1 Question 2.1.10

Table 1: Effet de k sur la performance de classification (Accuracy) avec 128 Hash Tables

Setup (Scénario)	$k = 50$	$k = 100$	$k = 150$	$k = 200$
Scénario 1 : Unigrammes (Mots)	0.62	0.62	0.62	0.62
Scénario 2 : Bigrammes (2-gram)	0.62	0.62	0.62	0.62
Scénario 3 : Patterns fréquents	0.61	0.61	0.61	0.61
Scénario 4 : Combinaison (Article)	0.60	0.60	0.60	0.60

Comparaison : Contrairement à l'attente intuitive, le Scénario 4 (Combinaison) performe légèrement moins bien (0.60) que les approches plus simples comme les Unigrammes ou Bigrammes (0.62).

Explication possible : L'ajout de trop de caractéristiques (patterns + n-grams + mots) dans le Scénario 4 peut introduire du "bruit" ou rendre les signatures MinHash moins discriminantes avec seulement 128 tables de hachage, diluant l'information utile.

Stabilité : On remarque une très grande stabilité des résultats quel que soit k (variant à peine entre 0.60 et 0.62). Cela indique que les 50 premiers voisins sont déjà très représentatifs de la classe dominante.

2 Question 2.1.11

2.1 Implémentation du Filtre de Bloom

Contrairement à la méthode `HashingTF` standard de Spark qui projette chaque terme vers un unique index (provoquant des collisions directes), nous avons implémenté un **Filtre de Bloom**. Cette structure probabiliste permet de tester l'appartenance d'un élément à un ensemble avec une gestion contrôlée des faux positifs.

Pour chaque token t d'un tweet, au lieu d'activer un seul bit dans le vecteur de caractéristiques, nous activons k bits distincts. Pour garantir l'efficacité computationnelle et éviter de calculer k fonctions de hachage lourdes (comme MD5 ou SHA-256), nous avons utilisé l'optimisation de **Kirsch-Mitzenmacher**. Cette technique permet de simuler k fonctions de hachage indépendantes à partir de seulement deux fonctions de base $h_1(x)$ et $h_2(x)$:

$$g_i(x) = (h_1(x) + i \cdot h_2(x)) \pmod{m} \tag{1}$$

Où :

- m est la taille du vecteur de caractéristiques (vector size).
- $h_1(x)$ est calculé via `CRC32` (rapide).
- $h_2(x)$ est calculé via `Adler32` (rapide et distinct de CRC32).

- i varie de 0 à $k-1$ (nombre de hachages).

Cette approche densifie l'information binaire du tweet. Si deux tweets partagent sémantiquement les mêmes mots, leurs signatures auront une distance de Jaccard plus significative qu'avec un simple hachage unique, augmentant ainsi potentiellement la robustesse de la classification k-NN.

3 Question 2.1.12

3.1 Comparaison et impact du Filtre de Bloom

Nous avons comparé notre meilleur classeur standard (Scénario 2 : Bigrammes avec HashingTF) au nouveau classeur intégrant un Filtre de Bloom (Scénario 5), en maintenant les hyper-paramètres constants ($L = 128$ tables, $k = 100$).

Table 2: Comparaison des performances : HashingTF vs Filtre de Bloom

Métrique	**Standard (HashingTF)**	**Filtre de Bloom ($h = 8$)**	**Variation**
Exactitude (Accuracy)	0.6194	**0.6217**	+0.23%
Temps d'entrainement (s)	~ 0.010 s	~ 0.064 s	$\times$6.4 (plus lent)
Temps d'évaluation (s)	~ 24.70 s	~ 39.65 s	$\times$1.6 (plus lent)

Analyse des résultats :

1. **Impact sur l'Exactitude :** Comme le montre le tableau 2, l'intégration du Filtre de Bloom apporte une légère amélioration de l'exactitude (+0.23%), atteignant un pic de **62.28%** avec 250 tables (voir courbes ci-dessous). Cette amélioration s'explique par la nature du Filtre de Bloom qui projette chaque mot sur $h = 8$ indices différents. Cette redondance réduit l'impact des collisions de hachage accidentelles qui pourraient survenir avec un simple `HashingTF`, préservant ainsi mieux la sémantique des tweets dans l'espace de signature MinHash.

2. **Impact sur les Temps de calcul :** L'amélioration de la précision se fait au prix d'un coût computationnel accru :

   - **Entraînement :** Le temps de vectorisation a été multiplié par 6. Contrairement à `HashingTF` qui est une fonction native optimisée, notre implémentation du Filtre de Bloom nécessite le calcul de plusieurs hachages (CRC32, Adler32) pour chaque mot en Python, ce qui introduit une latence CPU.
   - **Évaluation :** La recherche des plus proches voisins est environ 60% plus lente. Les vecteurs issus du Filtre de Bloom étant plus denses (plus de bits activés), le processus de LSH génère potentiellement plus de candidats à vérifier ou des comparaisons de signatures plus coûteuses.

Conclusion : L'intégration du Filtre de Bloom permet de franchir le seuil des 62% d'exactitude, validant son utilité pour capturer des nuances fines dans un contexte de haute dimensionnalité. Cependant, pour une application temps réel stricte, le surcoût de traitement (passage à l'échelle moins efficace en temps) doit être pris en compte.