottenere la lista delle release del progetto viene interrogato *Jira* attraverso le **Rest API**:

```
// Fills the array list with releases dates and orders them
// Ignores releases with missing dates
releases = new ArrayList<>();
Integer i;
String url = "https://issues.apache.org/jira/rest/api/2/project/" + projName;
JSONObject json = Utilities.readJsonFromUrl(url);
JSONArray versions = json.getJSONArray("versions");
releaseNames = new HashMap<>();
releaseID = new HashMap<>();
for (i = 0; i < versions.length(); i++) {
    var name = "";
    var id = "";
    if (versions.getJSONObject(i).has("releaseDate")) {
        if (versions.getJSONObject(i).has("name"))
            name = versions.getJSONObject(i).get("name").toString();
        if (versions.getJSONObject(i).has("id"))
            id = versions.getJSONObject(i).get("id").toString();
        addRelease(versions.getJSONObject(i).get("releaseDate").toString(), name, id);
    }
}
```

  - Vengono scartate le release senza data di pubblicazione.
  - Viene istanziato un oggetto *Release* per ogni release valida trovata.

Progettazione – Commit

- Attraverso il comando *log()* vengono estratti tutti *commit* dalla repository:

```java
public static List<RevCommit> getAllCommits(List<Release> releases, Path repoPath) throws GitAPIException, IOException {
    List<RevCommit> commits = new ArrayList<>();
    try (Git git = Git.open(repoPath.toFile())){
        Iterable<RevCommit> logs = git.log().all().call();
        for (RevCommit commit : logs) {
            commits.add(commit);
        }
    }
```

- In seguito vengono assegnati alle varie *release* in base alla loro data:

```java
// Add commits to releases
for (RevCommit commit : commits) {
    LocalDateTime commitDate = Instant.ofEpochSecond(commit.getCommitTime()).atZone(ZoneId.of("UTC")).toLocalDateTime();
    LocalDateTime before = LocalDateTime.MIN;
    for (Release release : releases) {
        if (commitDate.isAfter(before) && commitDate.isBefore(release.getReleaseDate()) || commitDate.isEqual(release.getReleaseDate())) {
            release.getCommits().add(commit);
        }
        before = release.getReleaseDate();
    }
}
```

Progettazione – Ticket Jira

- Sempre attraverso le **Rest API**, vengono estratti tutti i ticket *Jira* di tipo «Bug Fixed»:

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

- Per ognuno dei ticket ottenuti, viene istanziato un nuovo oggetto *Ticket*, che mantiene l'ID e la data di creazione del ticket:

```
JSONObject json = Utilities.readJsonFromUrl(url);
JSONArray issues = json.getJSONArray("issues");
total = json.getInt("total");

for (; i < total && i < j; i++) {
    // Iterate through each bug
    String key = issues.getJSONObject(i%1000).get("key").toString();
    LocalDateTime creationDate = LocalDateTime.parse(issues.getJSONObject(i%1000).getJSONObject("fields").getString("created").substring(0,16));

    JSONArray versions = issues.getJSONObject(i % 1000).getJSONObject("fields").getJSONArray("versions");
    List<Integer> listAV = getAV(versions, releases);
    Ticket ticket = new Ticket(key, creationDate, listAV);

    if (!(listAV.isEmpty() || listAV.get(0) == null)) {
        ticket.setIv(listAV.get(0));
    }
    else {
        ticket.setIv(0);
    }

    ticket.setOv(getOVIndex(creationDate, releases));
    tickets.add(ticket);
}
```

- **IV**: calcolata come la più vecchia delle *affected version*.
- **OV**: ottenuta dal campo «created» dell'array *JSON*.

Progettazione – Jira & Git

- Dopo aver ottenuto le release, i ticket e i commit, le informazioni ricavate da *Jira* e da *Git* vengono mappate tra loro.
- Vengono mantenuti soltanto i ticket che hanno almeno un commit di fix associato:
  - Viene verificato se l'ID del ticket è contenuto nel messaggio di almeno un commit.

```
for (RevCommit commit : commits) {
    String message = commit.getFullMessage();
    if (message.contains(ticketID +",") || message.contains(ticketID + " ") || message.contains(ticketID + ":")
            || message.contains(ticketID + ".") || message.contains(ticketID + "\r") || message.contains(ticketID + "_")
            || message.contains(ticketID + "-") || message.contains(ticketID + "\n") || message.contains(ticketID + ")")
            || message.contains(ticketID + "/") || message.endsWith(ticketID) || message.contains(ticketID + "]")) {

        LocalDateTime commitDate = commit.getAuthorIdent().getWhen().toInstant().atZone(ZoneId.systemDefault()).toLocalDateTime();
        commitDates.add(commitDate);
        ticket.getCommits().add(commit);
    }
}
```

- Viene calcolata la **FV** di ogni ticket in base alla data di risoluzione ottenuta dai commit associati al ticket stesso:

```
if (!commitDates.isEmpty()) {
    Collections.sort(commitDates);
    LocalDateTime resolutionDate = commitDates.get(commitDates.size()-1);
    ticket.setResolutionDate(resolutionDate);
    ticket.setFv(compareDateRelease(resolutionDate, releases));
}
```

Progettazione – Proportion

- Per alcuni ticket non è presente una **IV**, di conseguenza non è possibile calcolare la lista delle **AV**.
- Attraverso il **Proportion** è possibile stimare la *Injected Version* di un ticket:
  - Per ogni ticket viene calcolata la **IV** attraverso la seguente formula:
    - $IV = FV - (FV - OV) * P$
  - È stato utilizzato un approccio «Moving Window» per il **Proportion**, che consiste nel calcolare **P** in base all'ultimo 1% di ticket:

```
public static void proportion(List<Ticket> tickets) {
    List<Ticket> checkedTickets = initialCheck(tickets);

    int numTickets = tickets.size();
    percentage = numTickets * 1/100;

    List<Ticket> propList = new ArrayList<>();
    for (Ticket ticket : tickets) {
        if (!checkedTickets.contains(ticket)) {
            if (ticket.getIv() != 0) {
                updatePropWindow(propList, ticket);
            }
            else {
                setIV(propList, ticket);
            }
        }
    }
}
```

Progettazione – Affected Versions

- Una volta calcolata l'**IV** e la **FV**, è possibile determinare le **AV** di un bug:
  - In questa fase vengono scartati i ticket che contengono informazioni incorrette relativamente a **IV**, **OV** e **FV**:
    - **IV > OV** o **IV > FV**: informazione errata in quanto il bug sarebbe stato introdotto dopo essere stato scoperto/corretto.
    - **IV = FV**: non esistono **AV** per il ticket poiché il bug è stato introdotto e risolto nella stessa versione.
  - I ticket rimasti a seguito di tale processo, sono quelli in cui:
    - **IV < OV ≤ FV**: è possibile determinare facilmente la lista delle *Affected Version* del ticket.
  - Le *Affected Versions* sono quelle comprese tra *Injected Version* inclusa e *Fixed Version* esclusa:

```
// Fill AV list of a ticket with all versions between IV and FV
private static void setAV(Ticket ticket) {
    ticket.getAv().clear();
    for (int i = ticket.getIv(); i < ticket.getFv(); i++) {
        ticket.getAv().add(i);
    }
}
```

Progettazione – Classi Java

- A questo punto, viene rimossa l'ultima metà di release per ottenere un dataset meno rumoroso:

```java
public static void removeHalfReleases(List<Release> releases, List<Ticket> tickets) {
    int numReleases = releases.size();
    int halfReleases = numReleases / 2;

    Iterator<Release> release = releases.iterator();
    while (release.hasNext()) {
        Release r = release.next();
        if (r.getIndex() > halfReleases) {
            release.remove();
        }
    }

    TicketController.removeTickets(halfReleases, tickets);
}
```

- Per ogni release presente nel progetto vengono estratte le relative classi *Java*:

```java
for (Release release : releases) {
    List<String> fileNames = new ArrayList<>();

    try (TreeWalk treeWalk = new TreeWalk(git.getRepository())) {
        ObjectId treeId = release.getLastCommit().getTree();

        treeWalk.reset(treeId);
        treeWalk.setRecursive(true);

        while (treeWalk.next()) {
            addFile(treeWalk, release, fileNames, fileMap);
        }
    }
}
```

  - Ogni release mantiene l'ultimo commit eseguito per essa.
  - Tramite *Jgit* è possibile ottenere i file presenti nella repository al momento del commit considerato.
  - Per ogni classe *Java* individuata, viene istanziato un oggetto *JavaFile*, che contiene i dati e le metriche della classe stessa.

Progettazione – Metriche

- Il programma procede con il calcolo delle metriche associate alle classi individuate attraverso l'analisi dei commit ottenuti in precedenza.
- Sono state prese in considerazione le seguenti metriche:
  - **loc**: numero di linee di codice.
  - **locTouched**: numero di linee di codice modificate.
  - **locAdded**: numero di linee di codice aggiunte.
  - **maxLocAdded**: numero massimo di linee di codice aggiunte tra le revisioni della release.
  - **avgLocAdded**: numero medio di linee di codice aggiunte tra le revisioni della release.
  - **chgSetSize**: numero di file "committed" insieme alla classe.
  - **maxChgSetSize**: numero massimo di file "committed" insieme alla classe tra le revisioni della release.
  - **avgChgSetSize**: numero medio di file "committed" insieme alla classe tra le revisioni della release.
  - **churn**: numero di linee di codice "added – removed".
  - **maxChurn**: numero massimo di "added – removed" tra le revisioni della release.
  - **avgChurn**: numero medio di "added – removed" tra le revisioni della release.
  - **numRevisions**: numero di revisioni della classe nella release corrente.
  - **numFix**: numero di bug risolti nella classe nella release corrente.
  - **numAuth**: numero di autori che hanno apportato modifiche alla classe nella release corrente.
  - **age**: età della classe in settimane a partire dalla data di creazione nella release corrente.

Progettazione – Bugginess

- Attraverso la classe *DiffFormatter* di *Jgit* è possibile analizzare le modifiche introdotte ad ogni commit:
  - Attraverso il metodo *scan()* si ottiene una lista di *DiffEntry* che rappresentano le differenze tra i commit e i suoi parent.
  - Analizzandoli è possibile calcolare le varie metriche per le classi toccate dal commit.

```
for (RevCommit commit : release.getCommits()) {
    DiffFormatter df = new DiffFormatter(DisabledOutputStream.INSTANCE);
    df.setRepository(repository);
    df.setDiffComparator(RawTextComparator.DEFAULT);
    df.setDetectRenames(true);

    String authName = commit.getAuthorIdent().getName();
    List<DiffEntry> diffs = JavaFileController.getDiffs(commit);
    if (diffs != null) {
        getMetrics(diffs, files, authName, df, commit, releases);
    }
}
```

- Per calcolare la ***bugginess*** di una classe bisogna analizzare dei commit specifici:
  - Se una classe è stata toccata da un commit relativo ad un ticket di tipo «Bug Fixed», vuol dire che la modifica introdotta ha risolto un bug e quindi la classe era precedentemente buggy.
  - È possibile accedere alle *Affected Versions* riportate nel ticket a cui fa riferimento il commit ed impostare la classe come *buggy* in tutte le release considerate.

```
for (Release release : releases) {
    for (JavaFile file : release.getJavaFiles()) {
        if (file.getPath().equals(fileName) || checkMapRename(file.getPath(), fileMap)) {
            setBugginess(file, av, release);
        }
    }
}
```

Progettazione – Weka

- Una volta calcolate la ***bugginess*** e le *metriche* delle varie classi, viene generato il dataset in formato *CSV*.
- Per valutare l'accuratezza dei vari classificatori sono state utilizzate le *API* di **Weka**:
  - Il dataset viene convertito in formato *ARFF.*
  - Le metriche di accuratezza vengono calcolate per ogni *classificatore*, metodo di *feature selection*, metodo di *balancing* e tipologia di *cost sensitivity* utilizzati:
    - Viene eseguito un run per ogni combinazione possibile.
    - Vengono calcolati i valori di *recall*, *precision*, *AUC* e *kappa*.
  - Infine, i risultati delle varie esecuzioni vengono salvati all'interno di un file *CSV*, in modo tale da poter analizzare l'accuratezza dei vari classificatori, anche attraverso dei grafici.

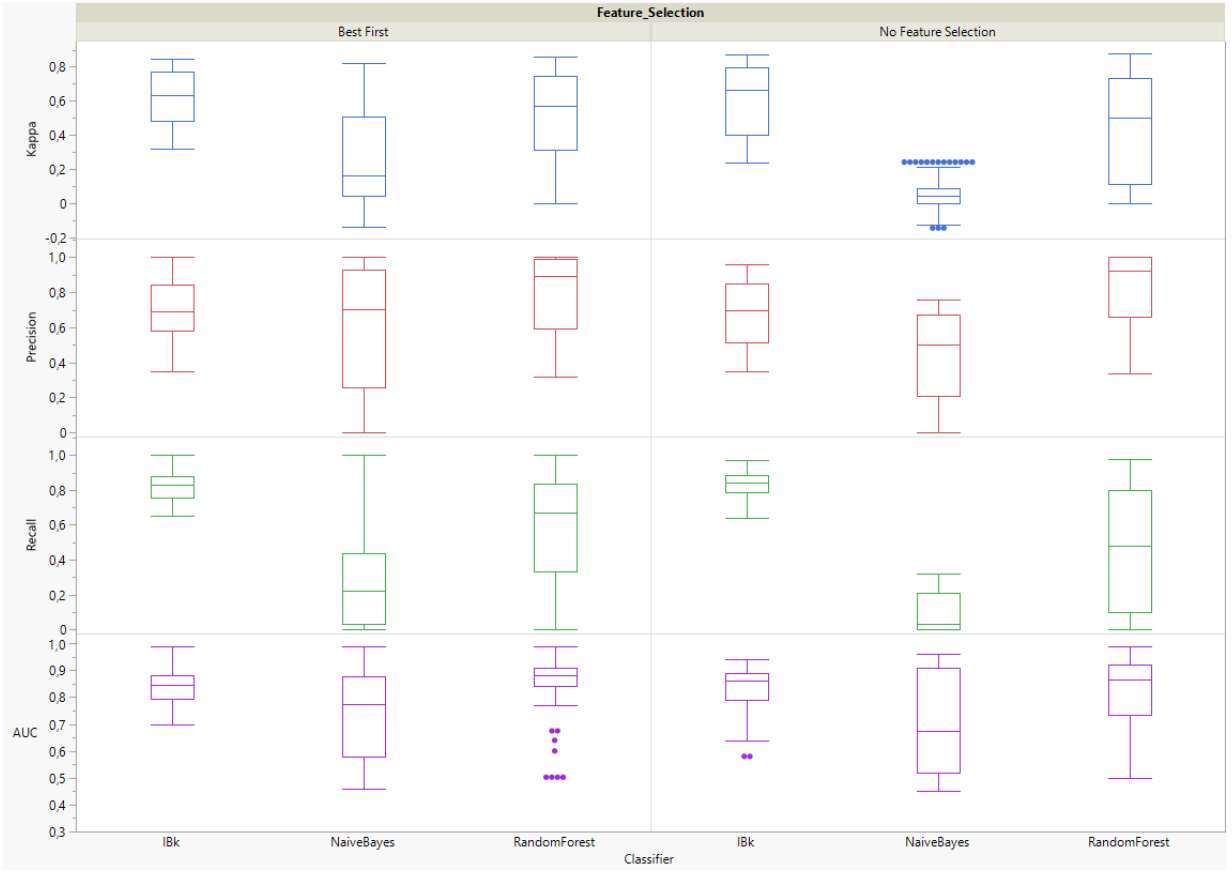
```
for (var a = 2; a <= parts; a++) {

    // The dataset is splitted in training and testing sets
    filter.setInvertSelection(true);
    filter.setSplitPoint(a);
    Instances training = Filter.useFilter(dataset, filter);

    filter.setSplitPoint(a + 1.0);
    Instances tmp = Filter.useFilter(dataset, filter);

    filter.setInvertSelection(false);
    filter.setSplitPoint(a);
    Instances testing = Filter.useFilter(tmp, filter);

    // Perform the evaluation with all the settings
    for (var i = 0; i <= 3; i++) {
        for (var j = 0; j <= 1; j++) {
            for (var k = 0; k <= 2; k++) {
                applyFilters(training, testing, i, j, k, a - 1);
            }
        }
    }
}
```

Risultati

- I due dataset generati a seguito dell'esecuzione del programma, contengono il seguente numero di classi:
  - **Bookkeeper**: 3108 classi / 783 difettose (25.19%)
  - **Tajo**: 5884 classi / 3799 difettose (64.56%)
- I dataset ottenuti sono risultati entrambi abbastanza sbilanciati:
  - Il primo verso i negativi.
  - Il secondo verso i positivi.
- Dai risultati ottenuti sono stati generati vari grafici che sono stati analizzati considerando le tipologie di **Feature Selection**, **Balancing** e **Cost Sensitivity** utilizzate.
- Infine, è stata discussa l'accuratezza generale dei classificatori in base al dataset utilizzato.

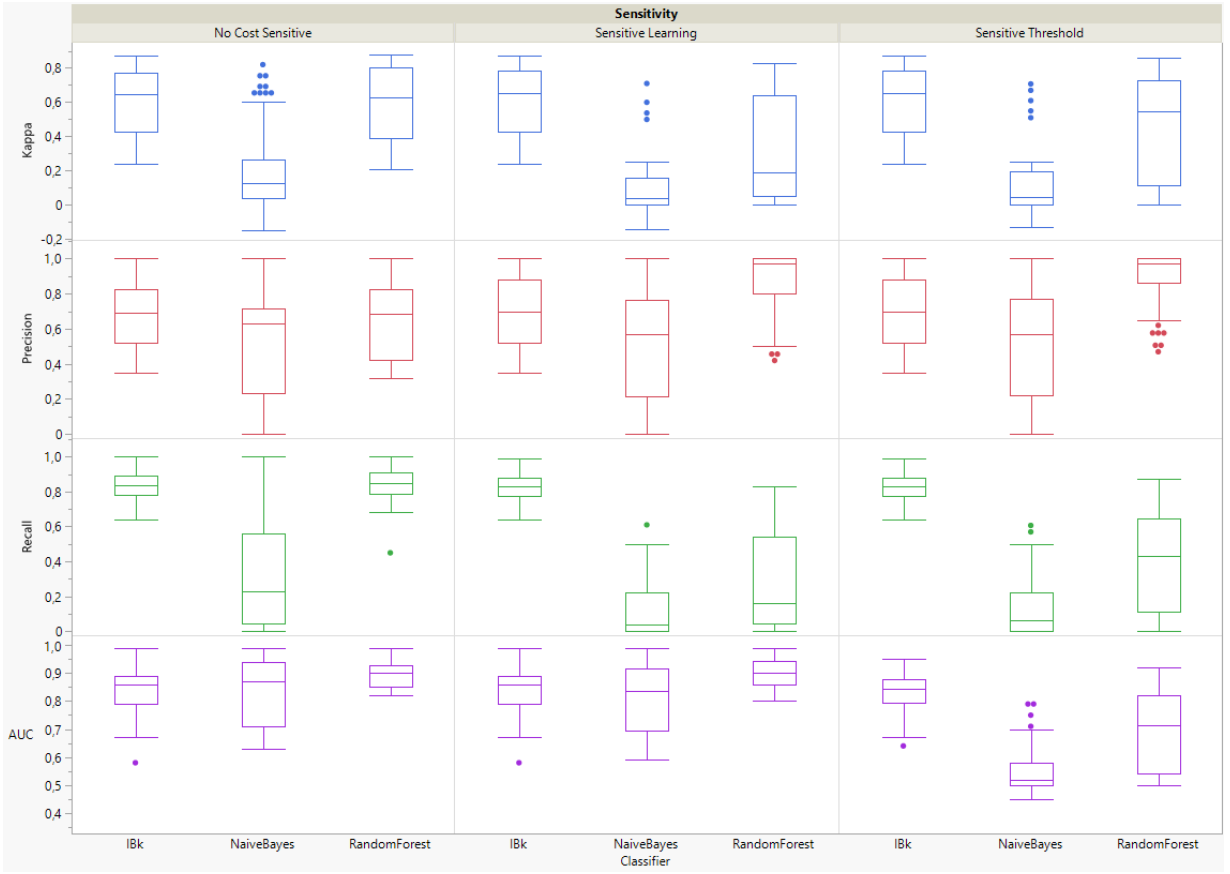
Bookkeeper – Feature Selection

- **Ibk** presenta un'accuratezza molto simile a prescindere dall'utilizzo o meno di *Best First*:
  - **Best First**: kappa **0.62**, precision **0.69**, recall **0.83**, AUC **0.85**
  - **No feature selection**: kappa **0.61** , precision **0.68**, recall **0.84**, AUC **0.83**
- **Naive Bayes** ha raggiunto un'accuratezza maggiore utilizzando *Best First*:
  - **Best First**: kappa **0.24**, precision **0.64**, recall **0.29**, AUC **0.75**
  - **No feature selection**: kappa **0.05** , precision **0.40**, recall **0.10**, AUC **0.70**
- **Random Forest** presenta un'accuratezza molto simile a prescindere dall'utilizzo o meno di *Best First*:
  - **Best First**: kappa **0.50**, precision **0.78**, recall **0.57**, AUC **0.85**
  - **No feature selection**: kappa **0.43** , precision **0.81**, recall **0.46**, AUC **0.82**
- Con il dataset generato, è consigliato utilizzare *Best First* in quanto causa un miglioramento generale dell'accuratezza.



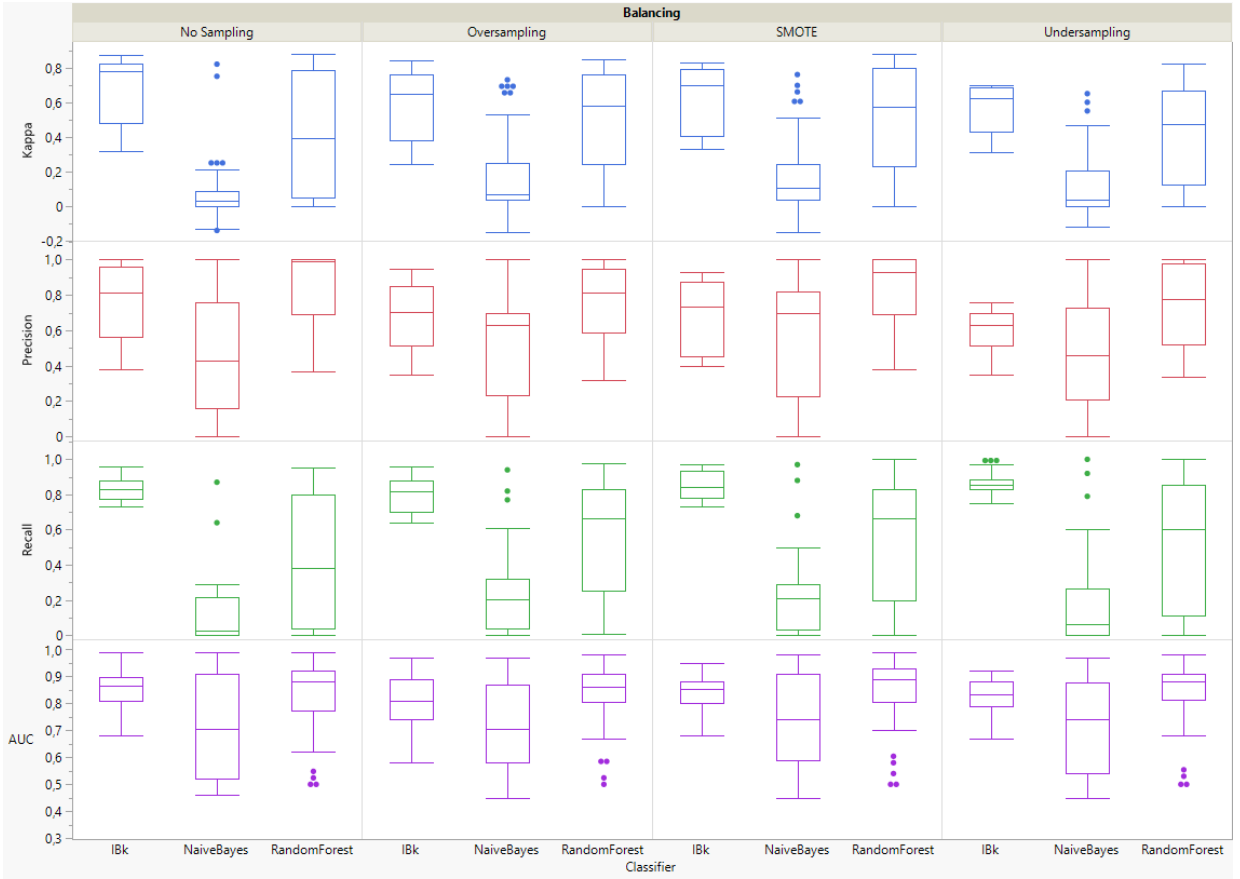
Bookkeeper – Cost Sensitivity

- **Ibk** presenta un'accuratezza molto simile a prescindere dall'utilizzo o meno di *Cost Sensitivity*:
  - kappa **0.61**, precision **0.67**, recall **0.84**, AUC **0.84**
  - Attraverso *Cost Sensitive* aumenta leggermente la *precision* a **0.70** ma diminuisce la *recall* a **0.83**
- **Naive Bayes** ha raggiunto un'accuratezza maggiore senza utilizzare *Cost Sensitivity*:
  - kappa **0.22**, precision **0.51**, recall **0.32**, AUC **0.83**
- **Random Forest** presenta un'accuratezza molto instabile in base all'utilizzo o meno di *Cost Sensitivity*:
  - Si può notare dal grafico come diminuiscano notevolmente *kappa* e *recall* ma aumenta di molto il valore di *precision*.
- Il *Cost Sensitivity* sembra avere poco impatto sul classificatore **Ibk** ma un grande impatto su **Naive Bayes** e **Random Forest**.



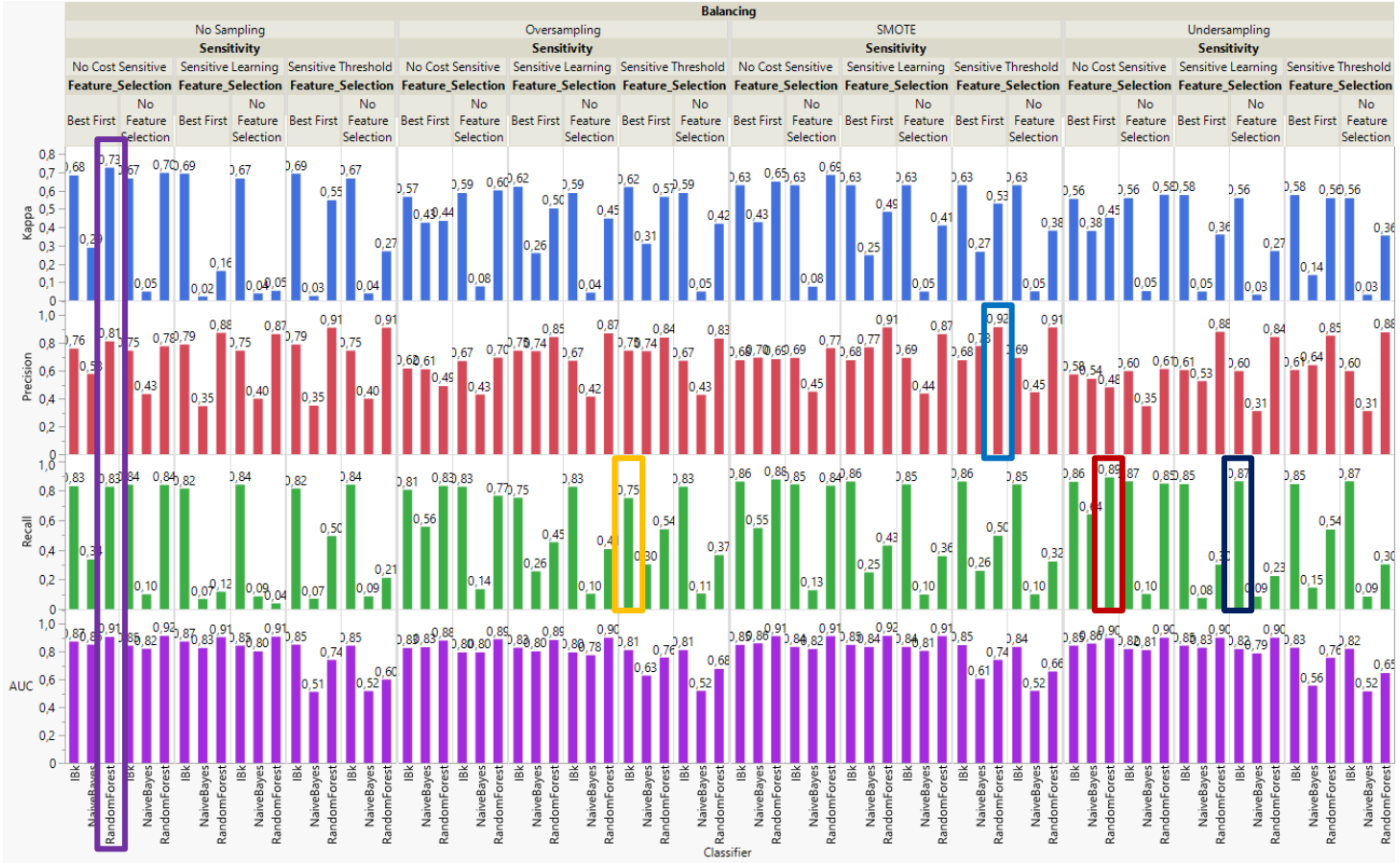
Bookkeeper – Balancing

- **Ibk** presenta un'accuratezza maggiore senza balancing:
  - kappa **0.68**, precision **0.76**, recall **0.83**, AUC **0.86**
  - Sia *SMOTE* che *Undersampling* aumentano la recall: **0.85**
- **Naive Bayes** ha raggiunto un'accuratezza maggiore attraverso *SMOTE*:
  - kappa **0.19**, precision **0.60**, recall **0.23**, AUC **0.83**
  - *Oversampling* migliora leggermente la recall: **0.24**
- **Random Forest** ha raggiunto un'accuratezza maggiore attraverso *SMOTE*:
  - kappa **0.52**, precision **0.84**, recall **0.55**, AUC **0.84**
  - Senza balancing viene migliorata leggermente la precision: **0.86**
- Per il dataset costruito, *Undersampling* risulta essere la scelta peggiore, in quanto non migliora nessuna metrica, tranne *recall*, a prescindere dal classificatore utilizzato.



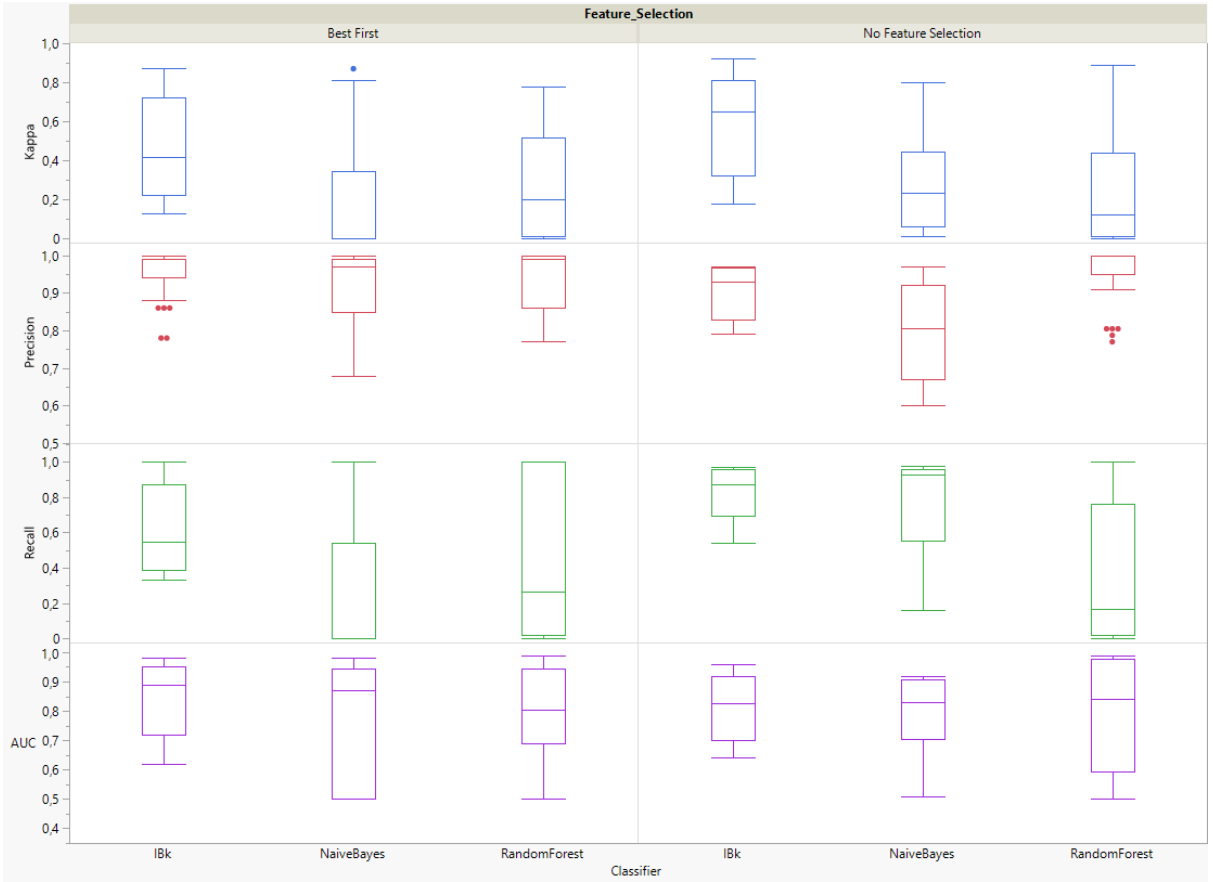
Bookkeeper – Riepilogo

- Se l'obiettivo è quello di massimizzare *precision*, il classificatore migliore è **Random Forest** utilizzando *SMOTE* / *Sensitive Threshold* / *Best First*:
  - Precision **0.92**
  - Tuttavia si ha una recall mediocre: **0.50**
- Se l'obiettivo è quello di massimizzare *recall*, il classificatore migliore è **Random Forest** utilizzando *Undersampling* / *No Sensitive* / *Best First*:
  - Recall **0.89**
  - Si ottengono dei valori discreti per le altre metriche: precision **0.48**, kappa **0.45**, AUC **0.90**
- La *massima accuratezza media* viene raggiunta con il classificatore **Random Forest** utilizzando *No Balancing* / *No Sensitive* / *Best First*:
  - kappa **0.73**, precision **0.81**, recall **0.83**, AUC **0.91**
- A prescindere dalle tecniche utilizzate, **Ibk** offre sempre dei valori molto elevati di *recall*:
  - Min = **0.75**
  - Max = **0.87**



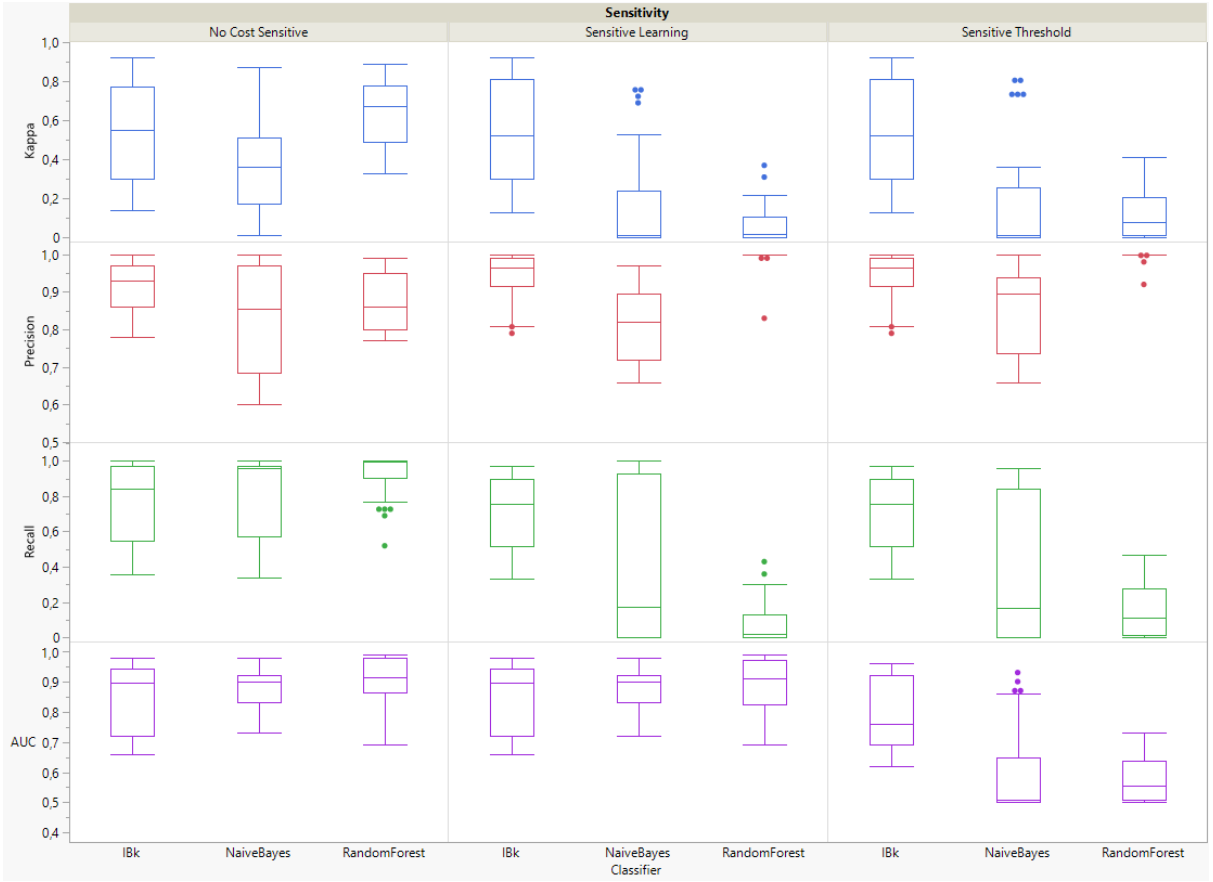
Tajo – Feature Selection

- **Ibk** presenta un'accuratezza maggiore senza feature selection:
  - **Best First**: kappa **0.47**, precision **0.96**, recall **0.61**, AUC **0.84**
  - **No feature selection**: kappa **0.59** , precision **0.90**, recall **0.82**, AUC **0.81**
- **Naive Bayes** presenta due valori di accuratezza completamente diversi in base all'utilizzo o meno di *Best First*:
  - **Best First**: kappa **0.18**, precision **0.93**, recall **0.27**, AUC **0.78**
  - **No feature selection**: kappa **0.28** , precision **0.80**, recall **0.74**, AUC **0.79**
- **Random Forest** presenta un'accuratezza molto simile a prescindere dall'utilizzo o meno di *Best First*:
  - **Best First**: kappa **0.29**, precision **0.93**, recall **0.42**, AUC **0.79**
  - **No feature selection**: kappa **0.26** , precision **0.96**, recall **0.36**, AUC **0.79**
- Possiamo vedere che l'utilizzo di *Best First* comporta in generale una diminuzione della *precision* e un aumento della *recall*, tranne per il classificatore **Random Forest** in cui si verifica il contrario.



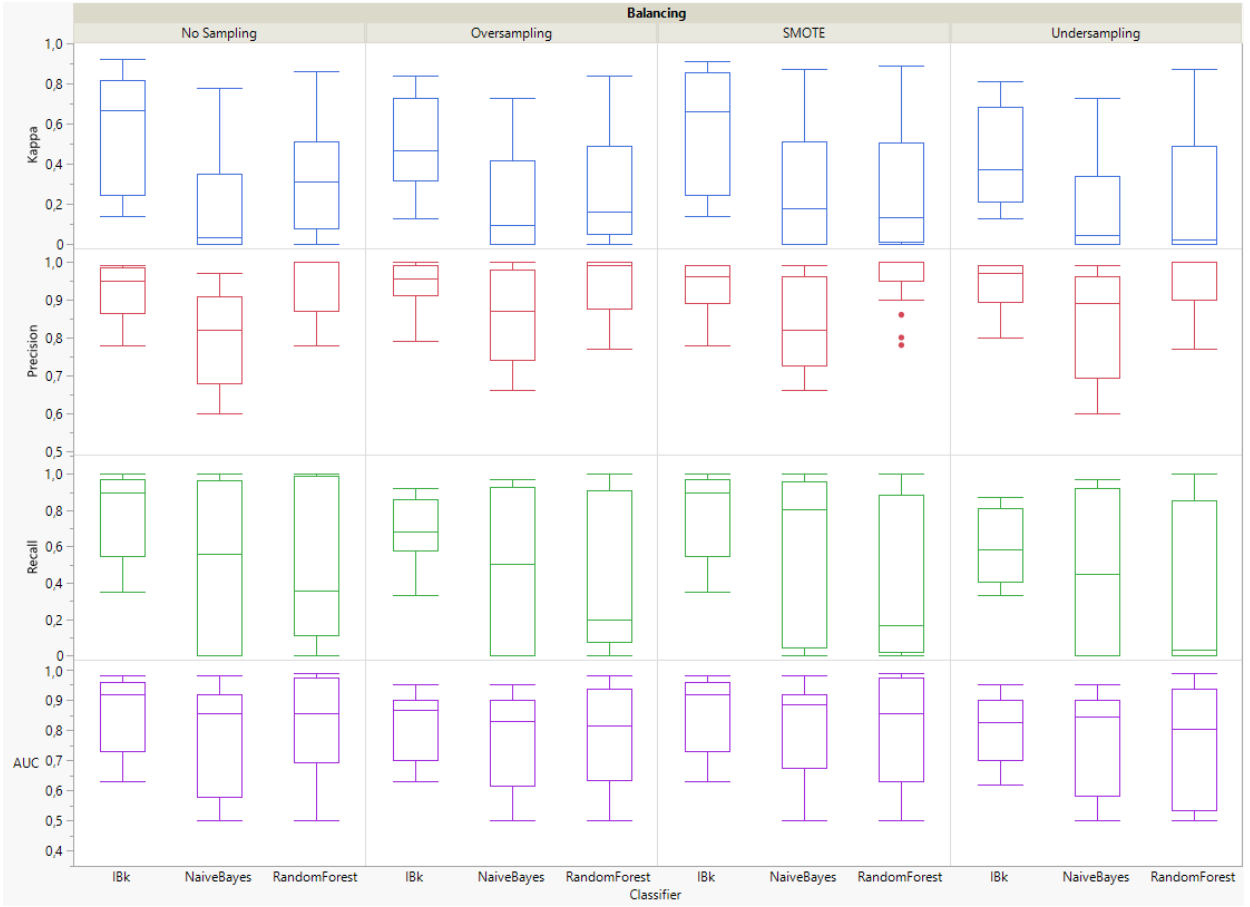
Tajo – Cost Sensitivity

- **Ibk** presenta un'accuratezza molto simile a prescindere dall'utilizzo o meno di *Cost Sensitivity*:
  - kappa **0.53**, precision **0.91**, recall **0.75**, AUC **0.84**
  - Attraverso *Cost Sensitive* aumenta leggermente la *precision* a **0.84** ma diminuisce la *recall* a **0.70**
- **Naive Bayes** ha raggiunto un'accuratezza maggiore senza utilizzare *Cost Sensitivity*:
  - kappa **0.36**, precision **0.84**, recall **0.80**, AUC **0.88**
  - Attraverso *Sensitive Threshold* migliora leggermente la precision: **0.85**
- **Random Forest** presenta un'accuratezza molto instabile in base all'utilizzo o meno di *Cost Sensitivity*:
  - Si può notare dal grafico come diminuisca notevolmente *kappa* e *recall* ma aumenta di molto il valore di *precision*.
- Il *Cost Sensitivity* sembra avere poco impatto sul classificatore **Ibk** ma un grande impatto su **Naive Bayes** e **Random Forest**.



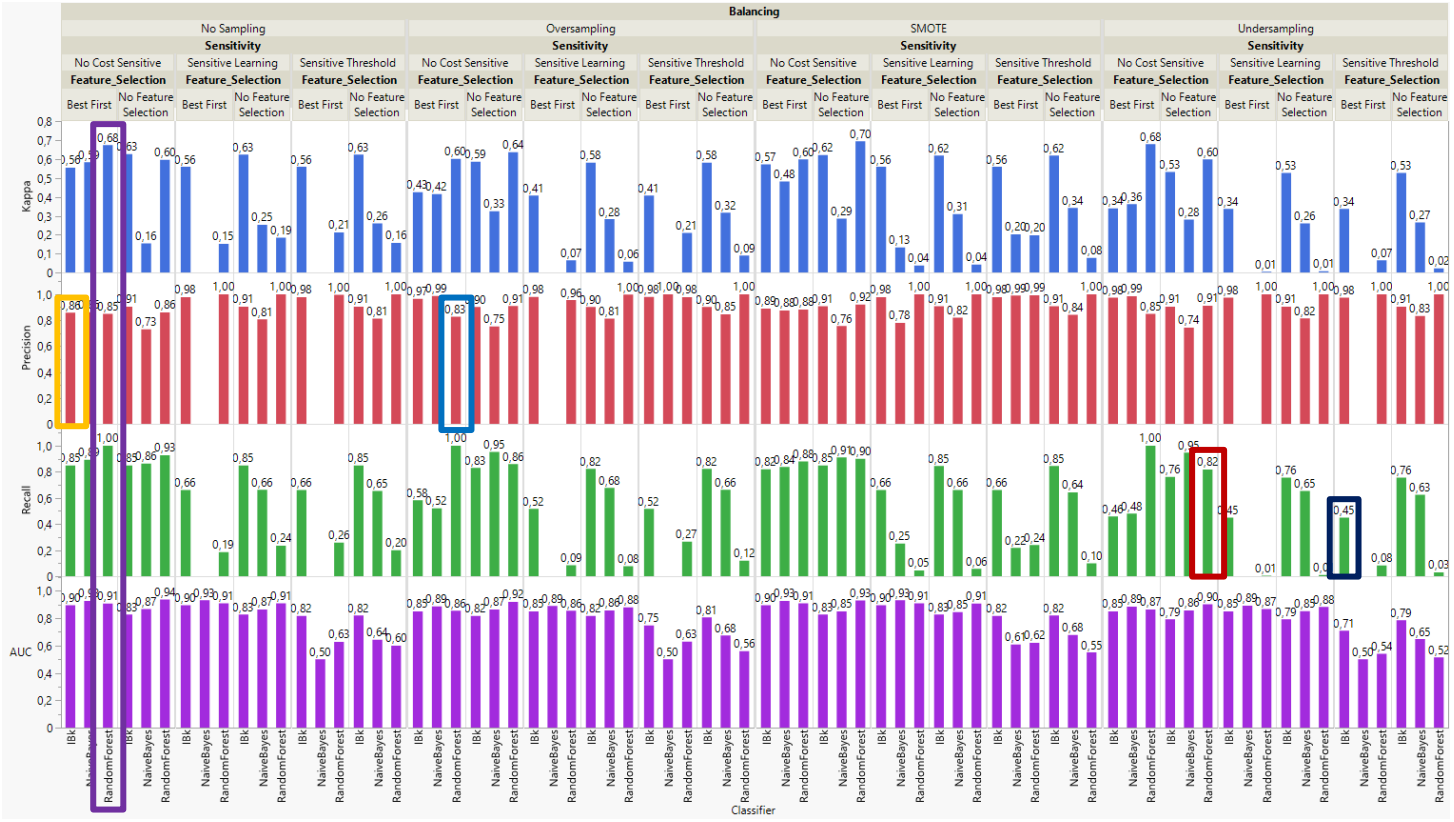
Tajo – Balancing

- **Ibk** presenta un'accuratezza maggiore attraverso *SMOTE*:
  - kappa **0.59**, precision **0.93**, recall **0.78**, AUC **0.85**
  - Sia *Oversampling* che *Undersampling* migliorano la precision: **0.94**
- **Naive Bayes** ha raggiunto un'accuratezza maggiore attraverso *SMOTE*:
  - kappa **0.29**, precision **0.83**, recall **0.59**, AUC **0.81**
  - *Oversampling* migliora la precision: **0.86**
  - *Undersampling* migliora leggermente la precision: **0.84**
- **Random Forest** ha raggiunto un'accuratezza maggiore senza balancing:
  - kappa **0.33**, precision **0.94**, recall **0.47**, AUC **0.81**
  - Sia *Undersampling* che *SMOTE* migliorano la precision: **0.96**
- Per il dataset costruito, *Undersampling* risulta essere la scelta peggiore, in quanto non migliora nessuna metrica, tranne *precision*, a prescindere dal classificatore utilizzato.



Tajo – Riepilogo

- Se l'obiettivo è quello di massimizzare *precision*, il classificatore migliore è **Random Forest**:
  - Min = **0.83**
  - Max = **1.00**
  - Tuttavia, nella maggioranza dei casi, si ha una recall bassa.
- Se l'obiettivo è quello di massimizzare *recall*, il classificatore migliore è **Random Forest** con *No Sensitive*:
  - Min = **0.82**
  - Max = **1.00**
  - Si ottengono dei valori buoni anche per le altre metriche.
- La *massima accuratezza media* viene raggiunta con il classificatore **Random Forest** utilizzando *No Balancing* / *No Sensitive* / *Best First*:
  - kappa **0.68**, precision **0.85**, recall **1.00**, AUC **0.91**
- A prescindere dalle tecniche utilizzate, **Ibk** offre sempre dei valori buoni di *precision* e *recall:*
  - Min precision = **0.86**
  - Min recall = **0.45**



Conclusioni

- Per entrambi i dataset sono stati riscontrati valori sempre positivi di *kappa*, tranne per il classificatore **Naive Bayes** con il dataset **Tajo**:
  - In generale, tutti i classificatori sono più accurati rispetto ad un classificatore random.
- Per il primo dataset, *Best First* tende a migliorare l'accuratezza, per il secondo tende a peggiorarla:
  - Il dataset **Bookkeeper** contiene diverse feature con una bassa correlazione con la variabile di interesse.
  - Il dataset **Tajo** contiene feature con una buona correlazione con la variabile di interesse.
- Utilizzando *Undersampling* otteniamo risultati diversi per i due dataset:
  - Per **Bookkeeper** vengono ridotte le istanze negative, di conseguenza vengono classificati più positivi e meno negativi: aumenta *recall* ma diminuisce *precision*.
  - Per **Tajo** vengono ridotte le istanze positive, di conseguenza vengono classificati più negativi e meno positivi: aumenta *precision* ma diminuisce *recall*.
- I classificatori più accurati sono stati **Ibk** e **Random Forest** per entrambi i dataset:
  - Permettono di massimizzare le varie metriche, al contrario di **Naive Bayes**.
- Lo studio eseguito sui due dataset mostra come i risultati ottenuti su **Bookkeeper** sono migliori rispetto a quelli ottenuti su **Tajo**:
  - Le metriche di adeguatezza dipendono dal dataset considerato.

Riferimenti

- GitHub: https://github.com/gabrielequatrana/Deliverable2
- Travis CI: https://app.travis-ci.com/gabrielequatrana/Deliverable2
- SonarCloud: https://sonarcloud.io/dashboard?id=gabrielequatrana_Deliverable2