SAE 1.01/02

I. Introduction :

L'objectif de ce projet est de créer un programme capable de classer automatiquement les dépêches d'actualité aussi rapidement et précisément que possible. Une dépêche d'actualité est un court texte fournissant des informations journalistiques. Pour ce faire, le programme se concentrera sur les mots qui indiquent à quelle catégorie appartient la dépêche. Par exemple, des termes comme « match » et « joueur » suggéreront que la dépêche est liée au sport.

Chaque catégorie sera associée à un ensemble de mots clés permettant de déterminer son sujet. Chaque mot se verra attribuer une pondération indiquant son lien avec la catégorie. La pondération peut être de 1, 2 ou 3. Où 3 signifie que le mot est fortement associé à cette catégorie.

Par exemple, le lexique de la catégorie SPORT pourrait inclure des mots comme « sport » (3), « coupe » (3), « match » (3), « joueur » (2) et « domicile » (1). Dans la première partie du projet, nous créerons manuellement ces listes en examinant une centaine d'exemples de dépêches pour chaque catégorie. L'objectif est de sélectionner les mots les plus représentatifs de chaque catégorie, c'est-à-dire les mots spécifiques à cette catégorie et peu utilisés dans d'autres. Dans la deuxième partie, nous créerons une méthode pour générer automatiquement ces listes et, à l'issue du classement, obtenir un fichier unique.

II. Programme :

II.a) Mise en place du mode manuelle et automatique + résultat

Dans la première partie, nous avons mis en place la classification manuelle en utilisant le système (simplifiée) suivant :

-Pour chaque catégorie, regarder chaque dépêches de cette catégorie et mettre dans un fichier texte les mots les plus redondant (entre 40 et 50 mots)

-Créer la classe « PaireChaineEntier » qui va servir a plusieurs choses mais qui ici va permettre d’attribuer une catégorie et son score pour une dépêche

-Création de la fonction « classementDepeches » qui va utiliser plusieurs autres fonction (crée au préalable) pour créer un nouveau fichier texte de classement qui va afficher l’ID + la catégorie de chaque dépêches et a la fin le pourcentage de précision de chaque Catégorie ainsi qu’un moyenne globale

-Dans le main, on créer un objet Catégorie par catégorie et on y initialise le lexique qu'on a créer à l'étape 1 puis on appelle la fonction « classementDepeches »

Dans la deuxième partie, une génération automatique du lexique est mise en place en utilisant le nombre de mots similaire dans la même catégorie.

Nous allons donc étudiée les résultat sur un jeux de données « test »

```
SPORT: 71%
POLITIQUE: 73%
CULTURE: 65%
ECONOMIE: 43%
SCIENCES: 44%
MOYENNE: 59.2%
```

Manuelle

```
SPORT: 92%
POLITIQUE: 92%
CULTURE: 92%
ECONOMIE: 67%
SCIENCES: 78%
MOYENNE: 84.2%
```

Automatique

On constate que la version automatique est plus précise et dans la fonction « poidsPourScore » il est possible, en modifiant le barème de scores, d'augmenter le pourcentage de précision.

II.b) Ajout du flux RSS

Il est possible d'ajouter des dépêches d'une catégorie dans la base de donnée directement d'internet en utilisant la fonction « ajoutDepeches » et ceux-ci vont s'ajouter à la suite de la catégorie déjà présente à l'intérieur. Il est également possible d'extraire le lexique des nouvelles dépêches avec « generationLexique » qui va générer un lexique automatiquement.

II.c) Comparaison de la version non triée et triée

Nous avons ajoutée les fonction « classementDepeches_triee », « score_triee » et « entierPourChaine_triee » qui ensemble vont pouvoir calculée le score en faisant une recherche dichotomique dans un lexique triée. Nous avons donc du également ajouter la méthode « lexiques_triee » de la classe Categorie qui permet de prendre un fichier texte de lexique non triée et de crée un nouveau fichier similaire et triée.

Tout ceci va nous permettre de reprendre les lexiques générée automatiquement, les triée et les utiliser de la même façon qu'avant, c'est à dire en créant un classement automatiquement mais en comparant l'efficacité des versions non triée et triée.

Nous avons donc outillé les 3 fonctions précédente ainsi que leur version non triée pour regarder le nombre de comparaisons. Nous avons également ajoutée un calcule de temps de traitement totale et un calcule spécifique aux fonctions « classementDepeches » et « classementDepeches_triee ».

(Attention: le test a été effectuer sur mon pc personnel avec le fichier « test »)

Résultat :

```
Unsorted version:
the number of score comparisons : 18124408
file Résultat_Partie2.txt successfully created
Your ranking was made by: 112ms
sorted version:
the number of score comparisons : 1148302
file Résultat test_triee_partie2.txt successfully created
Your ranking was made by: 54ms
Total execution time: 217ms

Process finished with exit code 0
```

On peut constater que la versions non triée a beaucoup plus de comparaisons et que cela a un impacte sur le temps de traitement et donc devient lent pour de grande quantités de données alors que la version triée et elle beaucoup plus rapide et efficace. Dans l'exemple il y a 193 millisecondes de différence.

Nous allons donc poursuivre en utilisant la version triée.

II.d) Mise en place de la méthode KNN

Pour implémenter la méthode KNN, nous avons ajouté une fonction « methode_KNN » qui permettra de l'exécuter.

La méthode KNN consiste à trouver les K voisins les plus proches de la recherche de répartition parmi ceux de la base de données en fonction d'une distance, ici le nombre de mots en commun, afin de trouver la catégorie la plus répétée parmi les K voisins.

Nous utiliserons ici la classe PaireChaineEntier, qui aura pour chaîne la catégorie de la répartition de la base de données et, au total, le nombre de mots en commun avec la répartition que nous cherchons à déterminer.

Après avoir parcouru chaque mot de chaque répartition, nous obtenons une Arraylist de PaireChaineEntier, qui sera triée (par « tri_entier ») par Integer. Nous conserverons ensuite les K premiers et rechercherons la catégorie parmi les K voisins (avec « choix_categorie »). Nous avons ajouté une boucle qui utilisera cette méthode pour chaque fichier de test et avec la base de données. L'ID et la catégorie déterminés par la méthode sont affichés dans le fichier test_knn, puis une moyenne indiquant le pourcentage de réponses correctes est ajoutée. Nous avons ainsi pu déterminer pour quels K la méthode est la plus efficace.

Résultat :

Pour k = 3 moyenne : 51.8%

Pour k = 4 moyenne : 55.2%

pour k = 5 moyenne : 53.6%

Donc la méthode est plus efficace avec k=4.

II.e) Création de Classification 2

Classification 2 est une nouvelle classe qui va se baser sur la classe Classification mais qui automatiser absolument tout pour permettre de traiter n'importe quelle données.

Le système fonctionne de la façon suivante :
-Nous avons crée une nouvelle fonction « classement_automatique » qui prendre une liste de dépêches et qui va renvoyer un fichier classement et par intermédiaire les différent lexiques de chaque catégorie présent parmi les dépêches.
-Cette fonction va commencer par parcourir chaque dépêches afin d'extraire les différentes catégories puis par la réutilisation de « generationLexique » va créer un lexique pour chaque catégorie
-Puis va triée les lexiques avec la méthode « lexiques_triee ».
-Il va finir en appelant la fonction « classementDepeches_triee » qui va créer le fichier de classement basée sur les lexiques et les catégories trouvées précédemment.

Dan le fichier « lexiques_classement_automatique » on a tester avec les données de depeches2 et on a obtenus le résultat suivant :

```
CINEMA: 96%
GEEK: 92%
LIVRES: 93%
MUSIQUES: 90%
TV-RADIO: 77%
MOYENNE: 89.6%
```

On peut conclure que le système marche parfaitement, automatiquement et avec un nombre de comparaisons fortement diminuer.

III. Conclusion :

Afin d'améliorer notre programme, nous pourrions chercher des moyens d'améliorer nos résultats en optimisant notre code et en le complexifiant, mais notre niveau de programmation ne nous le permet pas pour le moment. Nous pourrions également améliorer l'ergonomie des tests afin de les rendre plus compréhensibles et efficaces pour la résolution de problèmes de code.

Cependant, en étudiant la complexité de « calculScores », qui se situe entre 40 et 45 millions de comparaisons selon le lexique généré, je pense qu'il serait possible de réduire ce nombre en triant directement les mots des lexiques lors de leur création, notamment en modifiant la fonction « initDico ».