

Rapport du TER GMIN401 : Intégration et optimisation d'algorithmes de classifications supervisées pour Weka

Par : ALIJATE Mehdi - NEGROS Hadrien - TURKI Batoul

31 Janvier 2014

Table des matières

1	Introduction	2
2	Exploration de WEKA	2
2.1	L'API Weka/Sources avec Eclipse	2
2.2	L'utilisation des classes	2
2.3	Ajout d'un algorithme dans Weka	2
3	De nouvelles méthodes de classification	3
3.1	Pondérations intra-classe	3
3.2	Pondérations inter-classe	3
3.3	Algorithmes de classifications	4
4	Développement des différentes classes	4
4.1	Méthodologie	4
4.2	Extension de Naive Bayes Multinomial	5
4.3	Class-Feature-Centroid	5
5	Intégration et tests	5
5.1	Intégration dans l'écosystème de Weka	5
5.2	Tests	6
5.3	Résultats	6
6	Discussion et Conclusion	7
7	Sources	8

Résumé

Ce sujet vise à intégrer et à optimiser des algorithmes de classifications supervisées de documents dans la suite logiciel WEKA. Ces algorithmes sont issus de travaux de recherche menés récemment au sein du LIRMM.

<https://github.com/sorgenh/TERWeka/>

1 Introduction

La classification de documents est le mécanisme consistant à classer automatiquement des ressources la classe prédéfinie lui correspondant le mieux.

Plusieurs formes de classification existent (par genre, par opinion, par thème...etc), et se font via des algorithmes de classifications spécifiques. Ceux-ci se basent sur des méthodes principalement numériques (probabilistes), avec des algorithmes utilisant les mathématiques ou basés sur la recherche d'information.

Ce TER vise justement à intégrer des algorithmes de classifications supervisées de documents dans la suite logiciel WEKA¹, se basant sur un nouveau modèle de classification à partir d'un faible nombre de document, intégrant de nouvelles pondérations adaptées.

Tout d'abord, il faudra explorer l'API de WEKA, pour prendre en main du code source, la maniabilité des classes et explorer une méthode d'ajout d'un algorithme de classification. Ensuite, nous nous pencherons sur le développement des différentes classes en établissant une méthodologie concrétisant le travail mené au laboratoire du LIRMM, s'en suivra une phase d'intégration et de tests.

2 Exploration de WEKA

Après la réunion du 24/01/14, nous avons établi un plan de travail pour bien mener et répartir les tâches de ce TER. Il a été décidé de le diviser en trois grandes parties. La première, qui est décrite ci-dessous consiste à explorer et prendre en main l'API de WEKA, afin de pouvoir y rajouter les algorithmes que l'on aura développé lors de la deuxième partie, et qui seront testés et intégrés lors de la troisième.

2.1 L'API Weka/Sources avec Eclipse

Pour explorer l'API, nous nous sommes aidés de l'IDE Eclipse, qui permet facilement parcourir les sources d'une librairie externe. Après avoir étudié l'arborescence des classes de l'API, nous avons pu cibler les différentes classes et méthodes qui nous intéressent, et étudié leurs fonctionnement. Nous nous sommes aidé de ce wiki².

2.2 L'utilisation des classes

Une fois familiarisés avec l'API Weka, on a creusé un peu plus du côté des classes qui pourraient nous être utiles pour ce TER. Il s'agit des certaines classes présentes dans le package "weka.classifiers". En effet, notre but étant d'intégrer des algorithmes de classification, il est utile de savoir comment tournent les algorithmes de classifications, leur paramétrage et l'architecture pour organiser les ressources pour ces derniers.

Quelques tests ont été menés notamment pour bayes naïf multinomial, que nous avons fait tourné sur différentes données, et avec différentes options.

2.3 Ajout d'un algorithme dans Weka

Après avoir étudié en détail la classe *NaiveBayesMultinomial*, nous avons remarqué que le calcul des pondérations (dans l'implémentation de Weka, seul la mesure intra-classe Tf est utilisée) se fait dans la méthode **buildClassifier**. Nous allons donc créer une sous classe de *NaiveBayesMultinomial*, contenant une méthode surchargeant **buildClassifier** dans laquelle nous calculerons toutes les pondérations supplémentaires.

Une fois tout cela creusé et vu en détails, il faudra intégrer l'algorithme dans l'écosystème de Weka, c'est à dire, pour le rendre disponible dans l'Explorateur, expérimentateur, etc . Weka prend en charge les classes dérivées dans le package, ceci est géré par le *GenericPropertiesCreator*. Il faudra donc dire à Weka où trouver notre nouveau classificateur et il s'occupera de l'afficher dans la *GenericObjectEditor*.

1. Weka est une suite populaire de logiciels d'apprentissage automatique. Écrite en Java, développée à l'université de Waikato, Nouvelle-Zélande. Weka est un Logiciel libre disponible sous la Licence publique générale GNU.

2. <http://weka.wikispaces.com/>

Nous y reviendrons plus en détails lors de la troisième étape de notre TER : L'intégration des algorithmes dans WEKA.

3 De nouvelles méthodes de classification

Dans cette partie, nous allons vous présenter les différentes pondérations que nous allons utiliser pour construire nos classifieurs. Nous allons d'abord définir les mesures intra-classe inspirées du TF-IDF, puis les mesures inter-classe développées au *LIRMM*. Ces mesures vont nous permettre de définir si un terme (un élément d'un document) est plus ou moins représentatif de la classe.

Toutes ces mesures ont été définies dans l'article *De nouvelles pondérations adaptées à la classification de petits volumes de données textuelles*. [1].

3.1 Pondérations intra-classe

Les pondérations que nous définissons ci-dessous sont dites **intra-classe** car les différentes valeurs que nous utilisons pour les calculer sont dépendante d'une classe.

intra-classe document

Cette mesure dépend du nombre de documents contenant le terme dans la classe.

$$inner-weight_{ij}^{Df} = \frac{DF_{ti}^j}{|d_j|}$$

Avec :

- DF_{ti}^j : Nombre de documents contenant le terme t_i dans la classe C_j
- $|d_j|$: Nombre de documents dans C_j

intra-classe terme

Cette mesure dépend du nombre d'occurrences du terme dans la classe.

$$inner-weight_{ij}^{Tf} = \frac{TF_{ti}^j}{|n_j|}$$

Avec :

- TF_{ti}^j : Nombre d'occurrences du terme t_i dans la classe C_j
- $|n_j|$: Nombre de termes total dans la classe C_j

3.2 Pondérations inter-classe

Les pondérations inter-classes en revanche utilisent des valeurs calculées à partir de l'ensemble du corpus (depuis les classes extérieures à celle qui nous intéresse).

inter-classe terme

Cette mesure dépend du nombre de classes contenant le terme.

$$inter-weight_{ij}^{class} = \log_2 \frac{|C|}{C_{ti}}$$

Avec :

- $|C|$: Nombre de classes
- C_{ti} : Nombre de classes contenant le terme t_i

inter-classe document

Cette mesure dépend du nombre de documents extérieurs à la classe contenant le terme.

$$inter-weight_{ij}^{doc} = \log_2 \frac{|d \notin C_j| + 1}{|d : t_i \notin C_j| + 1} = \log_2 \frac{|d| - |d \in C_j| + 1}{|d : t_i| - |d : t_i \in C_j| + 1}$$

Avec :

- $|d \notin C_j|$: Nombre de documents n'appartenant pas à la classe C_j
- $|d : t_i \notin C_j|$: Nombre de documents n'appartenant pas à la classe C_j qui contient t_i
- $|d|$: Nombre de documents dans l'ensemble des classes
- $|d \in C_j|$: Nombre de documents de la classe C_j
- $|d : t_i|$: Nombre de documents dans l'ensemble des classes contenant le terme t_i
- $|d : t_i \in C_j|$: Nombre de documents de la classe C_j qui contient t_i
- En ajoutant 1, permet de prévenir le cas où t_i est uniquement utilisé dans C_j (quand $|d : t_i \notin C_j| = |d : t_i| - |d : t_i \in C_j| = 0$)

3.3 Algorithmes de classifications

Un algorithme de classification permet de calculer la probabilité de l'appartenance d'un document aux différentes classes du corpus, et donc de l'affecter à la plus probable. Nous allons implémenter un classifieur *Naive Bayes* et *Class-Feature-Centroid*[1] en utilisant les mesures définies plus haut. Pour calculer la probabilité w_{ij} d'un terme i dans une classe j , nous allons combiner les différentes pondérations de 4 façons :

$$\begin{aligned} - w_{ij}^{Tf-Class} &= inner-weight_{ij}^{Tf} \times inter-weight_{ij}^{class} \\ - w_{ij}^{Df-Class} &= inner-weight_{ij}^{Df} \times inter-weight_{ij}^{class} \\ - w_{ij}^{Tf-Doc} &= inner-weight_{ij}^{Tf} \times inter-weight_{ij}^{doc} \\ - w_{ij}^{Df-Doc} &= inner-weight_{ij}^{Df} \times inter-weight_{ij}^{doc} \end{aligned}$$

Nous allons aussi mettre en place une combinaison de ces mesures dépendante de deux paramètres $\alpha, \beta \in [0, 1]$:

$$w_{ij}^{\alpha\beta} = (\alpha \times innerweight_{ij}^{Tf} + (1 - \alpha) \times innerweight_{ij}^{Df}) \times (\beta \times interweight_{ij}^{class} + (1 - \beta) \times interweight_{ij}^{doc})$$

On remarque qu'en faisant varier α et β , on peut retrouver les quatre premières formules.

4 Développement des différentes classes

4.1 Méthodologie

Après avoir étudié l'API Weka, nous avons pu identifier les différentes méthodes de la super-classe *Classifier* que nous allons devoir implémenter :

buildClassifier() C'est la méthode dans laquelle est calculé le tableau des w_{ij} (La probabilité d'un mot par rapport à une classe).

distributionForInstance(Instance) Renvoie les probabilités du document (Instance) en entrée pour chacune des classes du corpus.

Pour notre implémentation, nous allons nous baser sur la classe *NaiveBayesMultinomial*³

3. <http://weka.sourceforge.net/doc.dev/weka/classifiers/bayes/NaiveBayesMultinomial.html>

4.2 Extension de Naive Bayes Multinomial

Nous avons donc créé une nouvelle classe, *NaiveBayesMultinomialTER* dans laquelle sont implémentées les pondérations $inner-weight_{ij}^{Tf}$, $inner-weight_{ij}^{Df}$, $inter-weight_{ij}^{class}$ et $inter-weight_{ij}^{doc}$. Celles ci sont calculées dans la méthode *buildClassifier* que nous avons donc surchargé. Nous avons ensuite créé une seconde classe *NaiveBayesMultinomialTERab* sur le même principe, mais implémentant cette fois $w_{ij}^{\alpha\beta}$

4.3 Class-Feature-Centroid

La classe *CFCTERab* que nous avons créé utilise les même pondérations que *NaiveBayesMultinomialTERab*, nous en avons donc fait une sous classe de celle ci afin de réutiliser la méthode **buildClassifier()** déjà implémentée. La différence est ici dans le calcul de la probabilité qu'un document appartienne à une classe, et donc dans la méthode **distributionForInstance(Instance)**. C'est celle ci que nous avons surchargé.

- Pour calculer la probabilité qu'un document appartienne à une classe, on procède de la façon suivante :
- On représente les classes comme des vecteurs ayant pour taille le nombre d'attributs différents dans le corpus, et pour valeurs la probabilité du terme dans la classe. (exemple : $\vec{C}_j = (0.1, 0.3, 0.2, 0)$)
 - On représente les documents comme des vecteurs ayant aussi pour taille le nombre d'attributs, et pour valeurs la probabilité du terme dans la classe **ou 0 si le terme n'apparaît pas dans le document**. (exemple : $\vec{d} = (0.1, 0, 0.2, 0)$, le terme 2 n'apparaît pas dans le document)
 - On mesure ensuite la proximité entre ces vecteurs. La métrique que nous avons choisi est la *Similarité Cosinus* définie ci dessous.

$$simcos(\vec{u}, \vec{v}) = arccos\left(\frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\| \cdot \|\vec{v}\|}\right)$$

Cette similarité normalisée nous donne la probabilité du document dans la classe.

Le problème est que un document peut ne contenir aucun terme, si par exemple il y a eu une sélection d'attribut en amont (c'était le cas sur nos fichiers de test). Cela résulte en un vecteur nul, donc de norme nulle, et d'une division par zéro lors du calcul de la similarité. La solution trouvée est de rajouter une dimension de valeur 1 aux vecteurs.

5 Intégration et tests

5.1 Intégration dans l'écosystème de Weka

Une fois nos classes, et donc nos trois algorithmes implémenter en langage Java (Cf. Chapitre 4) :

- **NaiveBayesMultinomialTER.java** : construit le modèle de classification avec les quatre pondérations définies (le choix de la pondération se fait via les options).
- **NaiveBayesMultinomialTERab.java** : construit le modèle de classification avec les quatre pondérations définies en variant les valeurs de $a : \alpha$ et $b : \beta$ (le choix des valeurs de α et β se fait via les options).
- **CFCTERab.java** : construit le modèle de classification Class-Feature-Centroïde en variant les valeurs de $a : \alpha$ et $b : \beta$ (le choix des valeurs de α et β se fait via les options).

Une fois nos trois .java prêts, l'intégration dans Weka est prise en charge via *GenericPropertiesCreator*. C'est là où on peut dire à Weka où trouver nos nouveaux classifieurs et il s'occupera de les afficher dans *GenericObjectEditor*, et donc dans l'interface graphique. La procédure détaillée est :

Prérequis :

- ANT : logiciel créé par la fondation Apache qui vise à automatiser l'opération de construction d'un JAR. ⁴
- JDK
- Weka (version 3.6.10 dans notre cas)

4. <http://ant.apache.org/>

Préparation de Weka

À l'aide d'un gestionnaire d'archives, il faudra désarchiver `weka-src.jar`, qui se trouve dans le répertoire de Weka une fois ce dernier installé. Ceci vous donne accès aux différents répertoires et sources du logiciel.

NB : Pour éviter toutes confusions ou conflits, il est préférable de créer un dossier *temp* par exemple, et d'y mettre votre répertoire *weka-src*.

Ajout des nouveaux classifieurs dans Weka

À ce stade du processus d'ajout des algorithmes dans Weka, il faut modifier le fichier `GenericObjectEditor.props` (non le `.java`, se trouvant au même répertoire), qui se trouve dans

`/temp/weka/weka-src/src/java/weka/gui/`, en y ajoutant les trois lignes suivantes :

```
weka.classifier.bayes.CFCTERab,\nweka.classifier.bayes.NaiveBayesMultinomialTER,\nweka.classifier.bayes.NaiveBayesMultinomialTERab,\
```

et ce, en respectant l'ordre alphabétique défini dans le fichier, et surtout au bon endroit (dans la liste après le marquage suivant : `# Lists the Classifiers I want to choose from`).

Ensuite, il suffit de placer les trois fichiers `.java` développés dans le répertoire :

`/temp/weka/weka-src/src/java/weka/classifiers/bayes/`.

Reconstruction de Weka

Comme son nom l'indique bien, cette dernière étape permet de reconstruire Weka avec ses nouvelles propriétés. Il suffit, après avoir installé ANT (voir prérequis ci-dessus), de se placer dans le répertoire

`/temp/weka/weka-src/`, à l'aide d'un terminal, et lancer la commande suivante :

```
ant exejar
```

Celle-ci fera appel automatiquement au fichier `build.xml`, qui, comme expliqué dans la sous-section 2.3, construira de nouveau Weka, et donnera en sortie dans le répertoire `/temp/weka/weka-src/dist/` un nouvel exécutable `weka.jar`, contenant les nouveaux algorithmes de classifications.

5.2 Tests

Afin de tester l'exactitude des résultats de nos nouveaux algorithmes de classifications, nous avons créé un petit fichier de données aléatoire, puis nous avons calculé à la main les pondérations et comparé avec nos résultats. Pour tester la performance de nos algorithmes, nous avons utilisé des fichiers de données créés

lors d'un précédent projet. Ceux-ci contiennent un grand nombre de données textuelles (résumés de films) divisés en plusieurs classes :

- `test3classes.arff` : **150** instances et **41** attributs (une sélection d'attributs a été faite dessus), avec **3** classes : Policier, Fantastique, Comédie.
- `test5classes.arff` : **248** instances et **5082** attributs au complet (sans sélection d'attributs), avec **5** classes : Thriller, Western, Guerre, Policier, Sciences.

Le but étant de classer les films selon leur catégorie cinématographique. Nos deux fichiers `.arff` ont subi de multiples tests avec nos trois algorithmes, avec différentes valeurs pour nos variables α et β , que ce soit pour le `NaiveBayesMultinomialTERab` ou le `CFCTERab`.

Les résultats des tests sont donnés dans la section ci-dessous.

5.3 Résultats

Les deux tableaux suivants montrent les résultats relevés suite aux expérimentations effectuées avec nos trois algorithmes sur nos deux fichiers de tests.

- Le tableau 1 présente les résultats de classifications correctes atteintes avec distinctement les quatre pondérations $Nb^{Tf-Class}$, $Nb^{Df-Class}$, Nb^{Tf-Doc} et Nb^{Df-Doc} de notre algorithme `NaiveBayesMultinomialTER`. Nous constatons que l'utilisation de la pondération $W^{Df-Class}$, pour nos deux jeux de données, donne des résultats supérieurs à ceux d'une classification avec `NaiveBayesMultinomial`.

NBMultinomialTER/fichierTest	$Nb^{Tf-Class}$	$Nb^{Df-Class}$	Nb^{Tf-Doc}	Nb^{Df-Doc}	NBMultinomial
test3classes.arff	66%	67%	64%	66%	66%
test5classes.arff	52%	68%	51%	50%	63%

TABLE 1 – Expérimentations avec les quatre pondérations et comparaison avec NBMultinomial

Algo/FichierTest	α	β	NBMTER $\alpha\beta$	CFCTER $\alpha\beta$	NBMultinomial
test3classes.arff	0.0	1.0	67%	68%	66%
	0.6	0.6	66%	74%	
	0.7	0.3	66%	73%	
test5classes.arff	0.0	1.0	67%	68%	63%
	0.6	0.6	65%	70%	
	0.7	0.3	58%	60%	

TABLE 2 – Expérimentations avec différentes valeurs de α et β pour NBTER $\alpha\beta$ et CFCTER $\alpha\beta$

• Le tableau 2 présente les résultats de classifications correctes atteintes avec nos deux algorithmes NaiveBayesMultinomialTERab et CFCTERab, en variant les valeurs de α et β . Nous constatons que dans tous les cas où $\alpha \leq \beta$, on est toujours supérieurs à NaiveBayesMultinomial. Parcontre, seul notre corpus réduit test3classes.arff donne des résultats \geq à ceux de NaiveBayesMultinomial quand $\alpha > \beta$. Nous constatons aussi que CFCTER $\alpha\beta$ présente une meilleure classification que NaiveBayesMultinomialTER $\alpha\beta$.

6 Discussion et Conclusion

Dans ce TER, nous avons produit et intégré des bibliothèques Java, implémentant des nouvelles méthodes pertinentes et validées via des prototypes au sein du laboratoire LIRMM. Ces nouvelles méthodes de classifications sont maintenant intégrées dans l'environnement Weka. Elles peuvent être utilisées aussi bien via l'interface graphique qu'en ligne de commande. Ces nouvelles mesures sont particulièrement adaptées aux faibles volumes de données, et, comme vu au chapitre ??, ces mesures ont été testées dans un cadre supervisé notamment via Naive Bayes (Multinomial) et une approche basée sur les centroides (CFC).

Les expérimentations menées aussi sur nos deux corpus de 3 et 5 classes ont permis de montrer que celles-ci ont généralement un meilleur comportement que l'approche existante de Naive Bayes Multinomial.

Les différents objectifs de ce TER, qui sont :

- Prise en main de Weka
- Développement des différentes bibliothèques en java
- L'intégration dans l'écosystème Weka

ont été atteints, les difficultés principales rencontrées étant de bien comprendre à la fois l'architecture interne et le fonctionnement de WEKA, ainsi que l'intuition qui porte les nouvelles mesures, ont axé et constitué la ligne directrice de notre travail, où nous avons mené une étude bibliographique sur l'article scientifique présentant la nouvelle méthodologie [1], ainsi qu'un parcours prolixe de la documentation de WEKA (cf. Wiki de Weka 7).

7 Sources

- Wiki de Weka : <http://weka.wikispaces.com/Writing+your+own+Classifier>
- Naive Bayes : http://scikit-learn.org/stable/modules/naive_bayes.html

Références

- [1] M.Roche F.Bouillot, P.Poncelet. De nouvelles pondérations adaptées à la classification de petits volumes de données textuelles. In *Actes des 14ièmes Journées Francophone "Extraction et Gestion des Connaissances" (EGC 2014)*, 2014.