SAE 1.012 – Automatic classification – Report

Table of contents

I - Introduction

Introduction to the Classification System Project

We need to code an "Automatic Classification" project as part of SAE S1.01. This project is a key part of our learning, where we are challenged to develop a program for the efficient and accurate classification of journalistic press dispatches into five categories: ENVIRONMENT-SCIENCES, CULTURE, ECONOMY, POLITICS, and SPORTS.

Our task involves the initial creation of lexicons for each category. We will manually select relevant words from sample press dispatches to build the foundation of our classification system to identify a category for each press dispatch. After this, we will use these lexicons to calculate scores for each category in a given press dispatch. The project is designed in several stages, from understanding the objectives, processing text data and to developing methods for automatic classification. It requires us to work with specific formatted files, such as press dispatch and lexicon files with a special format (keyword:power). We are expected to work in pairs and will present our results in a report and an oral presentation.

This project is an excellent opportunity for us to apply our theoretical knowledge in computer science, data analysis, and textual understanding in a practical and engaging way. It's a great way for us to demonstrate our skills and knowledge in a real-world context

I - Project Achievements

Project review :

Part 1

In this first part of the project, we first needed to manually write lexicons for each category and loaded them into arrays using the "initLexique" method in the Categorie class.

Reading files involved creating a method called "lectureDepeches" to get the press dispatches. We used FileInputStream and Scanner to parse the file and generate Depeche objects stored in an ArrayList.

To enhance efficiency, we introduced a class named "PaireChaineEntier" for creating dictionaries, consisting of two fields: "chaine" (a String) and "entier" (an int), representing frequently encountered data pairs in our project.

The "classementDepeches" method played a crucial role in classifying Depeches into categories, tracking correct classifications, and writing results to a file. We made it with the help of few methods :

- "entierPourChaine" method, which locates a string in an ArrayList of PaireChaineEntier and returns its index if found.
- "score" method to calculate the total score of a Depeche object
- "chaineMax" method to determine the word which has the maximum integer value in an ArrayList of PaireChaineEntier.
- "indicePourChaine" which searches for a specific string in an ArrayList
- "moyenne" to calculate the precision of our classification system.

In brief, It calculated category scores, identified the highest-scoring category for each press dispatch, and recorded correct classifications with accuracy percentages in the output file. This comprehensive process handled potential IOExceptions effectively

Part 2:

It was fine to create a hand-written lexicon, but we needed to improve our approach. In fact, the classification system wasn't precise with around 60% of correct detections.

The solution was to automate the entire process. To achieve this, we created a method to rank each word in the press dispatches, determining which words were repeated the most in each category. This approach allowed us to precisely identify the words to include in our lexicons.

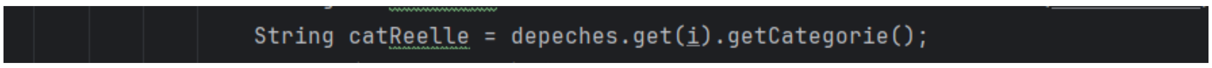
We made the "generationLexique" method which creates a lexicon file for a specified category from a list of Depeche objects with the help of three other methods :

- "initDico" which initializes a dictionary for a specific category from a list of Depeche.
- "calculScores" that adjusts the scores in a dictionary.
- "poidsPourScore" method that assigns a weight to a given score: 0 for scores of 0 or less, 1 for scores below 4, 2 for scores below 7, and 3 for higher scores.

Basically, "generationLexique" initializes a dictionary, adjusts scores based on word frequency, sorts the dictionary, filtering out unwanted characters and ensuring a beautiful formatting

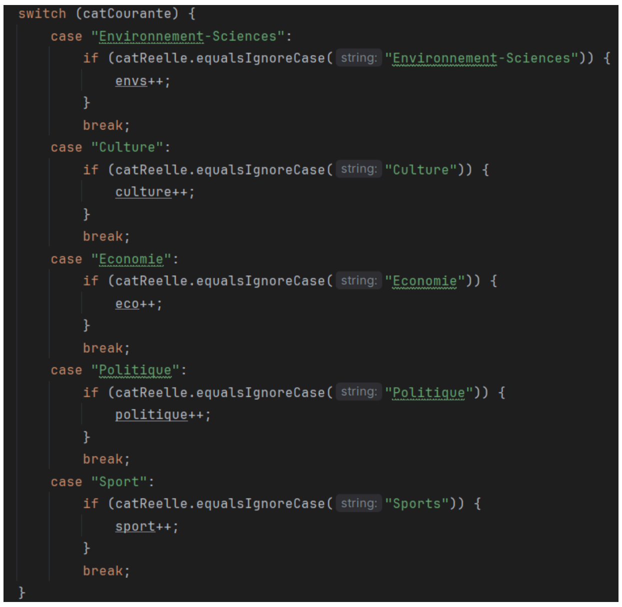
Improving program processing times:

Firstly, we have modified our "classementDepeches" method. As we call the same function several times, we have stored it in a variable to improve execution time.



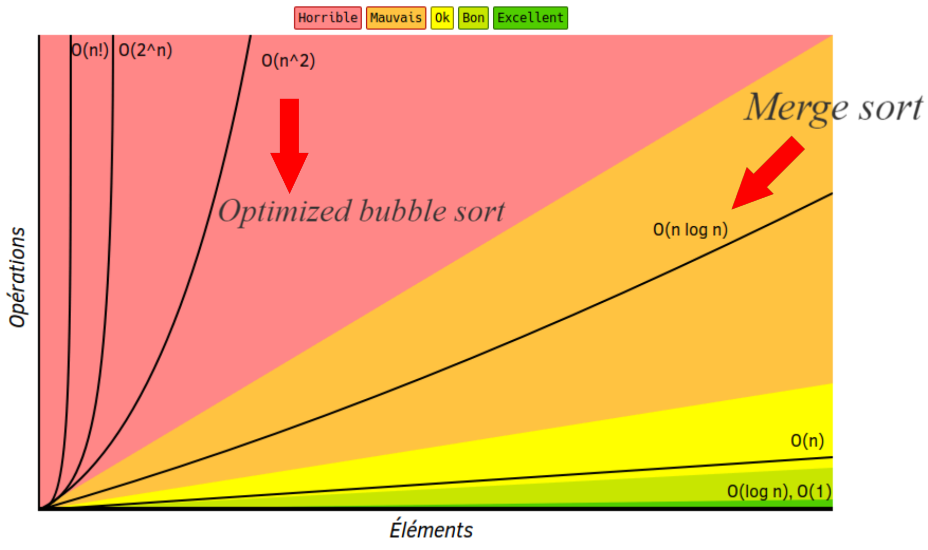
```
String catReelle = depeches.get(i).getCategorie();
```

Then, we had many "if" for each category. We replaced these "if" with a "switch case" which determines the category of depeches and adapts the situation.

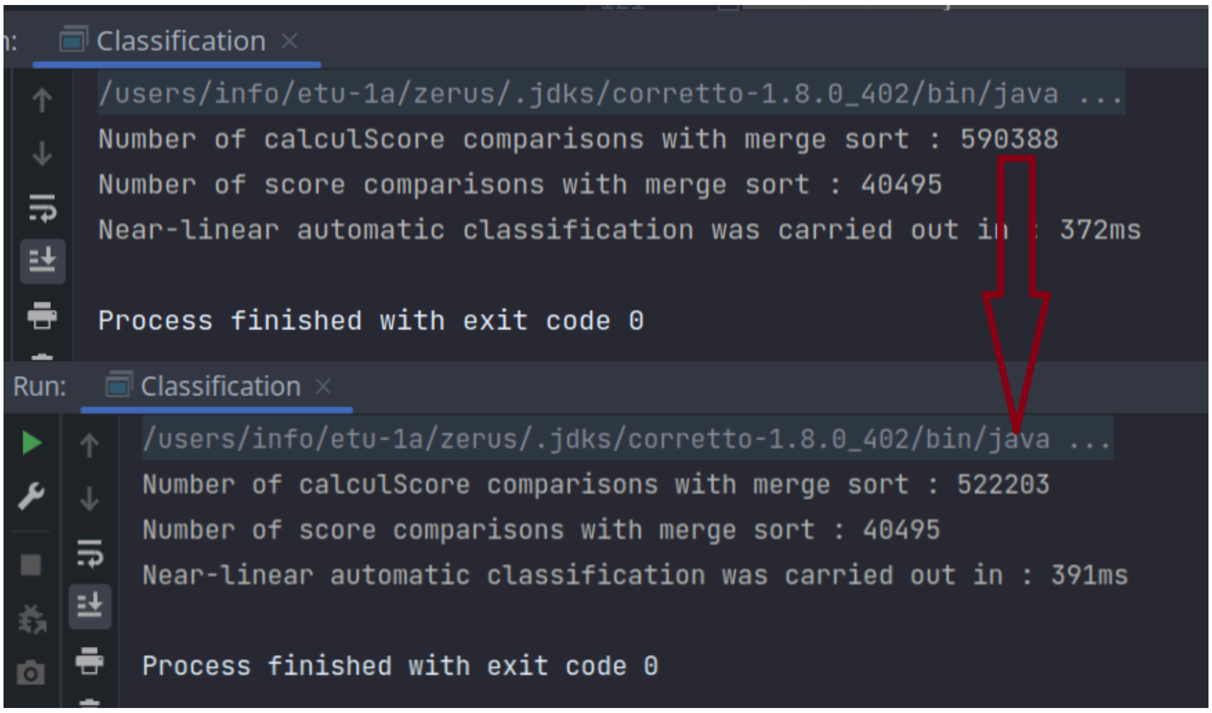


```java
switch (catCourante) {
    case "Environnement-Sciences":
        if (catReelle.equalsIgnoreCase( string: "Environnement-Sciences")) {
            envs++;
        }
        break;
    case "Culture":
        if (catReelle.equalsIgnoreCase( string: "Culture")) {
            culture++;
        }
        break;
    case "Economie":
        if (catReelle.equalsIgnoreCase( string: "Economie")) {
            eco++;
        }
        break;
    case "Politique":
        if (catReelle.equalsIgnoreCase( string: "Politique")) {
            politique++;
        }
        break;
    case "Sport":
        if (catReelle.equalsIgnoreCase( string: "Sports")) {
            sport++;
        }
        break;
}
```

We have also replaced our bubble sorting system with a fusion sorting system, which is much more efficient. We have gained a lot of time by changing this sorting. Bubble sorting is quadratic in average compared to near-linear fusion sorting.



Finally, we have improved our fusion sorting method. The goal was to lower our amount of comparisons



```
/users/info/etu-1a/zerus/.jdks/corretto-1.8.0_402/bin/java ...
Number of calculScore comparisons with merge sort : 590388
Number of score comparisons with merge sort : 40495
Near-linear automatic classification was carried out in : 372ms

Process finished with exit code 0
```

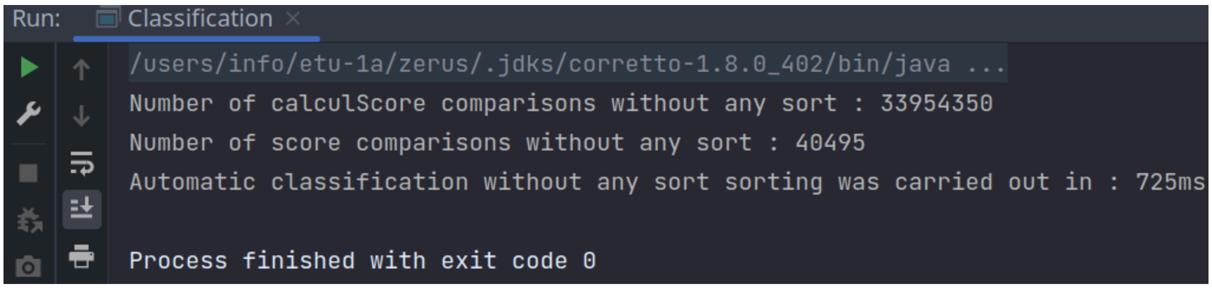
```
/users/info/etu-1a/zerus/.jdks/corretto-1.8.0_402/bin/java ...
Number of calculScore comparisons with merge sort : 522203
Number of score comparisons with merge sort : 40495
Near-linear automatic classification was carried out in : 391ms

Process finished with exit code 0
```

What we didn't do :

We did some research about the KNN classification system and we thought about making it, but we couldn't find a way to calculate the distance between press dispatches to categorize them. Our hypothesis was using the Levenstein distance algorithm, which measures the minimum number of modifications (additions, deletions, substitutions) required to change one text into another. It is best suited for short texts or for measuring similarity between sentences. With a bit more time and research we could have dug a bit more deeper about this part.

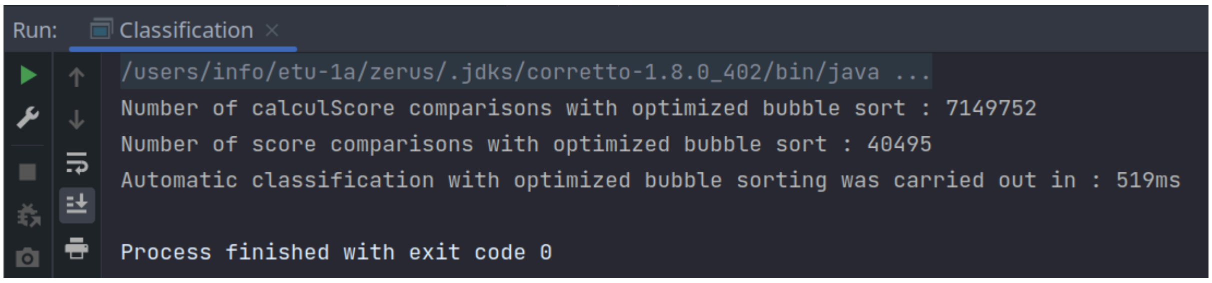
III - Performance (or complexity) analysis

Without sort and sequential search :



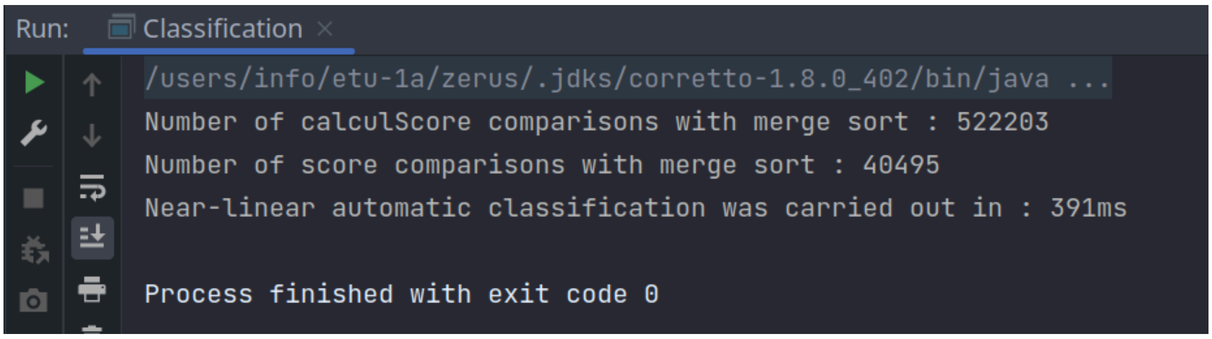
```
/users/info/etu-1a/zerus/.jdks/corretto-1.8.0_402/bin/java ...
Number of calculScore comparisons without any sort : 33954350
Number of score comparisons without any sort : 40495
Automatic classification without any sort sorting was carried out in : 725ms

Process finished with exit code 0
```

With optimized bubble sorting and dichotomy search (quadratic in average ~ O(n^2)):



```
/users/info/etu-1a/zerus/.jdks/corretto-1.8.0_402/bin/java ...
Number of calculScore comparisons with optimized bubble sort : 7149752
Number of score comparisons with optimized bubble sort : 40495
Automatic classification with optimized bubble sorting was carried out in : 519ms

Process finished with exit code 0
```

With merge sorting and dichotomy search (near-linear O(n log n)) :



```
/users/info/etu-1a/zerus/.jdks/corretto-1.8.0_402/bin/java ...
Number of calculScore comparisons with merge sort : 522203
Number of score comparisons with merge sort : 40495
Near-linear automatic classification was carried out in : 391ms

Process finished with exit code 0
```

As you can see, the results demonstrate a better sorting system. The merge dichotomy sorting system is the best with the lowest amount of comparisons and processing time.
It is why we choose to use it in the final version of this project

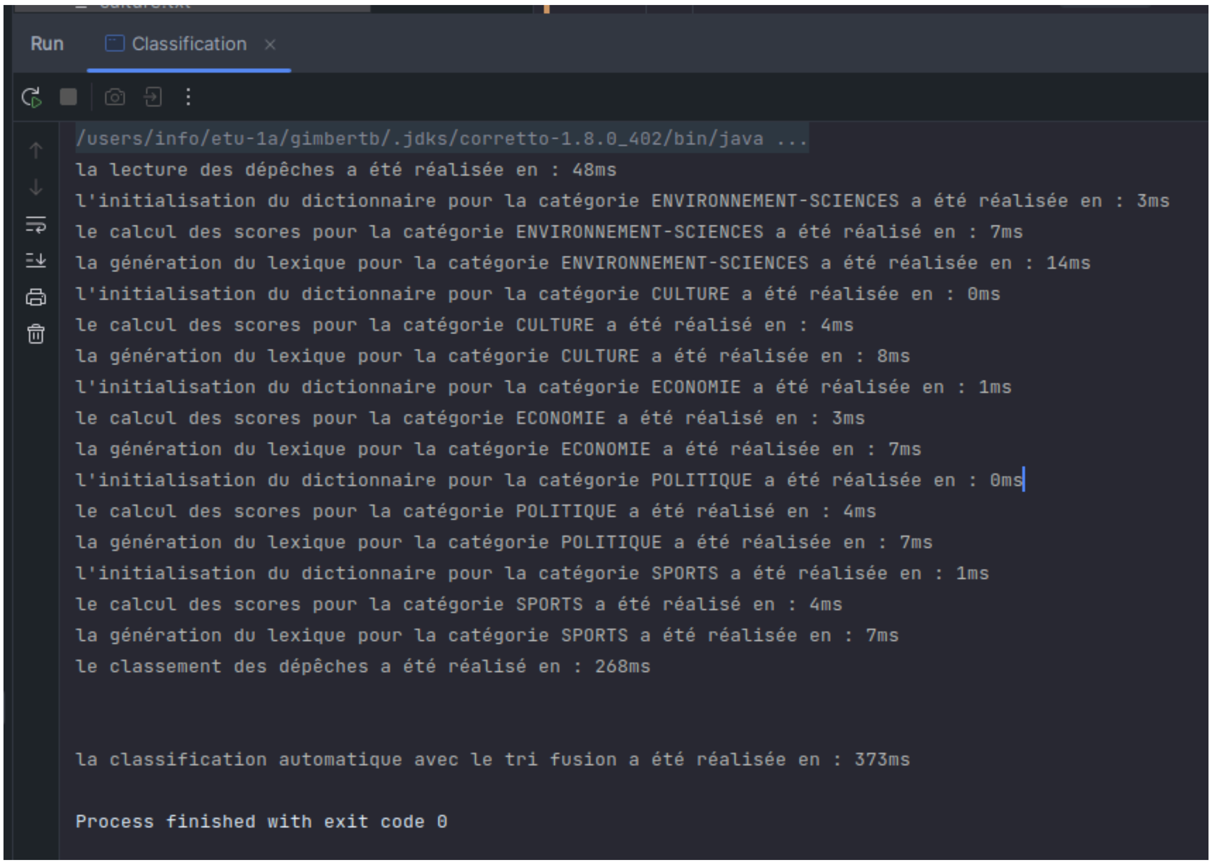
IV - Conclusion & Upgrade

We can conclude that the number of comparisons and the execution time is really variable depending on the method you choose, you have to adapt to your needs.
It's important to think things through and take a step back before you start coding so that you can choose the right method. We've optimized our 'artificial intelligence' as much as possible and carried out tests to get the best detection performance possible. We've achieved satisfactory results with a high detection rate (over 50% for a list of 21 daily news articles). This shows us the real reliability of our system, which is still far from perfect.

To improve the speed of our java code, we could use HashMaps instead of Arraylists. HashMaps allow us to attach a keyword to a value. They can only be in the form "key:value".

We could also use a "BufferedReader" instead of a "FileInputStream". BufferedReader would allow multiple lines to be read. This way, you can read files faster, rather than line by line.

We also discovered that the most time-consuming method was obviously “classementDepeches”. We know that improving it would sky rocket our overall performance. Now, the real question is how to make it better ? Well, this is a question that hasn’t been answered yet, but we’ll work on it.



```
/users/info/etu-1a/gimbertb/.jdks/corretto-1.8.0_402/bin/java ...
la lecture des dépêches a été réalisée en : 48ms
l'initialisation du dictionnaire pour la catégorie ENVIRONNEMENT-SCIENCES a été réalisée en : 3ms
le calcul des scores pour la catégorie ENVIRONNEMENT-SCIENCES a été réalisé en : 7ms
la génération du lexique pour la catégorie ENVIRONNEMENT-SCIENCES a été réalisée en : 14ms
l'initialisation du dictionnaire pour la catégorie CULTURE a été réalisée en : 0ms
le calcul des scores pour la catégorie CULTURE a été réalisé en : 4ms
la génération du lexique pour la catégorie CULTURE a été réalisée en : 8ms
l'initialisation du dictionnaire pour la catégorie ECONOMIE a été réalisée en : 1ms
le calcul des scores pour la catégorie ECONOMIE a été réalisé en : 3ms
la génération du lexique pour la catégorie ECONOMIE a été réalisée en : 7ms
l'initialisation du dictionnaire pour la catégorie POLITIQUE a été réalisée en : 0ms
le calcul des scores pour la catégorie POLITIQUE a été réalisé en : 4ms
la génération du lexique pour la catégorie POLITIQUE a été réalisée en : 7ms
l'initialisation du dictionnaire pour la catégorie SPORTS a été réalisée en : 1ms
le calcul des scores pour la catégorie SPORTS a été réalisé en : 4ms
la génération du lexique pour la catégorie SPORTS a été réalisée en : 7ms
le classement des dépêches a été réalisé en : 268ms


la classification automatique avec le tri fusion a été réalisée en : 373ms

Process finished with exit code 0
```