Automatic classification report in java

1st year, group C1,

BUT INFORMATIQUE

IUT 2 of Grenoble - 2 Pl. Doyen Gosse, 38000 Grenoble

Team composed of :

- Metayer Kylian

Table of contents

Introduction

The objective of this project was to produce an automatic classification program in Java. It had to classify different dispatches by determining their category.

To do this, we had in our possession various data such as several dispatches in a ".txt" format file. Then for each dispatch an identification number, a date, a category and a text.

We were also provided with different programs on which to base our automatic classification. Such as a Category, Classification, Dispatch object and finally examples of how to write a file in ".txt" format and calculate the execution time of a program.

The difficulty in this project is to simply and quickly read and calculate the information and values required to determine the category of a dispatch. Then to do it on a large scale (several dispatches).

Work completed

For this automatic classification program with the supplied database, we had to create several methods, to be precise four in a utility file for objects of type « PaireChaineEntier », five others in the « Classification » file and finally two in the « Catégorie » file.

Utility file for objects « PaireChaineEntier »

Note: an object of type « PaireChaineEntier » has two attributes. « Chaine » to type String corresponds to the object name, « Entier » to type Integer valeur the object value.

The purpose of the methods in this file is to calculate and search for objects of type « PaireChaineEntier » stored in vectors.

- The first, «indicePourChaine » has « listePaires » of type « PaireChaineEntier » and « Chaine » of type String as parameters. Returns the position (index) in « listePaires » of an object whose « Chaine » attribute is equal of the « Chaine » parameter.
- The second, « entierPourChaine », has the « listePaires » parameter of type « PaireChaineEntier » and the « Chaine » parameter of type String. Returns the « Entier » attribute of an object in « listePaires » whose « Chaine » attribute is the « Chaine » parameter.
- The third « chaineMax » with « listePaires » parameter of type « PaireChaineEntier » vector. Returns the « chaine » attribute of an object in « listePaires with the largest « Entier » attribute in the entire list.
- The fourth, « moyenne », whose parameter is « listePaires » of type vector « PaireChaineEntier», returns the average of all the « Entier » attributes of the objects in the list.

Category file

Note: a category object is composed of two attributes. The first is « nom » of type String, and the second is « lexique » of type « PaireChaineEntier » vector.

The purpose of the methods in this file is to calculate and store lexicons. Later, these data can be used to determine the category of a dispatch.

- The first, « initLexique », has the String parameter « nomFichier ». The method will read a ".txt" format file set as a parameter, with this [word]:[weight] parameter on each line. The program will read each line one by one and retrieve the word and the weight to create a « PaireChaineEntier » object with the word as the value of the « chaine » attribute, and the weight as the « entier »attribute, then store this object in a « PaireChaineEntier » vector and repeat the operation for each line of the file.

- The second « score » has a « d » parameter of type « dépêche ». The method will read the text of the dispatch added as a parameter, then return the sum of all « entier » attributes whose « chaine » attribute is a word in the dispatch text. To retrieve the integers, we'll use « entierPourChaine » from the utility file.

Classification file

- The first, « initdico » has « depeches », a vector type of « Depeche » and « catégorie » a String type, as parameters. The method traverses each dispatch of the same category given in parameter. Then it scans each of the words and stores them in a « PaireChaineEntier » vector if they don't already appear in it, with the « entier » attribute being the value zero. Finally, it returns a vector filled with words to be used as a dictionary for the category set as a parameter.
- The second, « calculScores » has as parameters « depeches » of type vector de « Depeche », « catégorie » of type String and « dictionnaire » of type vector « PaireChaineEntier ». The method scans the dispatches and each word of their text. If a word in the text appears in the dictionary and the dispatch belongs to the category set as a parameter, then the word in the dictionary will have its « Entier » attribute increased (incremented). If the dispatch's category is not the same, then the attribute will be reduced (decremented).

- The Third, « poidsPourScore » with parameter « score» of type Integer. The method returns an arbitrarily determined weight according to the score set as parameter.

- The fourth is « generationLexique », whose parameters are « depeches » of type « Depeche », « catégorie » of type String and « nomFichier » of type StringThe method reuses the previous methods, so that the category given as a parameter has its lexicon of words for which a score has been calculated, then determines a weight according to this score and finally writes a summary in a file in ".txt" format with the parameter « nomFichier » as its name, and each line inside presented in this way [words]:[weight].

- The fifth, « classementDepeches » with the parameters « depeches » of type vector « Depeche », « catégorie » of type vector « Catégorie » and « nomFichier » of type String. This program will scan each dispatch and then calculate 3 elements:

  1. The category with the highest score for the dispatch text, automatically classifying each dispatch.
  2. The number of times a category appears on the correction.
  3. The number of dispatches classified under the correct category compared to the original file.

Then, in a text file, the category assigned to the dispatch ID N°x is written in this way [ID]:[Category]. Then at the end, the percentage of correct answers for each category and the average correct answer for the whole category.

AnalyseurRSS file

We've also added a file with a program that can read RSS files, allowing us to reuse the data read to classify our dispatches with new lexicons.

Results obtained

Here is the success rate of our automatic classification of dispatches from the depeches.txt file

With the manual lexicon :

```
ENVIRONNEMENT-SCIENCES: 31%
CULTURE: 41%
ECONOMIE: 45%
POLITIQUE: 55%
SPORTS: 62%
MOYENNE: 46.8%
```

With the lexicon created automatically from the depeches.txt file:

```
ENVIRONNEMENT-SCIENCES: 99%
CULTURE: 100%
ECONOMIE: 93%
POLITIQUE: 98%
SPORTS: 100%
MOYENNE: 98.0%
```

With the lexicon created automatically from RSS resources :

```
ENVIRONNEMENT-SCIENCES: 13%
CULTURE: 70%
ECONOMIE: 49%
POLITIQUE: 14%
SPORTS: 33%
MOYENNE: 35.8%
```

Here is the success rate of our automatic classification of dispatches from the test.txt file

With the manual lexicon :

```
ENVIRONNEMENT-SCIENCES: 16%
CULTURE: 34%
ECONOMIE: 36%
POLITIQUE: 36%
SPORTS: 46%
MOYENNE: 33.6%
```

With the lexicon created automatically from the depeches.txt file:

```
ENVIRONNEMENT-SCIENCES: 71%
CULTURE: 82%
ECONOMIE: 75%
POLITIQUE: 71%
SPORTS: 89%
MOYENNE: 77.6%
```

With the lexicon created automatically from RSS resources :

```
ENVIRONNEMENT-SCIENCES: 11%
CULTURE: 71%
ECONOMIE: 53%
POLITIQUE: 10%
SPORTS: 19%
MOYENNE: 32.8%
```

The more words in the lexicon, the more precise we are. Manual lexicons are much less extensive than those created automatically. Except for lexicons created from RSS resources, we can assume that these resources speak of elements in a different context, so their words in common are consequently fewer.

Complexity analysis

For the « Score » method:

```
public int score(Depeche d) {
    int res = 0;
    for (int i = 0; i<d.getMots().size(); i++) {
        res += UtilitairePaireChaineEntier.entierPourChaine(lexique,
d.getMots().get(i));
    }
return res;
}
```

```
public static int entierPourChaine(ArrayList<PaireChaineEntier>
listePaires, String chaine) {
   int i = 0;
   int res = 0;
    while (i < listePaires.size() && res == 0) {
        if (listePaires.get(i).getChaine().compareTo(chaine) == 0) {
            res = listePaires.get(i).getEntier();
        }
        i++;
   }
   return res;
}
```

Instruction	Nb exec	Asymptotic
int res = 0;	1	$\Theta(1)$
for (int i = 0; i<d.getMots().size(); i++) {	n	$\Theta(n)$
res += UtilitairePaireChaineEntier.*entierPourChaine*(lexique, d.getMots().get(i));	1*4+n*4 ou 1*5	$4O(1) + O(n)$ ou $5\Omega(1)$
return res;	1	$\Theta(1)$
int i = 0;	1	$\Theta(1)$

int res = 0;	1	$\Theta(1)$
while (i < listePaires.size() && res == 0) {	N ou 1	$O(n)$ ou $\Omega(1)$

if (listePaires.get(i).getChaine().compareTo(chaine) == 0) {
res = listePaires.get(i).getEntier();
	n-1 ou 1	$O(n)$ ou $\Omega(1)$

i++;	0	
return res;	1	$\Theta(1)$

We sum: score is in 2 (1) + 4n$O(1)$ + $O(n^2)$ in the worst case

We sum: score is in 2 (1) + 5n$O(1)$ + $O(n)$ in the best case

Pour la méthode calculScores :

```
public static void calculScores(ArrayList<Depeche> depeches, String
categorie, ArrayList<PaireChaineEntier> dictionnaire) {
    int i = 0;
    while (i<depeches.size()) {
        for (int j = 0; j<depeches.get(i).getMots().size(); j++){
            String motSav = depeches.get(i).getMots().get(j);
            // Recherche dochotomique
            int born_A = 0, born_B = dictionnaire.size() - 1;
            int milieu;
            while (born_A < born_B) {
                milieu = (born_A + born_B) / 2;
                if (dictionnaire.get(milieu).getChaine().compareTo(motSav)
>= 0) {
                    born_B = milieu;
                } else {
                    born_A = milieu + 1;
                }
            }
            // Le mots à l'indice j de la dépêche à l'incide i apparaît
dans le lexique, et la catégorie du lexique n'est pas la même que celle de
la dépêche.
            if (dictionnaire.get(born_B).getChaine().compareTo(motSav) == 0
&& depeches.get(i).getCategorie().compareTo(categorie) != 0)  {

dictionnaire.get(born_B).setEntier(dictionnaire.get(born_B).getEntier() -
2); // retire -2 à l'entier du mots à l'indice born_B
                // Le mots à l'indice j de la dépêche à l'incide i apparaît
dans le lexique, et la catégorie du lexique est la même que celle de la
dépêche.
            } else if
(dictionnaire.get(born_B).getChaine().compareTo(motSav) == 0 &&
depeches.get(i).getCategorie().compareTo(categorie) == 0) {

dictionnaire.get(born_B).setEntier(dictionnaire.get(born_B).getEntier() +
1);// ajoute 1 à l'entier du mots à l'indice born_B
            }
        }
        i++;
    }
}
```

I don't know the complexity…

Conclusion

Finally, I think my classification system could be improved on many points, first and foremost its success rate. With a more precise lexicon and a more thoughtful calculation of the score of each word on a case-by-case basis, it would be possible to achieve much higher success rates than what I have proposed. Then simplified the program, if I had followed the subject less and taken more liberty in creating two or three more methods. I think that at my level I could have improved the readability and above all the speed of my program, especially on the « classementDepeche » method, which in my opinion is far too slow in execution.