REPORT – SAE 1.012

Léa GARAIX, Lynn HAYOT (D2)

Introduction

We had a week to develop a Java program, with the final goal of automatically classifying dispatches into appropriate categories (politics, culture, economics, environment-sciences and sports).To achieve this, we created lexicons for each category. They contain words frequently used in dispatches from the respective category, each assigned a weight of importance from 1 to 3. These lexicons are employed to identify keywords in the dispatches that may provide indications about their category. The weights of the identified words are then summed, and the category with the highest result is checked against the actual category of the dispatch to determine if it is correct. In the end, the goal is for the program to assign the category to the dispatch.

I – The program

To understand how the program works, it's essential to be familiar with three objects used in the program. One of them is `PaireChaineEntier`, which associates a string with an integer. It was very useful for lexicons and category scores. Another vital object is `Depeche` (a *dépêche* means *dispatch* in French), where information about each dispatch is stored. It includes its identifier ('id'), its release date ('date'), its category ('categorie'), and its content ('contenu'). Additionally, the words contained in the content are stored as an ArrayList of strings labeled 'mots'. The third object is `Categorie`, associating a category name with a lexicon ('lexique') specific to that category.

First step: providing lexicons to the program, from a handwritten text file.

We created the lexicons by examining the dispatches in the depeches.txt file, extracting keywords for each lexicon category. Each word was assigned a weight, following a simple syntax to ensure the program could interpret it later.

The `initLexique` method initializes the 'lexique' attribute of a category. It reads the file we created to obtain the necessary information and stores it in an ArrayList of PaireChaineEntier. To create the ArrayList<PaireChaineEntier>, the method requires both a string and an integer, necessitating the need to distinguish between them when processing each line. Initially, no sorting was applied to the ArrayList, but it was later introduced for program optimization.

Second step: determining the score of a dispatch in the categories.

The method `score` returns the sum of the weights of the words of the dispatch that have been found in the lexicon of the dispatch category.

By comparing the scores obtained by a dispatch in each category, we can identify the category that is supposed to be the best for it (note that the lexicons are not optimized yet, so it might not be the best

one). This comparison is done using the `chaineMax` method, which returns the string associated with the greater integer in an ArrayList. To test this process, we created in the JavaClass `Classification` a new ArrayList<PaireChaineEntier> gathering a category name and a score (obtained with `score`).

Third step: recording the results in a file.

With the help of the previous methods we can do a new one, `classementDepeches` that will classify all the dispatches in a category and then compare the results obtained to the real categories. At the end we obtain a percentage of the dispatches correctly classified for each category.

To do that we first create a new ArrayList <PaireChaineEntier> with the category and their score, which will be return at the end. Then, for each dispatch, we calculate the score and find the category which they seemed to belong thanks to the methods `score` and `chaineMax`, and we write it in a file**.** The next step consists in compare the category obtained with `chaineMax` to the real category of the dispatch. If the two categories are the same, we add 1 at the score of the category in the ArrayList initialized at the beginning.

At the end we write the score for each category in a file and the average of these scores to see the percentage of correct answers obtained with `chaineMax`.

Fourth step: developing methods to enable the program to create lexicons automatically.

To automate lexicon creation, we begin by breaking down the content of each dispatch. Every word in the dispacth becomes a string in an ArrayList of Strings (the ArrayList of words is a component of a Depeche object).

Next, we implement a method called `initDico`, which creates an ArrayList of PaireChaineEntier (named 'resultat') based on the 'Depeches' of a given category. For each Depeche of this category, the ArrayList of words from the Depeche is scanned word by word. Words not yet in 'resultat' are added. However, since they are Strings, they also need to be included in a PaireChaineResultat, associated with a score (initially set to 0; the value is later updated by the method below).

Fifth step: allocating a score to the words, and assigning a weight based on the score.

To assign a score to the words in an ArrayList created with `initDico`, we iterate through each word, comparing it to the content of all the dispatches in depeches.txt using the `calculScores` method. Additionally, a category is provided, matching the one used to create the dictionary.

The score of words is increased when they appear multiple times in the given category. Conversely, each time they appear in a dispatch from another category, the score is decreased. In the end, the words associated with higher scores are particularly valuable for determining a category. They appear frequently, if not exclusively, in dispatches for that category. Words unrelated to the given category have a negative score.

With these scores, we can estimate the weight of words using the `poidsPourScore` method. Words with a negative or null score are not considered in estimating the category of a dispatch, so their weight is set to 0 (since they already have the value 0).

For words with positive scores, we define intervals to determine the corresponding weight values (1, 2 or 3). If a word has a score of 3 or less, it is assigned a weight of 1. For scores between 4 and 5, the weight is set to 2. Words with higher scores are assigned a weight of 3.

Sixth step: generating a lexicon file.

We create a method called `generationLexique` that incorporates the previous methods. This method writes a lexicon in a dedicated file for each category.

Seventh step: incorporating new learning data from RSS feeds.

We gathered data from RSS feeds to enhance the lexicons and, consequently, improving the classification of dispatches. The RSS feeds from France TV Info were adapted to our needs as they are already organized by categories. However, the RSS feeds have a specific syntax which that differs from the syntax in depeche.txt and test.txt. To handle these dispatches, we developed another method called `lectureDepeches_rss`. The RSS data was copied into a text file, which the methods read. We specified the locations of the information for the program to search and create new Depeches.

Eighth step: optimizing the program.

Finally, we optimized the methods we previously wrote. One way to achieve this is by sorting the data before processing, allowing to use dichotomic search when needed instead of a sequential search. Dichotomic searches are faster.

The method `triBulles` in the UtilitairePaireChaineEntier class sorts the PaireChaineEntier objects in an ArrayList<PaireChaineEntier> based on string order. We used it in the `initLexique` and `ajouteLexique`. Following this, we introduced the method `entierPourChaineDicho`, a modified version of `entierPourChaine` that performs a dichotomic search.

II – Results

• To test the `initLexique` method, we chose to print the glossary of the SPORTS category. For each word in the glossary, we opted to display the word, its index, and the associated score as shown in the image.

```
Contenu du lexique à l'indice 0 :
- Le mot : antidopage
- Le poids : 3
```

For instance, the first word in the SPORTS category is “antidopage” with a score of 3, indicating it is a highly specific word for this category.

• The `entierPourChaine` method returns the score associated with the searched string, where the string is provided by the user.

```
Entrez un mot :
course
Le poids associé est: 2
```

In these examples, we can observe that the word “course” is used in dispatches of the SPORTS category with a score of 2, while the word “parapluie” is never used in the dispatches, resulting in a score of 0.

```
Entrez un mot :
parapluie
Le poids associé est: 0
```

• We tested the `Score` method with three objects of the Depeche class. The method returns the sum of the scores of each word in the Depeche.

In this example, we present the scores of three dispatches. The first and the second received a score of 0, indicating they don't contain any words that belong to the glossary of the SPORTS category. Thus, they likely belong to another category.

```
Le score de la dépêche 001 pour la catégorie sports est: 0
Le score de la dépêche 211 pour la catégorie sports est: 0
Le score de la dépêche 425 pour la catégorie sports est: 2
```

However, the third one received a score of 2, indicating that one or two of its words belong to the glossary of the SPORTS category. The score is relatively low, making it challenging to definitively conclude whether it belongs to this category or not. This result highlights the need to enrich the glossaries as they appear to be incomplete.

• With the `classementDepeches` method, we created the file testDepeches.txt classifying the dispatches into categories based on the glossaries. We then calculated the percentage of dispatches correctly classified using this method. Here are the results.

Considering the average, 67% of the dispatches were correctly classified by the method. The category with the lowest correct answer percentage is ENVIRONNEMENT-SCIENCES with 40%, possibly due to the use of common vocabulary found in other categories. Conversely, the category with the highest answer percentage is CULTURE, benefiting from a richer glossary that enhances the reliability of the classifications.

```
CULTURE: 90%
ECONOMIE: 62%
POLITIQUE: 67%
ENVIRONNEMENT-SCIENCES: 40%
SPORTS: 78%
MOYENNE: 67.0%
```

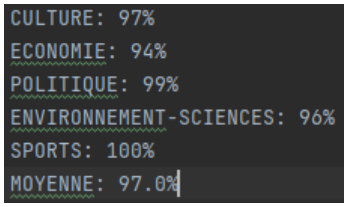
• We applied the `classementDepeches` method to another file, test.txt, containing additional news.

The results are very similar to the ones obtained with the file depeches.txt, but they are slightly lower. This difference can be attributed to the glossaries, which were created based on the content of depeches.txt. Therefore, it's logical to expect higher percentages with depeches.txt than with test.txt.

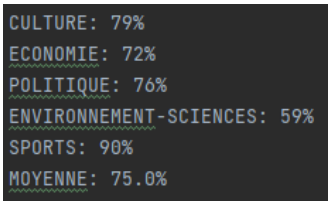
```
CULTURE: 89%
ECONOMIE: 53%
POLITIQUE: 63%
ENVIRONNEMENT-SCIENCES: 28%
SPORTS: 72%
MOYENNE: 61.0%
```

• After the creation of the new glossaries, we applied the `classementDepeches` method to the two files once again, and we obtained new results:

Results found in testDepeches2.txt:



Results found in testTest2.txt:



We can observe that the results are significantly higher than those of the first test. The glossaries are now more complete, providing more data for reference. As a result, the classification of the dispatches is more reliable, and the number of correct answers in each category increases considerably. However, there is still a difference between the results of depeches.txt and test.txt because the glossaries are still based on the content of depeches.txt.

III – Complexity analysis

- **`score(Depeche d)`**

In the `Score` method, the only line of comparison is: `for (int i=0; i<mots.size(); i++)` where I is compared to the size of the 'mots' ArrayList. We have 'i=0' at the beginning and 'mots.size()' means the number of words of the Depeche. The worst-case scenario involves scanning all elements of the mots ArrayList, and the best-case scenario is the same. The number of comparisons depends solely on the size of the ArrayList, resulting in linear complexity: (n) = **Θ**(n).

- **`calculScore(ArrayList<Depeche> depeches , String categorie, ArrayList<PaireChaineEntier> dictionnaire)`**

In the `CalculScore` method, the first line of comparison is: `for(int j=0; j<depeches.size(); j++)`. The worst and the best situations are the same: we have to scan all the elements of the ArrayList depeches (here there are five dispatches). We name n the number of elements in depeches.

The second line of comparison is: `for(int i=0; i<dictionnaire.size(); i++)`. Here 'i=0' and 'dictionnaire.size()' refers to the number of words in the dictionnaire ArrayList. The number of comparisons depends again only on the ArrayList's size and nothing changes between the worst and the best situation. We name 'm' the number of elements of dictionnaire.

The third line of comparison is `for(int k=0; k<depeches.get(j).getMots().size(); k++)`. Like the two other lines, the number of comparisons doesn't change between the best and worst situations. The program must scan all the elements of the ArrayList depeches.get(j).getMots() which represents the words in the dispatches. We name 'p' the number of elements of depeches.get(j).getMots().

The fourth line of comparison is `if(depeches.get(j).getMots().get(k).compareTo(dictionnaire.get(i).getChaine())==0)`. This comparison is performed m*n*p times, and each time we do one comparison. There is no difference between the best and the worst situations.

The last line of comparison is `if(depeches.get(j).getCategorie().compareTo(categorie)==0)`. We execute it only if the condition of the fourth line is respected. So, in the best situation, if no word in the dictionary is in the depeche, we do no comparison. And in the worst situation, if all the words in the dictionary are in the Depeche, we add each time one comparison to the total.

In the best situation, we perform m*n*p*(1+0) comparisons, and in the worst situation, m*n*p*(1+1) comparisons. The complexity is (n) = Ω(m*n*p) (n) = O(2*m*n*p).

Conclusion

While our program currently yields satisfactory results in terms of time and classification, there are potential avenues for improvement. For instance, the k-nearest neighbors algorithm was suggested to enhance our program; however, we spent time optimizing our program and preparing this report, so we were unable to explore this option.

Another avenue for improvement involves merging the lexicon created from RSS feed data with the lexicon from depeches.txt and sorting it. Additionally, we aim to incorporate more RSS feed data into the program to further enhance its capabilities.