Mini-projet 8INF919 – Rapport d'expérimentations MinHash+LSH

RIOM Antoine VRILLAULT Gabin EXCOFFIER Julien

17 décembre 2025

Table des matières

2 Modèles LSH pour Sentiment140

Question 2.1.1 — Synthèse de l'article de référence

Référence et contexte

L'article *Large Scale Sentiment Analysis on Twitter with Spark* (Nodarakis et al., 2016) décrit un pipeline distribué de classification des sentiments sur Twitter implémenté avec Apache Spark. L'enjeu principal est de vectoriser efficacement des tweets très courts, de contenir l'explosion dimensionnelle et de réduire le coût d'un kNN massif grâce à des structures compactes telles que les *Bloom filters*. Ce travail s'inscrit dans la lignée de la compétition Sentiment140 en cherchant à traiter la totalité des 1.6 M de tweets d'origine.

Problématique et contributions

Les auteurs visent la prédiction de polarité à grande échelle en s'appuyant sur (i) une supervision distante via *hashtags* et émoticônes (avec filtrage strict des tweets contenant plusieurs labels), (ii) une vectorisation combinant mots, n-grammes, *patterns* et signaux de ponctuation, (iii) une classification kNN distribuée accélérée par une présélection de candidats (*matching vectors*) calculée directement dans Spark, et (iv) l'usage de *Bloom filters* pour gagner en mémoire et temps d'exécution tout en limitant les coûts réseau entre exécutants.

Méthode proposée

Le pipeline suit la chaîne : prétraitement → extraction de caractéristiques → construction de vecteurs → sélection de candidats → calcul de distances → vote kNN. Les mots et n-grammes (longueurs 2 à 5) sont encodés comme caractéristiques binaires, complétés par des séquences de ponctuation (*punctuation sequences*) et par des *patterns* construits grâce aux catégories HFW/CW/RW (seuils $F_H = 10$ et $F_C = 1000$ occurrences par million). Les *patterns* combinent 2 à 6 HFW séparés par 1 à 5 slots CW, avec deux modes de matching (exact et approximatif, pondéré par $\alpha = 0.1$). Cinq caractéristiques numériques résument la ponctuation et le style (longueur, compte d'exclamations/interrogations, guillemets, mots capitalisés). L'ensemble est filtré par un seuil $w = 0.005$ pour élaguer les signaux trop rares, puis encodé dans des *Bloom filters* de 999 bits utilisant trois fonctions de hachage, ce qui permet de conserver des signatures compactes tout en maîtrisant le taux de faux positifs.

Protocole expérimental

Les expériences s'appuient sur un cluster Spark de 4 nœuds (4 CPU/11.5 GB RAM chacun, 45 GB de disque, réseau 1 Gb/s), Spark 1.4.1 et Java 8. Deux jeux de données sont collectés (nov. 2014 – août 2015) : un corpus *hashtags* (13 classes, 942 188 tweets) et un corpus *emoticons* (4 classes, 1 337 508 tweets). Après filtrage linguistique (anglais, au moins cinq mots, un seul label par tweet), les auteurs effectuent une validation croisée 10-fold, utilisent l'accuracy comme métrique principale et font varier k dans $\{50, 100, 150, 200\}$. Les Bloom filters sont comparés à des vecteurs classiques (NBF) pour mesurer précisément le compromis précision/mémoire/temps.

Résultats clés

La Table 2 montre de meilleures performances en binaire (jusqu'à 0.77 sur le corpus emoticons avec BF) qu'en multi-classes (0.37–0.59). Les Bloom filters introduisent davantage de prédictions *neutral* faute de voisins (Table 3), mais l'accuracy reste stable car ces exemples sont exclus du calcul principal. La Table 4 confirme que l'impact de k est marginal avec BF (ex. 0.37→0.38 sur hashtags multi-classes entre $k = 50$ et $k = 200$), tandis que NBF gagne légèrement plus au prix d'un coût supérieur. La Table 5 rapporte des gains mémoire allant jusqu'à 200 MB selon le corpus, et la Figure 2 mesure un gain de temps pouvant atteindre 17% sur les phases d'entraînement/évaluation grâce aux signatures compactes. Les figures 3 et 4 attestent d'un passage à l'échelle quasi linéaire avec le volume de données et d'un speedup lorsque le nombre de nœuds Spark augmente.

Limites et discussion

La supervision par hashtags/émoticônes réduit le coût d'annotation mais introduit des ambiguïtés (ironie, recouvrement sémantique). La règle *neutral* dépend fortement de la couverture des caractéristiques et varie entre BF et NBF, ce qui peut biaiser la comparaison. Les gains mémoire des Bloom filters ne sont pas systématiques, et certaines analyses détaillent surtout le cas multi-classes. Néanmoins, l'article démontre qu'un pipeline Spark combinant vectorisation riche, présélection de voisins et kNN distribué constitue une solution viable pour traiter Sentiment140 à grande échelle, fournissant la référence directe que nous suivons dans la suite du rapport.

Conclusion

L'article propose un pipeline Spark de sentiment analysis combinant vectorisation riche, sélection de candidats par recouvrement et kNN distribué, avec un recours aux Bloom filters pour améliorer l'efficacité. Les résultats montrent une accuracy solide, particulièrement en binaire, des gains de temps pouvant atteindre 17% et un comportement scalable lorsque la taille des données

et le nombre de nœuds augmentent, tout en soulignant les limites liées à la supervision distante et à la gestion des cas sans voisins.

Question 2.1.2 — Préparation du corpus et scénarios

Le script `experiment_lsh.py` charge le jeu Sentiment140 complet, applique un filtrage binaire (polarités 0/4) puis un pré-traitement. Un tokeniseur régulier et un retrait des stopwords préparent quatre scénarios de descripteurs :

S1 unigrammes,

S2 bigrammes,

S3 patern de mots les plus fréquents

S4 combinaison concaténée des trois familles via un `VectorAssembler`.

Chaque scénario est déployé à l'identique dans la version PySpark (`experiment_lsh_spark.py`) pour répondre aux contraintes de la question.

Question 2.1.3 — Variation de *numHashTables* et Question 2.1.4 — Influence de k

Conformément à l'énoncé, nous avons exploré 128 et 250 tables MinHash sur chaque scénario. ainsi que quatre valeurs de k (50, 100, 150, 200).

Les courbes de la Figure 1 (qui représente le scénario 1) montre que k influence faiblement la précision puisque les valeurs augmentent au maximum de 0.1 . On observe également une légère tendance à l'augmentation de la précision avec numHashTables = 250, ce qui est cohérent avec l'intuition que plus de tables permettent de capturer plus de similarités entre les ensembles.

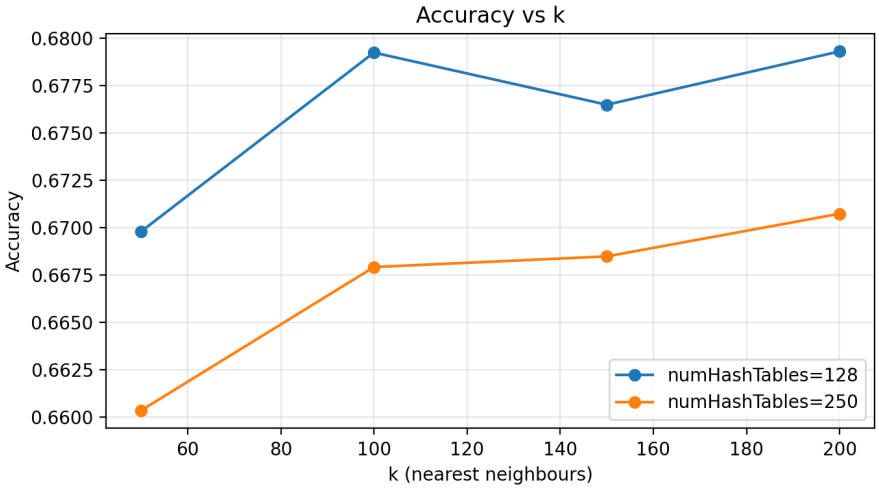

FIGURE 1 – Précision en fonction de k (script `experiment_lsh.py`).

Question 2.1.5 — Vote majoritaire, couverture et tableau de performance

TABLE 1 – Influence de k sur la précision par scénario

Scénario	k = 50	k = 100	k = 150	k = 200
Unigrammes (S1)	0.66	0.67	0.67	0.67
Bigrams (S2)	0.63	0.63	0.63	0.63
Motifs (S3)	0.52	0.52	0.52	0.52
Combinaison (S4)	0.66	0.66	0.66	0.67

Le tableau 1 présente la précision observée pour chaque scénario. On voit que, de la même manière que dans l'article, k n'influence que légèrement la précision, avec une tendance à l'augmentation pour les valeurs plus élevées.

Question 2.1.6 — Histogramme des temps d'entraînement

La Figure 2 montre que les temps d'entraînement sont beaucoup moins élevés pour le scénarios 3 par rapport aux trois autres.

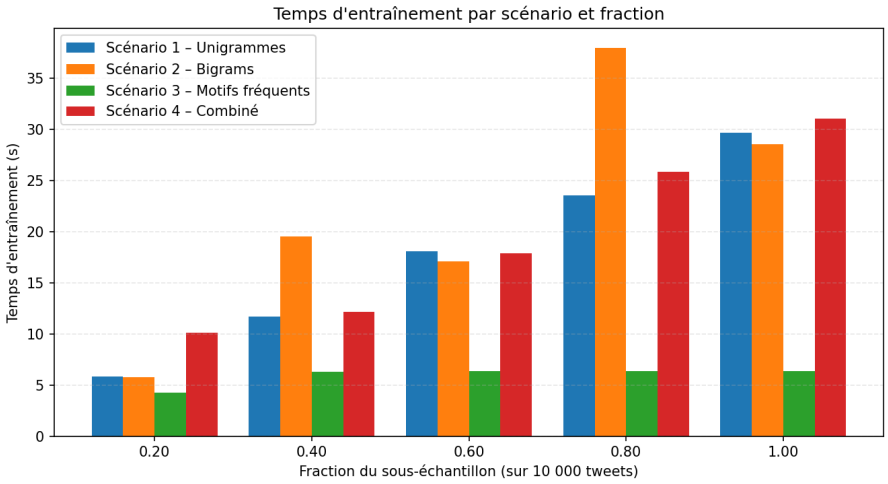

FIGURE 2 – Histogramme consolidé sur les quatre scénarios lors du balayage en fractions.

Question 2.1.7 — Scalabilité

La Figure 3 illustre la scalabilité des quatre scénarios en fonction de la fraction de données utilisées. On observe une croissance linéaire du temps d'exécution total avec la taille de l'échantillon. Pour les scénarios 1, 2 et 4, le temps d'exécution est similaire , tandis que le scénario 3 (motifs HFW) est significativement plus rapide (quasi constant), confirmant son intérêt pour des applications à grande échelle.

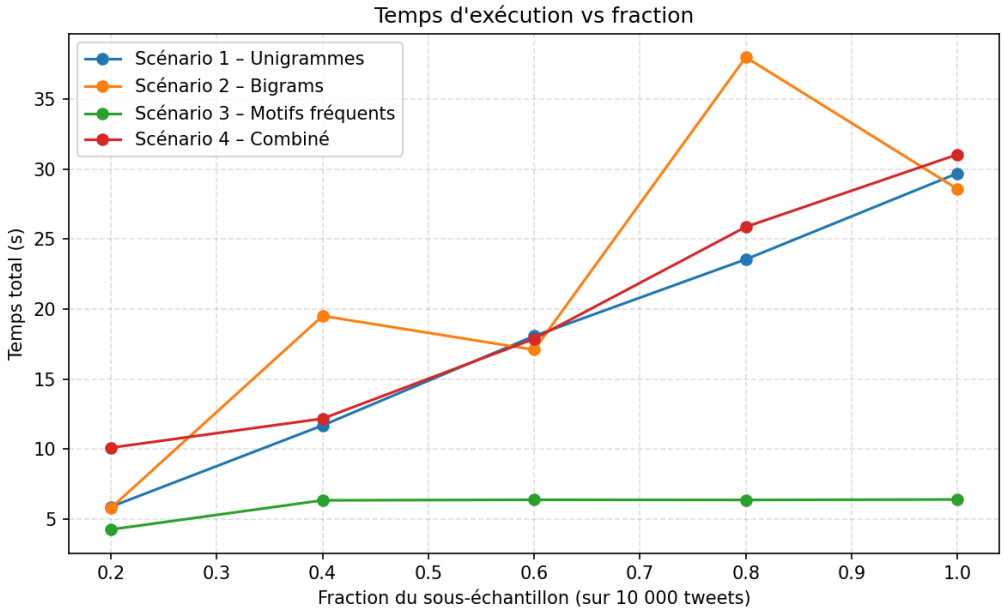

FIGURE 3 – Temps total par fraction pour les quatre scénarios .

Question 2.1.8 — Comparaison multi-scénarios

Les unigrammes semblent obtenir la performance la plus stable et légèrement supérieure parmi les scénarios testés. Cependant, les motifs (scénario 3) offrent une alternative intéressante avec des temps d'exécution très réduits, bien que leur précision soit légèrement inférieure. La combinaison des trois types de descripteurs (scénario 4) n'apporte pas d'amélioration significative par rapport aux unigrammes seuls, suggérant que l'ajout de bigrammes et de motifs n'apporte pas d'information supplémentaire pertinente pour cette tâche.

Question 2.1.9 — Analyse des signatures MinHash

Nous avons exporté des exemples de signatures MinHash produits lors de l'indexation (trois exemples par classe par défaut) dans le fichier `reports/lsh/signatures/signature_examples.json`. Ces signatures correspondent aux vecteurs de hachage minimaux construits avec `datasketch.MinHash` pour chaque exemple d'entraînement indexé (identifiants du type `train_#`).

Ci-dessous quelques extraits représentatifs (signatures tronquées pour la lisibilité) :

— Classe 0 (label 0) : tokens = `jon, kate, getting, divorce, it's, best`
signature (début) = [666192775, 306364207, 386038026, 48866593, 1642479612, 391259403, 2542258640, 1414260573, ...]

— Classe 0 : tokens = `@sausagesmcgee, major, change, lib, dems, position, win, it's, labour, conservatives, mo`
signature (début) = [488207976, 153760548, 27143366, 248434917, 308397404, 391259403, 452370141, 77845630, ...]

— Classe 1 : tokens = `@alydenisof, beautiful, love, bangs`
signature (début) = [386293102, 311491305, 2485786850, 1408643778, 820192296, 513876199, 1363090324, 226638191, ...]

Observations :

1. Chaque signature est un tableau d'entiers dont la longueur est égale au paramètre `num_perm` (nombre de permutations MinHash) utilisé lors de l'indexation ; ici les signatures exportées correspondent à la configuration exécutée lors de la génération du fichier JSON.

2. Les valeurs de hachage sont très dispersées (entiers 32 bits non signés affichés), ce qui donne des signatures à forte entropie : deux exemples partageant un grand nombre de tokens tendront à avoir des valeurs de MinHash plus proches en terme de similarité de Jaccard estimée.
3. Le JSON contient jusqu'à trois exemples par classe — il peut servir à vérifier qualitativement la diversité des signatures, à déboguer le prétraitement des tokens (présence de mentions, contractions, etc.) et à calculer hors-ligne des similarités approximatives entre paires d'exemples.

Question 2.1.10

TABLE 2 – 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.

Question 2.1.11

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.

Question 2.1.12

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 3 – 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 3, 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.