Dauphine | PSL
UNIVERSITÉ PARIS

Projet Machine Learning sur Big Data

k-NN and Naive Bayes

Fatoumata Wadiou & Pénélope Millet

August 31, 2025

Contents

1 Introduction

Le traitement de grandes quantités de données structurées est un enjeu majeur en sciences des données et en apprentissage automatique. Les frameworks distribués, tels qu'Apache Spark, permettent de paralléliser les calculs sur plusieurs nœuds, rendant possible l'implémentation efficace d'algorithmes coûteux comme k-nearest neighbors (k-NN) ou Naïve Bayes. L'objectif de ce travail est de comparer différentes approches de ces algorithmes dans Spark, en évaluant précision, performances et scalabilité selon la taille des données et le niveau de parallélisme. k-NN et Naïve Bayes ont été implémentés via RDD, DataFrame et Spark ML pour examiner l'impact des choix d'abstraction sur la clarté, la fidélité au paradigme MapReduce et les performances. Pour chaque algorithme étudié, le rapport décrit d'abord les méthodologies, puis présente les expériences et les résultats, en mettant l'accent sur l'accuracy, les temps d'exécution et l'efficacité du parallélisme.

2 Naive Bayes

2.1 Compréhension de l'algorithme

Cette section résume l'algorithme du papier de Zheng et al. [1].

Intuition. Nous voulons prédire la *classe* d'un objet (par exemple, *spam* ou *non-spam*) à partir de ses *attributs* $A_1, \ldots, A_M$ (mots présents, âge, couleur, etc.). L'idée de Naïve Bayes est de dire : si l'on connaît la classe, alors les attributs se comportent comme s'ils étaient indépendants les uns des autres. Cette hypothèse, bien que simplificatrice, permet d'estimer des probabilités à partir de simples comptages et de combiner ces informations de façon très efficace.

Concrètement :

- on compte, dans les données d'entraînement, combien d'exemples tombent dans chaque classe (N_j) et, pour chaque attribut A_i et chaque valeur possible a, combien de fois la combinaison *(classe c_j, attribut $A_i = a$)* apparaît (noté $n_i(a, c_j)$) ;
- on transforme ces comptes en probabilités estimées : la probabilité a priori de la classe et, pour chaque attribut, la probabilité conditionnelle d'observer sa valeur sachant la classe ;
- pour un nouvel objet $x = (x_1, \ldots, x_M)$, on calcule pour chaque classe un score qui agrège ces probabilités (on additionne des log-probabilités pour éviter les problèmes numériques);
- on prédit la classe au score maximal.

Deux remarques pratiques : (1) si des attributs sont continus, on les discrétise (binning) pour pouvoir compter ; (2) pour éviter des probabilités nulles quand une combinaison n'a jamais été vue, on applique une petite correction dite *zéro-fréquence* en ajoutant +1 aux comptes avant de calculer les probabilités.

Formalisation probabiliste du modèle. Pour une instance d et une classe $c \in C$, le théorème de Bayes donne

$$P(c \mid d) = \frac{P(d \mid c)\, P(c)}{P(d)}.$$

La classe prédite maximise la probabilité a posteriori :

$$c_{\text{MAP}} = \arg\max_{c \in C} P(c \mid d) = \arg\max_{c \in C} P(d \mid c)\, P(c).$$

Si d est représentée par ses attributs $(x_1, \ldots, x_M)$ et en supposant l'indépendance conditionnelle des attributs étant donné la classe :

$$P(x_1, \ldots, x_M \mid c) = \prod_{i=1}^{M} P(x_i \mid c),$$

on obtient la règle de décision de Naïve Bayes :

$$\hat{c}(d) = \arg\max_{c \in C} \; P(c) \prod_{i=1}^{M} P(x_i \mid c).$$

Estimation des paramètres. Étant donné un échantillon d'entraînement de taille N et N_c occurrences de la classe c, on estime

$$\widehat{P}(c) = \frac{N_c}{N}.$$

Pour des attributs discrets, en notant $n_i(a, c)$ le nombre d'instances de classe c telles que $A_i = a$, et $|V_i|$ le nombre de modalités possibles de A_i, on applique la correction zéro-fréquence ($+1$) comme dans [1] :

$$\widehat{P}(A_i = a \mid c) = \frac{n_i(a, c) + 1}{N_c + |V_i|}.$$

2.2 Approche MapReduce & Traduction Spark

Approche MapReduce (papier). Le papier implémente Naïve Bayes en deux étapes principales sur Hadoop (prétraitement puis comptage), avec des *mappers* et *reducers* explicites en Java.

- **Prétraitement (discrétisation).** Quand des attributs continus existent (p. ex. *adult*, *cmc*), un *job MapReduce* dédié calcule pour chaque attribut son min/max (mapper: *(nom_ attribut, valeur)* → reducer: *(nom_ attribut_ min/max, min/max)*), puis applique un *binning* avant l'entraînement. Les jeux purement catégoriels n'en ont pas besoin.
- **Comptage MapReduce.** Chaque ligne est transformée en paires clé/valeur dont la clé encode *(attribut A_i = valeur a / classe c_j)*, de la forme A_i _ a _ c_j, et la valeur vaut 1; le *reduce* agrège par somme pour obtenir $n_i(a, c_j)$. Les *priors* N_j et N se déduisent par itération/agrégation simples (N_j occurrences de c_j, N total).
- **Paramétrage & décision.** À partir des comptages, le papier pose $P(c_j) = N_j/N$ et $P(A_i = a \mid c_j) = n_i(a, c_j)/N_j$; pour éviter la *zéro-fréquence*, ils ajoutent $+1$ aux comptes (Laplace) et recommandent d'employer les *log-vraisemblances* à l'inférence.

Traduction Spark (RDD DataFrame). Nous reproduisons la logique *clé* → *comptage* → *probas (log)* avec des primitives Spark, en deux variantes.

- **Version RDD.**

1. *Comptages.* Mapper chaque observation en `((i, a, c), 1)` et agréger par `reduceByKey` pour obtenir $n_i(a, c)$; compter aussi `(c,1)` $\rightarrow N_c$ et, si besoin, `(i,a,1)` pour $|\mathcal{V}_i|$.
2. *Paramètres (log).* Calculer $\log \widehat{P}(c) = \log \frac{N_c+1}{N+C}$ et, avec lissage de Laplace, $\log \widehat{P}(A_i = a \mid c) = \log \frac{n_i(a,c)+1}{N_c+|\mathcal{V}_i|}$; diffuser (*broadcast*) ces tables si elles tiennent en mémoire.
3. *Prédiction.* Pour un x, former les paires (i, x_i), joindre avec la table des log-vraisemblances, sommer par classe et ajouter $\log \widehat{P}(c)$, puis prendre l'*argmax*. Les valeurs non vues héritent naturellement de la proba lissée $\frac{1}{N_c+|\mathcal{V}_i|}$.

- **Version DataFrame.**

1. *Comptages.* Mise en forme longue (*unpivot*/`explode`) puis `groupBy(i, a, c).count()` $\rightarrow n_i(a, c)$; `groupBy(c).count()` $\rightarrow N_c$; `groupBy(i, a).count()` pour $|\mathcal{V}_i|$.
2. *Paramètres (log).* `join`s pour enrichir $n_i(a, c)$ avec N_c et $|\mathcal{V}_i|$, colonnes calculées pour $\log \widehat{P}(c)$ et $\log \widehat{P}(A_i = a \mid c)$.
3. *Prédiction.* Jointure de la table des attributs (i, x_i) avec les log-vraisemblances, agrégations par `groupBy(c).sum("loglik")`, ajout de $\log \widehat{P}(c)$, puis `argmax`.

Choix d'implémentation.

- **Discrétisation.** Le papier lance un *job MapReduce* séparé pour min/max et le *binning*. En Spark, on s'appuie sur des opérations `DataFrame` (p.,ex. calculs d'étendue/quantiles) ou on choisit un jeu catégoriel et on *n'applique pas* de binning si inutile, comme noté dans le papier.

- **Format intermédiaire.** Le papier stocke des *SequenceFiles* entre jobs. En Spark, les comptages et probabilités sont tenus en mémoire/disque via le moteur Catalyst et les `RDD`/`DataFrame` (pas de *SequenceFile* dédié).

- **Lissage.** Le papier illustre la formule $P(A_i = a \mid c) = n_i/N_c$ puis mentionne l'ajout de $+1$ aux comptes pour la *zéro-fréquence*; notre implémentation applique la normalisation complète de Laplace $\frac{n_i+1}{N_c+|\mathcal{V}_i|}$ *et* lisse également le prior via $\widehat{P}(c) = \frac{N_c+1}{N+C}$. Ce lissage évite les probabilités nulles, stabilise les estimations en présence de classes/valeurs rares et empêche qu'un prior tombe à zéro si une classe est absente dans un split.

- **Langage exécution.** Le papier code les *Mapper/Reducer* en Java Hadoop; nous traduisons en primitives Spark (`reduceByKey`, `groupBy().count()`, `join`, `withColumn`) en RDD *et* DataFrame.

2.3 Description des étapes du pipeline

Jeu de données. Nous utilisons le dataset *Poker-Hand (UCI)*, intégralement catégoriel, ce qui évite toute étape de discrétisation. Après chargement, nous séparons aléatoirement en *train* (70 %) et *test* (30 %) via `randomSplit([0.7, 0.3], seed=7)`, puis nous mettons en cache ces deux jeux.

Préparation minimale. Les colonnes d'attributs sont uniformisées en chaînes et renommées au format `f0`, `f1`, ..., `fK`, la cible en `y`. Cette normalisation garantit que les deux implémentations (RDD et DataFrame) travaillent exactement sur le même schéma propre ; voir Annexes A.2 et A.5.

Branche RDD. Le code projette chaque observation en une représentation compacte (classe + liste de paires position/valeur), calcule les statistiques suffisantes nécessaires au modèle, puis dérive les paramètres en log avec lissage (y compris une valeur par défaut pour les paires jamais vues). À l'inférence, chaque ligne de test est scorée en somme de logs et la classe prédite est celle au score maximal (arg max). La fonction d'enchaînement renvoie l'accuracy ainsi que les temps d'entraînement et d'inférence ; voir Annexe A.6.

Branche DataFrame. Les données sont d'abord mises au format *long* (une ligne par position/valeur), ce qui permet de produire, via agrégations et jointures, le même jeu de paramètres que la version RDD. La prédiction se fait entièrement en DataFrame : on associe à chaque (position, valeur) son poids, on agrège par ligne, puis on retient la meilleure classe. Comme pour la branche RDD, nous collectons l'accuracy et les temps ; voir Annexe A.8.

Baseline Spark ML. En parallèle, nous entraînons un `NaiveBayes` de la librairie Spark ML sur la même partition train/test, afin de disposer d'un point de référence (accuracy et temps). Cela permet de situer nos deux implémentations par rapport à une implémentation standard ; voir Annexe A.10.

Mesure et traçabilité. Tous les résultats (accuracy et durées train/inférence) sont journalisés de façon homogène pour les trois variantes (RDD, DataFrame, ML). Les paramètres clés (graine, α, nombre de partitions) sont conservés avec les métriques afin de pouvoir rejouer les expériences à l'identique. Les jeux et structures intermédiaires sont persistés uniquement le temps nécessaire puis libérés.

Comparaison de scalabilité. En fin de pipeline, nous évaluons la sensibilité aux volumes et au parallélisme en faisant varier (a) la taille du train par une fraction $f \in \{0.25, 0.5, 1.0, 2.0\}$ (avec $f = 2.0$ obtenu par sur-échantillonnage avec remise), et (b) le niveau de parallélisme via le nombre de partitions $p \in \{p_0, 2p_0, 4p_0\}$, où $p_0 = \max(\texttt{train\_df.rdd.getNumPartitions()}, 2)$. Pour chaque couple (f, p), nous fixons `spark.sql.shuffle.partitions` à p et nous repartitionnons *train* et *test* en p partitions, puis nous exécutons les branches RDD et DataFrame. Les métriques (`accuracy`, `train_time_s`, `infer_time_s`) sont agrégées dans une table unique et affichées pour comparaison ; voir Annexe A.11.

2.4 Résultats & Scalabilité

Les trois approches sont évaluées sur le même jeu de test, avec les mêmes métriques (accuracy, temps d'entraînement et d'inférence). On lit le tableau 1 selon trois axes : *qualité* (accuracy), *coût d'entraînement* et *latence d'inférence*.

Table 1: Résultats comparatifs (jeu de test) — Naïve Bayes

Approche	**Accuracy**	**Temps d'entraînement (s)**	**Temps d'inférence (s)**
RDD NB	0.4925	4.057	0.811
DataFrame NB	0.4924	9.464	16.036
Spark ML NB	0.5025	1.956	0.928

Les deux implémentations (RDD et DataFrame) obtiennent pratiquement la même accuracy (0,4925 vs 0,4924), ce qui est attendu puisqu'elles estiment le même modèle. La baseline

Spark ML atteint 0,5025, soit $\approx +1$ point de pourcentage par rapport au RDD, écart vraisemblablement lié à des détails d'encodage/régularisation (vectorisation interne, gestion des zéros, conventions de lissage).

En *entraînement*, Spark ML est le plus rapide (1,956 s), environ 2,07× plus vite que RDD (4,057 s), tandis que la version DataFrame est plus lente (9,464 s, $\approx$ 2,33× RDD) du fait des *groupBy/join* et des *shuffle* associés. En *inférence*, RDD est le plus rapide (0,811 s), légèrement devant Spark ML (0,928 s, soit RDD $\approx$ 1,14× plus rapide). L'inférence DataFrame est en revanche plus coûteuse (16,036 s, $\approx$ 20× RDD) car elle repose sur des jointures et agrégations par ligne plutôt que sur des accès direct en mémoire.

Le tableau 2 présente l'étude de *scalabilité* de Naïve Bayes pour nos deux implémentations (RDD et DataFrame).

approach	train_fraction	partitions	n_train	n_test	accuracy	train_time_s	infer_time_s
DataFrame	0.25	2	4474	7527	0.4810681546432842	5.054855108261108	12.388885736465454
DataFrame	0.25	4	4474	7527	0.4810681546432842	5.340859889984131	12.303122997283936
DataFrame	0.25	8	4474	7527	0.4810681546432842	7.696258783340454	13.142271518707275
DataFrame	0.5	2	8813	7527	0.48505380629732964	5.950777769088745	7.100423097610474
DataFrame	0.5	4	8813	7527	0.48505380629732964	4.478518486022949	9.860001564025879
DataFrame	0.5	8	8813	7527	0.48505380629732964	5.106024503707886	10.699915647506714
DataFrame	1.0	2	17483	7527	0.49236083432974626	3.757949113845825	9.732130765914917
DataFrame	1.0	4	17483	7527	0.49236083432974626	5.767150163650513	8.686744213104248
DataFrame	1.0	8	17483	7527	0.49236083432974626	6.86827540397644	11.580023288726807
DataFrame	2.0	2	34845	7527	0.489969443337319	5.005343675613403	11.16652226448059
DataFrame	2.0	4	34845	7527	0.489969443337319	5.356663703918457	10.400352478027344
DataFrame	2.0	8	34845	7527	0.489969443337319	7.978947639465332	12.657812595367432
RDD	0.25	2	4474	7527	0.481201009698419	5.082096099853516	0.9534296989440918
RDD	0.25	4	4474	7527	0.48120100969841188	8.320346117019653	1.1998825073242188
RDD	0.25	8	4474	7527	0.48120100969841906	11.993816614151001	1.8709495067596436
RDD	0.5	2	8813	7527	0.4850538062973302	3.042022943496704	0.7459406852722168
RDD	0.5	4	8813	7527	0.4850538062973294	6.290781497955322	1.0319523811340332
RDD	0.5	8	8813	7527	0.48505380629732964	10.915160894393921	1.7650704383850098
RDD	1.0	2	17483	7527	0.4924936893848816	4.3962647914886475	1.2804343700408936
RDD	1.0	4	17483	7527	0.49249368938488136	5.537881135940552	1.0892233848571777
RDD	1.0	8	17483	7527	0.4924936893848812	11.056878566741943	2.1728031635284424
RDD	2.0	2	34845	7527	0.4901022983924535	5.260909557342529	1.3262312412261963
RDD	2.0	4	34845	7527	0.49010229839245395	7.472438335418701	1.788057565689087
RDD	2.0	8	34845	7527	0.49010229839245395	13.146458387374878	2.080132007598877

Table 2: Comparaison de scalabilité — Naïve Bayes.

La précision reste stable (environ 0.48–0.49) quelle que soit la fraction d'entraînement et le nombre de partitions, signe que l'augmentation de volume n'apporte pas de gain significatif sur ce jeu.

Les temps d'entraînement croissent avec la taille du train : côté RDD, on passe d'environ 2.3–2.9 s à 9.2–13.1 s lorsque `train_fraction` va de 0.25 à 2.0 ; côté DataFrame, on observe une hausse du même ordre ($\sim$ 5–13 s). L'effet du parallélisme est modéré sur ce volume, augmenter `partitions` n'apporte qu'un gain partiel et parfois non monotone, car la coordination des tâches et les déplacements de données finissent par coûter plus cher que le calcul.

En inférence, l'écart est marqué : RDD reste le plus rapide (environ 0.9–2.1 s selon la fraction), tandis que DataFrame demeure nettement plus coûteux ($\sim$ 8–13 s).

2.5 Discussion

2.5.1 Points forts

Naïve Bayes offre d'abord une formulation simple et interprétable : des probabilités a priori et conditionnelles, une décision additive en log où chaque attribut contribue de manière lisible au score d'une classe. Cette transparence facilite le diagnostic des comportements du modèle (attributs dominants, modalités ambiguës) et rend la restitution des résultats claire pour le lecteur. Dans notre cas, l'alignement avec des données entièrement catégorielles (Poker-Hand) évite toute ingénierie lourde (pas de discrétisation), tandis que le lissage et le calcul en log-espace assurent une robustesse pratique face aux modalités rares et aux valeurs jamais vues, sans heuristiques complexes.

Ensuite, le modèle présente une bonne tenue computationnelle et une scalabilité naturelle sur Spark. L'apprentissage se ramène à des comptages/agrégations qui se parallélisent très bien, en un seul passage et sans optimisation itérative coûteuse. Les paramètres tiennent en mémoire et se prêtent à une diffusion (broadcast) pour un scoring rapide, avec une empreinte mémoire faible. Par ailleurs, avoir implémenté la même décomposition avec RDD et DataFrame, plus une baseline Spark ML, crée un cadre de comparaison fidèle entre styles d'exécution et renforce la reproductibilité.

Enfin, ce pipeline constitue une baseline solide et extensible. Sa stabilité et sa vitesse en font un point d'appui pour mesurer l'apport de features dérivées (résumés de combinaisons) ou de procédures de ré-équilibrage. La portabilité des paramètres (et, côté DF, la possibilité de broadcast des tables) permet d'explorer rapidement des variantes d'inférence plus efficientes. En bref, Naïve Bayes offre un socle fiable pour des améliorations ciblées, tout en garantissant des comparaisons à conditions équitables entre implémentations et réglages.

2.5.2 Limites et Perspectives

Jeu très déséquilibré. Le dataset Poker-Hand présente une forte asymétrie des fréquences de classes : la classe majoritaire ("nothing in hand") domine largement, tandis que certaines mains sont extrêmement rares. Dans ce contexte, l'accuracy peut être trompeuse, car un classifieur trivial qui prédit systématiquement la classe majoritaire atteint déjà un score non négligeable. Cela constitue une limite de l'évaluation actuelle centrée sur l'accuracy. Pour pallier cet effet, nous pourrions compléter l'accuracy par une baseline "classe majoritaire" et des métriques macro (macro-F1, précision et rappel macro) afin d'évaluer plus équitablement les classes rares.

Coût d'inférence : RDD vs DataFrame. L'implémentation RDD réalise l'inférence via des accès en mémoire des paramètres (priors/conditionnelles), ce qui est léger ; l'implémentation DataFrame, elle, s'appuie sur des joins (souvent un crossJoin avec les classes suivi de left join et d'agrégations), entraînant des shuffles coûteux et une latence plus élevée — c'est une limite de la version DF "full SQL". Pour atténuer ce surcoût, nous pourrions diffuser (broadcast) les tables de paramètres, et ajuster le parallélisme (repartition, spark.sql.shuffle.partitions) afin de limiter les shuffles.

Lissage. Le lissage de Laplace ($\alpha = 1$) stabilise les estimations pour les modalités rares et évite les probabilités nulles pour les valeurs jamais vues, ce qui est crucial sur un jeu déséquilibré. Toutefois, fixer $\alpha = 1$ a un impact implicite sur le biais/variance : trop faible, les classes rares

restent sous-apprises ; trop fort, on sur-lisse et on écrase les signaux. Il serait intéressant de mener une analyse de sensibilité (p. ex. $\alpha \in 0.5, 1, 2$).

Validation. L'évaluation actuelle repose sur un hold-out 70/30 à seed fixe, plus simple mais moins robuste que la validation croisée 6-folds utilisée dans le papier. Cela limite la stabilité des estimations et ne fournit pas d'intervalles de confiance.

3 k-Nearest Neighbor

3.1 Compréhension de l'algorithme

L'algorithme proposé par Maillo *et al.* [2] repose sur l'utilisation du paradigme *MapReduce* afin de rendre l'approche des k plus proches voisins (k-NN) applicable à de larges volumes de données. L'objectif est de paralléliser le calcul des distances entre instances de test et instances d'entraînement, puis d'agréger les résultats partiels pour obtenir la prédiction finale.

Le workflow s'organise en deux étapes :

1. **Phase Map : calcul des voisins partiels**
   - Le jeu d'entraînement est divisé en plusieurs partitions distribuées sur différents nœuds.
   - Chaque tâche Map reçoit une partition du jeu d'entraînement et l'ensemble des instances de test.
   - Pour chaque instance de test, la tâche calcule la distance avec toutes les instances de sa partition d'entraînement et conserve uniquement les k plus proches voisins locaux.
   - En sortie, le Map émet des paires associant une instance de test à sa liste de k voisins les plus proches issus de cette partition.
2. **Phase Reduce : agrégation et décision finale**
   - Le Reduce regroupe pour chaque instance de test l'ensemble des listes de voisins partiels produites par les Mappers.
   - Ces listes sont fusionnées et triées afin de sélectionner les k voisins globaux les plus proches sur l'ensemble du jeu d'entraînement.
   - Enfin, la classe prédite pour l'instance de test est déterminée en fonction de la majorité parmi ces k voisins.

Ce processus garantit que l'algorithme k-NN est appliqué sur l'intégralité des données tout en profitant du parallélisme offert par MapReduce. La phase Map permet de distribuer le calcul lourd des distances, tandis que la phase Reduce assure la consolidation des résultats partiels en une décision de classification cohérente.

3.2 Traduction Spark

- **Version RDD.**
  1. *Phase Map.* Chaque partition du jeu d'entraînement reçoit l'ensemble des points de test (*broadcast*); pour chaque test, calcul des distances avec les points de la partition et conservation des k plus proches voisins locaux.

2. *Phase Reduce.* Agrégation par identifiant de test (`groupByKey`), fusion des listes locales et sélection des k voisins globaux.
3. *Prédiction.* Vote majoritaire sur les k voisins globaux pour attribuer la classe finale à chaque instance de test.

- **Version DataFrame.**
  1. *Phase Map.* Produit de cartes complet (`crossJoin`) entre test et train; calcul des distances via UDF; dans chaque partition physique (`spark_partition_id`), on garde les k voisins locaux par fenêtre ordonnée.
  2. *Phase Reduce.* Fusion des résultats : nouvelle fenêtre ordonnée par distance pour chaque test, sélection des k voisins globaux.
  3. *Prédiction.* Comptage des étiquettes par fenêtre, sélection de la majorité et comparaison à la vérité terrain pour évaluer l'accuracy.

3.3 Description du pipeline k-NN.

Le code implémente un pipeline complet de k-nearest neighbors (k-NN) sur Spark, comparant trois approches : RDD, DataFrame et Spark ML, avec analyse de scalabilité et visualisation des performances.

- **Chargement et préparation des données.** Les datasets d'entraînement et de test (*Poker-Hand*) sont chargés en Spark, limités à un nombre donné d'exemples. Les colonnes des features sont renommées `f0-f9`, et la colonne cible est `label`. Pour les DataFrames et Spark ML, les features sont combinées en un vecteur unique via `array` ou `VectorAssembler` ; voir Annexe B.1.
- **Implémentation RDD.** Les points de test sont diffusés (`broadcast`) vers chaque partition. Chaque partition du train calcule les distances euclidiennes vers tous les points de test (*Map*), conserve les k voisins locaux, puis les fusionne (*Reduce*) pour obtenir les k voisins globaux. La prédiction finale s'effectue par vote majoritaire. Les performances sont mesurées en termes d'accuracy et de temps d'exécution ; voir Annexe B.2.
- **Implémentation DataFrame.** Les distances sont calculées via un `crossJoin` entre test et train, puis transformées en fenêtres locales et globales pour sélectionner les k plus proches voisins. Le vote majoritaire est appliqué avec des agrégations sur fenêtres. Cette approche est plus concise et exploite les optimisations du moteur Spark Catalyst, mais masque le contrôle explicite du partitionnement ; voir Annexe B.3.
- **Scalabilité.** Le pipeline est testé sur différentes tailles d'entraînement (500 à 15 000 exemples), en mesurant la variation de l'accuracy et du temps. Cette étape permet d'évaluer la robustesse et la montée en charge des approches RDD et DataFrame ; voir Annexe B.4.
- **Spark ML (LSH).** Une version approximative du k-NN est implémentée via `Bucketed Random Projection LSH`, qui projette les features dans des buckets pour réduire le coût des comparaisons. Les k voisins les plus proches sont extraits (*Map*) et un vote majoritaire (*Reduce*) produit la prédiction finale. Les effets de k et de `bucketLength` sur l'accuracy et le temps sont étudiés ; voir Annexe B.5.
- **Visualisation.** Les résultats sont collectés dans un `DataFrame` Pandas et visualisés avec `matplotlib` et `seaborn`. Quatre types de graphiques illustrent l'impact de k, du nombre de partitions et de la taille d'entraînement sur l'accuracy et le temps d'exécution pour chaque version ; voir Annexe B.6.

Résumé. Le pipeline permet de comparer la fidélité au paradigme MapReduce (RDD), la concision et l'optimisation (DataFrame), ainsi que la rapidité et l'approximativité via Spark ML. L'ensemble fournit un cadre reproductible pour analyser précision, performance et scalabilité d'un k-NN distribué sur Spark.

3.4 Résultats

Comparaison RDD vs DataFrame. Le Tableau 3 présente les résultats obtenus sur le dataset *Poker-Hand* pour différentes valeurs de $k \in \{3, 5, 7\}$ et pour un degré de parallélisme variant entre 4, 6 et 8 partitions. Nous reportons l'accuracy moyenne et le temps d'exécution en secondes pour les deux implémentations (`RDD` et `DataFrame`).

Version	k	Partitions	Accuracy	Temps (s)
RDD	3	4	0.5350	223.39
RDD	3	6	0.5390	217.03
RDD	3	8	0.5385	218.24
DataFrame	3	4	0.5380	239.51
DataFrame	3	6	0.5380	232.29
DataFrame	3	8	0.5380	229.35
RDD	5	4	0.5435	216.74
RDD	5	6	0.5380	216.72
RDD	5	8	0.5410	218.02
DataFrame	5	4	0.5420	266.28
DataFrame	5	6	0.5420	249.61
DataFrame	5	8	0.5420	261.07
RDD	7	4	0.5540	220.74
RDD	7	6	0.5545	217.27
RDD	7	8	0.5475	225.21
DataFrame	7	4	0.5520	249.96
DataFrame	7	6	0.5520	246.24
DataFrame	7	8	0.5520	247.52

Table 3: Résultats comparatifs RDD vs DataFrame

Analyse. Les résultats montrent une légère amélioration de l'accuracy avec l'augmentation de k, passant d'environ 0.535 pour $k = 3$ à 0.552 pour $k = 7$. Cette progression illustre le compromis classique du k-NN : un voisinage plus large réduit la sensibilité au bruit, au prix d'une perte éventuelle de finesse locale.

La comparaison des deux implémentations met en évidence plusieurs points :

- **Accuracy.** Les performances des deux approches sont quasi identiques (écart < 0.01), ce qui valide la cohérence méthodologique des deux pipelines.
- **Temps d'exécution.** La version `RDD` s'avère légèrement plus rapide (220 s) que la version `DataFrame` (240–260 s). Cet écart s'explique sûrement par le coût supplémentaire des jointures et des optimisations internes liées à l'API SQL.

- **Parallélisme.** La variation du nombre de partitions (4 à 8) n'a qu'un impact marginal sur les temps de calcul (écarts $< 5\,\%$), suggérant que le facteur limitant principal est la complexité quadratique des jointures, davantage que le degré de parallélisation.

La version `RDD` se distingue donc par une meilleure efficacité en temps tout en maintenant une qualité de prédiction équivalente, tandis que la version `DataFrame` sacrifie légèrement la performance au profit d'une intégration plus fluide dans l'écosystème Spark SQL.

Clarté et flexibilité RDD vs DataFrame. Les deux implémentations reproduisent explicitement la logique *Map/Reduce*, mais diffèrent par leur niveau d'abstraction :

- **RDD.** Chaque partition est traitée indépendamment pour calculer les distances (*Map*), avant d'être agrégée pour déterminer les k voisins globaux (*Reduce*). Cette approche est la **plus fidèle au paradigme MapReduce**, les transformations et agrégations étant exprimées de manière explicite et transparente.
- **DataFrame.** L'implémentation repose sur un enchaînement d'opérations déclaratives (cross-join, fenêtres analytiques, agrégations). Bien que l'esprit *Map/Reduce* soit préservé, la logique est masquée derrière l'optimiseur Catalyst. L'approche gagne en lisibilité et intégration avec Spark SQL, mais au prix d'une moindre visibilité sur le parallélisme et le partitionnement.

La version `RDD` offre donc un contrôle explicite et reste fidèle au modèle MapReduce, tandis que la version `DataFrame` est plus lisible.

Analyse de la scalabilité RDD vs DataFrame. Les résultats présentés dans le Tableau 4 illustrent l'évolution de l'accuracy et du temps d'exécution en fonction de la taille du jeu d'entraînement pour les implémentations k-NN en RDD et DataFrame.

Version	TrainSize	Accuracy	Time (s)
RDD	500	0.450	4.82
DataFrame	500	0.410	4.44
RDD	2000	0.520	11.30
DataFrame	2000	0.513	10.68
RDD	5000	0.533	56.32
DataFrame	5000	0.544	50.57
RDD	10000	0.553	219.17
DataFrame	10000	0.540	216.19
RDD	15000	0.550	488.37
DataFrame	15000	0.545	507.71

Table 4: Comparaison de scalabilité des implémentations RDD et DataFrame

Interprétation.

- **Accuracy.** La version RDD devance systématiquement la version DataFrame pour toutes les tailles sauf $n = 5000$, où la DataFrame obtient un léger avantage. Cette tendance montre que le contrôle explicite du flux de données dans RDD permet parfois de mieux conserver la précision des k voisins ; voir Annexe 1.

- **Temps d'exécution.** Les temps croissent fortement avec la taille du jeu d'entraînement, reflétant la complexité quadratique du k-NN. Pour de petites tailles ($n \leq 10000$), l'approche DataFrame est légèrement plus rapide ; voir Annexe 2.
- **Scalabilité relative.** L'approche DataFrame semble offrir une meilleure gestion automatique des plans d'exécution et des jointures, donnant des temps plus réguliers pour des tailles intermédiaires. L'approche RDD conserve un contrôle fin sur le partitionnement et le flux de données, ce qui favorise la précision.

Conclusion. Pour des jeux de données petits à moyens, l'approche DataFrame assure un temps d'exécution stable avec une précision équivalente ou légèrement inférieure. Cependant, pour des jeux volumineux, la version RDD semble préférable, dans la mesure où elle offre une meilleure précision et un contrôle plus explicite sur le parallélisme.

Analyse des résultats Spark ML. Le tableau 5 présente les performances d'un k-NN approximatif implémenté via `BucketedRandomProjectionLSH` dans Spark ML, en fonction du nombre de voisins k et du paramètre `bucketLength`.

k	bucketLength	Accuracy	Time (s)
3	1.0	0.5270	171.1
3	2.0	0.5295	280.7
3	3.0	0.5285	384.3
5	1.0	0.5220	130.9
5	2.0	0.5370	248.1
5	3.0	0.5365	277.5
7	1.0	0.5295	118.7
7	2.0	0.5450	224.4
7	3.0	0.5510	299.3

Table 5: Performances du k-NN Spark ML.

Interprétation.

- **Impact de k.** L'accuracy croît systématiquement avec k, avec $bucketLength = 2$ par exemple, elle passe de 0.5295 pour $k = 3$ à 0.5450 pour $k = 7$, ce qui reflète le compromis classique du k-NN : un plus grand nombre de voisins améliore la robustesse au bruit.
- **Impact de bucketLength.** Des buckets plus larges (`bucketLength`=3) conduisent à des temps d'exécution plus longs mais offrent une précision légèrement meilleure. Les plus grands buckets favorisent un meilleur regroupement approximatif des voisins, mais augmentent en contrepartie le nombre de comparaisons internes nécessaires, et donc le temps d'exécution.
- **Temps d'exécution.** Le coût croît significativement avec bucketLength et, dans une moindre mesure, avec k. Les temps passent de 120 s pour $k = 7$, bucketLength=1 à 380 s pour $k = 3$, bucketLength=3, illustrant le compromis entre précision et rapidité.

Conclusion. La version Spark ML fournit un k-NN approximatif efficace, où le paramètre `bucketLength` permet de moduler la précision et le temps d'exécution. Le choix de k et de la longueur des buckets doit être effectué en fonction du compromis souhaité entre performance et rapidité. Pour les expérimentations avec un budget temps limité, des buckets plus petits et un k modéré sont recommandés, tandis que pour maximiser l'accuracy, un k plus élevé et des buckets plus larges semblent préférables.

3.5 Limites et Perspectives

Analyse comparative des 3 approches. Le tableau 6 met en évidence les performances relatives des trois approches pour le k-NN sur un extrait du dataset Poker-Hand ($n = 10k$).

Version	Paramètres	Accuracy	Temps (s)	Remarques
RDD	$k = 7$, partitions=6	0.5545	217.3	Meilleure précision globale, contrôle explicite des partitions
DataFrame	$k = 7$, partitions=6	0.5520	246.2	Plus lisible, légèrement plus lent, fidélité au MapReduce moins explicite
Spark ML	$k = 7$, bucketLength=3	0.5510	299.3	Méthode approx. via LSH, temps plus long, pipeline optimisé automatiquement

Table 6: Comparaison des meilleures performances obtenues

- **Précision.** La version RDD atteint la meilleure accuracy (0.5545), légèrement supérieure à celle de DataFrame (0.5520) et Spark ML (0.5510). Cela confirme que le contrôle explicite sur les partitions et le flux de données favorise une sélection plus précise des voisins.
- **Temps d'exécution.** Le pipeline RDD est aussi le plus rapide (217 s), suivi par DataFrame (246 s) et Spark ML (299 s). Le coût supplémentaire de Spark ML semble provenir de la phase LSH et de l'approximation des distances, tandis que la version DataFrame semble payer la surcharge des jointures et des fenêtres.
- **Trade-off.** Spark ML offre un pipeline plus concis et optimisé automatiquement mais avec un temps d'exécution plus élevé, tandis que RDD privilégie la fidélité au modèle MapReduce et un contrôle fin du parallélisme. DataFrame constitue un compromis : code plus lisible et maintenable, mais avec une légère perte de performance en précision et temps.

Interprétation. La version RDD offre la meilleure précision et un temps d'exécution réduit grâce à un contrôle explicite du flux de données et du partitionnement. La version DataFrame constitue un compromis efficace entre lisibilité, maintenabilité et performance. Spark ML, bien que plus abstraite et optimisée automatiquement, présente un temps d'exécution plus élevé pour une précision similaire.

Conclusion. Pour des besoins de contrôle et d'optimisation fine, la version RDD semble préférable ; pour la lisibilité et l'intégration avec Spark SQL, DataFrame et Spark ML sont des alternatives valides.

4 Conclusion générale.

L'analyse des implémentations RDD, DataFrame et Spark ML pour Naïve Bayes et k-NN révèle un compromis clair entre contrôle et abstraction : les RDD permettent un suivi précis du flux de données et offrent en général la meilleure précision, notamment pour k-NN, tandis que les DataFrames rendent le code plus concis et profitent des optimisations automatiques du moteur Spark au prix de surcoûts liés aux jointures et agrégations. Spark ML se distingue par un entraînement rapide et reproductible, particulièrement efficace pour Naïve Bayes. Ainsi, RDD semble à privilégier pour l'analyse fine du parallélisme, DataFrame pour un usage industriel flexible et maintenable, et Spark ML pour des pipelines standards performants et facilement déployables.

References

[1] Songtao Zheng. "Naıve Bayes Classifier: A MapReduce Approach". Major Department: Computer Science. Master's thesis. Fargo, North Dakota: North Dakota State University, Oct. 2014.

[2] Jesús Maillo, Isaac Triguero, and Francisco Herrera. "A mapreduce-based k-nearest neighbor approach for big data classification". In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 2. IEEE. 2015, pp. 167–172.

A Code source (Naïve Bayes)

A.1 Imports & session

```
# --- Imports & session ------------------------------------------------------
from pyspark.sql import SparkSession, functions as F, types as T
from pyspark.ml import Pipeline
from pyspark.ml.feature import OneHotEncoder, VectorAssembler, StringIndexer
from pyspark.ml.classification import NaiveBayes
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

import math, time, os, shutil, random
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from pyspark.sql.window import Window

# --- Session Spark ----------------------------------------------------------
spark = (
    SparkSession.builder
    .appName("NaiveBayes_MapReduce_like")
    .getOrCreate()
)

# Réduit le bruit des logs Spark pour ne garder que les avertissements et erreurs.
spark.sparkContext.setLogLevel("WARN")

# --- Hyperparam / config par défaut -----------------------------------------
ALPHA = 1.0       # Lissage de Laplace appliqué aux proba conditionnelles
seed = 7          # Graine pour la reproductibilité
train_ratio = 0.7 # Protocole : 70% train / 30% test
has_header = False
sep = ","

# --- Données & sorties intermédiaires ---------------------------------------
data_path = "poker-hand-training-true.data"  # Jeu UCI Poker-Hand
out_dir = "/tmp/nb_counts"                   # Dossier pour stocker les comptages «
MapReduce-like »

# Remise à zéro des sorties intermédiaires
if os.path.exists(out_dir):
    shutil.rmtree(out_dir)
os.makedirs(out_dir, exist_ok=True)
```

A.2 Préparation des colonnes

```
def prepare_columns(df, has_header: bool):
    """
    Prépare un DataFrame pour l'entraînement :
    - la DERNI RE colonne est la colonne cible (label),
    - renomme proprement les colonnes si le fichier n'a pas d'en-tête,
      afin d'obtenir: features = f0, f1, ..., f{n-1} et label = "label".

    Paramètres
    ----------
    df : pyspark.sql.DataFrame
        Données d'entrée (colonnes ordonnées: features puis label en dernier).
    has_header : bool
        True si le fichier source contient un en-tête déjà exploitable.

    Retour
    ------
    (df, feature_cols, label_col) : tuple
        - df : DataFrame éventuellement renommé,
        - feature_cols : liste des noms de colonnes de caractéristiques,
        - label_col : nom de la colonne cible.
    """
    cols = df.columns  # ordre des colonnes tel que lu

    if has_header:
        # Cas "avec en-tête" : on garde les noms tels quels.
        # Hypothèse: la dernière colonne du fichier est bien le label.
        label_col = cols[-1]
        feature_cols = cols[:-1]
    else:
        # Cas "sans en-tête" : la dernière colonne = label, le reste = features
        label_col = cols[-1]
        feature_cols = cols[:-1]

        # Renommer les features en f0, f1, ..., et le label en "label"
        # -> rend le pipeline plus simple et déterministe
        ren = (
            [F.col(c).alias(f"f{i}") for i, c in enumerate(feature_cols)]
            + [F.col(label_col).alias("label")]
        )
        df = df.select(*ren)

        # Recrée les listes de noms après renommage
        feature_cols = [f"f{i}" for i in range(len(feature_cols))]
        label_col = "label"

    return df, feature_cols, label_col
```

A.3 Chargement du CSV et typage

```
# --- Lecture brute ------------------------------------------------------------
raw = (
    spark.read
    .option("header", has_header)   # True si le CSV a une ligne  d en -tête
    .option("inferSchema", True)    # Infère automatiquement les types
    .option("sep", sep)             # Séparateur de colonnes (",")
    .csv(data_path)                 # Fichier source
)

# --- Préparation des noms de colonnes (et label en dernière position) -------
df, feature_cols, label_col = prepare_columns(raw, has_header)

# --- Mise au bon type pour Spark ML -----------------------------------------
# Les algos/transformers Spark ML attendent des types numériques (DoubleType).
# On caste donc toutes les colonnes (features + label) en double.
for c in feature_cols + [label_col]:
    df = df.withColumn(c, F.col(c).cast("double"))

# --- Cache pour accélérer les étapes suivantes ------------------------------
# On met en cache car le DataFrame sera réutilisé plusieurs fois (splits, comptages, etc.).
df.cache()

# On vérifie visuellement le schéma/échantillon
df.show(5)

# Traçabilité : liste des features et nom du label
print("Features:", feature_cols)
print("Label␣␣␣:", label_col)
```

A.4 Split entraînement/test

```
# --- Split entraînement / test ----------------------------------------------
# randomSplit sépare aléatoirement selon des poids (approx.)
train_df, test_df = df.randomSplit([train_ratio, 1 - train_ratio], seed=seed)

# Ces DataFrames seront réutilisés (comptages, entraînement, évaluation) :
# on les met en cache pour éviter de relire/calculer plusieurs fois.
train_df.cache()
test_df.cache()

# count() force  l  évaluation paresseuse (lazy) de Spark, matérialise le cache
# et affiche les tailles effectives des splits.
print("Train␣=", train_df.count(), "␣/␣Test␣=", test_df.count())
```

A.5 Normalisation & format long

```
# 1) Détection des colonnes de features f0..fN
feature_cols = sorted(
    [c for c in train_df.columns if c.startswith("f")],
    key=lambda x: int(x[1:]) if x[1:].isdigit() else 10**9
)
assert feature_cols, "Aucune colonne 'f*' détectée dans train_df."

# 2) Normalisation (Naïve Bayes discret)
def normalize_df(df):
    """Cast features/label en string; renomme label -> 'y'."""
    cols = [F.col(c).cast("string").alias(c) for c in feature_cols]
    return df.select(*cols, F.col("label").cast("string").alias("y"))

# 3) Passage wide -> long
def to_long(df, keep_cols):
    """Wide     long: une ligne par (pos, value); conserve keep_cols."""
    arr = F.array(*[F.col(c) for c in feature_cols])
    return df.select(*keep_cols, F.posexplode(arr).alias("pos", "value"))

# 4) Redimensionnement du jeu  d entra înement (pour la scalabilité)
def scale_dataset(df, factor):
    """Resize: <1 sous-échant.; =1 inchangé; >1 sur-échant. avec remise."""
    if factor == 1.0:
        return df
    return df.sample(withReplacement=(factor > 1.0), fraction=factor, seed=42)

# 5) Datasets normalisés
train_norm = normalize_df(train_df).cache(); train_norm.count()
test_norm  = normalize_df(test_df ).cache(); test_norm.count()
```

A.6 Naïve Bayes RDD

```
# === Naive Bayes RDD (discret) avec lissage de Laplace ALPHA=1.0 =============

def rdd_train(train_df_norm):
    """Construit le modèle: logprior, logcond, valeur par défaut, classes, |V_i|."""
    # RDD de (y, [(pos, value), ...]) pour compter facilement par (i,v,c)
    def row_to_pair(row):
        feats = [(i, getattr(row, feature_cols[i])) for i in range(len(feature_cols))]
        return (row['y'], feats)

    rdd = train_df_norm.rdd.map(row_to_pair).cache()

    # Priors N_c puis P(c) lissé
    N_c = rdd.map(lambda yc: (yc[0], 1)).reduceByKey(lambda a, b: a + b)
    N = N_c.map(lambda kv: kv[1]).sum()
    classes = N_c.map(lambda kv: kv[0]).collect()
    C = len(classes)
    logprior = N_c.mapValues(lambda nc: math.log((nc + ALPHA) / (N + ALPHA *
    C))).collectAsMap()

    # Comptes N_{i,v,c}
    ivc = (
        rdd.flatMap(lambda yc: [((i, v, yc[0]), 1) for (i, v) in yc[1]])
           .reduceByKey(lambda a, b: a + b)
    )
```

```
    # Dénominateurs N_{i,*,c} pour chaque (i,c)
    ic = (
        ivc.map(lambda kv: ((kv[0][0], kv[0][2]), kv[1]))
           .reduceByKey(lambda a, b: a + b)
           .collectAsMap()
    )

    # Tailles de vocabulaires |V_i|
    Vi = (
        rdd.flatMap(lambda yc: [((i, v), 1) for (i, v) in yc[1]])
           .distinct()
           .map(lambda kv: (kv[0][0], 1))
           .reduceByKey(lambda a, b: a + b)
           .collectAsMap()
    )

    # log P(v|c,i) lissé
    def to_logcond(rec):
        (i, v, c), n = rec
        denom = ic[(i, c)]
        V = Vi[i]
        return ((i, v, c), math.log((n + ALPHA) / (denom + ALPHA * V)))

    logcond = ivc.map(to_logcond).collectAsMap()

    # Valeur par défaut pour v non vu: log(   / (N_{i,*,c} +   |V_i|))
    default_logcond = {
        (i, c): math.log(ALPHA / (ic[(i, c)] + ALPHA * Vi[i]))
        for (i, c) in ic.keys()
    }

    return {
        "logprior": logprior,
        "logcond": logcond,
        "default": default_logcond,
        "classes": classes,
        "Vi": Vi,
    }

def rdd_predict(test_df_norm, model):
    """Retourne un RDD de (pred, y_true) en scorant en log-espace."""
    logprior = model["logprior"]; logcond = model["logcond"]; default = model["default"]
    classes = model["classes"]

    def score_row(row):
        feats = [(i, getattr(row, feature_cols[i])) for i in range(len(feature_cols))]
        best_c, best_s = None, -1e100
        for c in classes:
            s = logprior[c]
            for (i, v) in feats:
                s += logcond.get((i, v, c), default[(i, c)])
            if s > best_s:
                best_s, best_c = s, c
        return (best_c, row['y'])

    return test_df_norm.rdd.map(score_row)

def rdd_run(train_df_norm, test_df_norm):
    """Entraîne, infère, renvoie (accuracy, train_time, infer_time)."""
    t0 = time.time()
    model = rdd_train(train_df_norm)
    train_time = time.time() - t0
```

```
    t1 = time.time()
    preds = rdd_predict(test_df_norm, model).cache()
    _ = preds.count()  # force l'exécution pour mesurer l'inférence
    infer_time = time.time() - t1

    accuracy = preds.map(lambda pr: 1 if pr[0] == pr[1] else 0).mean()
    return float(accuracy), train_time, infer_time
```

A.7 Exécution métriques - RDD

```
# Exécute le pipeline RDD NB et affiche les métriques principales
acc, ttrain, tinfer = rdd_run(train_norm, test_norm)
print(f"[RDD NB] accuracy={acc:.4f} | train_time={ttrain:.3f}s | infer_time={tinfer:.3f}s")
```

A.8 Naïve Bayes DataFrame

```
# === Naive Bayes DataFrame (discret), Laplace ALPHA=1.0 ======================

def df_train(train_df_norm):
    """Calcule prior/logprior, logcond et valeurs par défaut en format DF."""
    # Format long: une ligne par (pos, value) observé
    train_long = to_long(train_df_norm, keep_cols=["y"]).cache(); train_long.count()

    # Priors P(c) lissés
    prior = train_long.groupBy("y").count().cache(); prior.count()
    N = prior.agg(F.sum("count").alias("N")).first()["N"]
    C = prior.count()
    prior_log = (
        prior.withColumn(
            "logprior",
            F.log((F.col("count") + F.lit(ALPHA)) / (F.lit(N) + F.lit(ALPHA * C)))
        ).select("y", "logprior")
    )

    # Comptes conditionnels
    ivc = (
        train_long.groupBy("pos", "value", "y").count()
                  .withColumnRenamed("count", "n_ivc")
                  .cache()
    ); ivc.count()

    ic = ivc.groupBy("pos", "y").agg(F.sum("n_ivc").alias("n_i_star_c")).cache(); ic.count()
    Vi = train_long.groupBy("pos").agg(F.countDistinct("value").alias("V")).cache(); Vi.count()

    # log P(v|c,i) lissé
    cond = (
        ivc.join(ic, ["pos", "y"])
           .join(Vi, ["pos"])
           .withColumn(
               "logcond",
               F.log((F.col("n_ivc") + F.lit(ALPHA)) /
                     (F.col("n_i_star_c") + F.lit(ALPHA) * F.col("V")))
           )
           .select("pos", "value", "y", "logcond")
           .cache()
    ); cond.count()
```

```
    # Valeur par défaut pour v non vu
    default = (
        ic.join(Vi, ["pos"])
          .withColumn(
              "default_logcond",
              F.log(F.lit(ALPHA) / (F.col("n_i_star_c") + F.lit(ALPHA) * F.col("V")))
          )
          .select("pos", "y", "default_logcond")
          .cache()
    ); default.count()

    return {"prior_log": prior_log, "cond": cond, "default": default}


def df_predict(test_df_norm, model):
    """Score en log-espace + argmax par id; retourne (accuracy, preds DF)."""
    # Id unique + label vrai
    test_rows = (
        test_df_norm.withColumn("id", F.monotonically_increasing_id())
                    .withColumnRenamed("y", "y_true")
                    .cache()
    ); test_rows.count()

    # Format long
    test_long = to_long(test_rows, keep_cols=["id", "y_true"]).cache(); test_long.count()

    # Espace des classes
    classes = model["prior_log"].select(F.col("y").alias("y")).cache(); classes.count()

    # Étendre (id,pos,value) à toutes les classes
    base = (
        test_long.select("id", "pos", "value").dropDuplicates(["id", "pos", "value"])
                 .crossJoin(classes)
                 .cache()
    ); base.count()

    # Joins: logcond si observé, sinon default_logcond -> somme + prior
    joined = (
        base.join(model["cond"], ["pos", "value", "y"], "left")
            .join(model["default"], ["pos", "y"], "left")
            .withColumn("logc", F.coalesce(F.col("logcond"), F.col("default_logcond")))
            .groupBy("id", "y").agg(F.sum("logc").alias("sum_logc"))
            .join(model["prior_log"], ["y"])
            .withColumn("score", F.col("sum_logc") + F.col("logprior"))
            .cache()
    ); joined.count()

    # Argmax(y) par id
    w = Window.partitionBy("id").orderBy(F.col("score").desc())
    pred = (
        joined.withColumn("rn", F.row_number().over(w))
              .where("rn = 1")
              .select("id", F.col("y").alias("pred"))
              .cache()
    ); pred.count()

    # Accuracy
    acc = (
        pred.join(test_rows.select("id", "y_true"), "id")
            .withColumn("ok", (F.col("pred") == F.col("y_true")).cast("int"))
            .agg(F.avg("ok")).first()[0]
```

```
    )

    return float(acc), pred


def df_run(train_df_norm, test_df_norm):
    """Entraîne DF NB puis infère; renvoie (acc, train_time, infer_time)."""
    t0 = time.time()
    model = df_train(train_df_norm)
    # Matérialiser le "modèle" DF
    model["cond"].count(); model["default"].count(); model["prior_log"].count()
    train_time = time.time() - t0

    t1 = time.time()
    acc, pred = df_predict(test_df_norm, model)
    pred.count()  # force  l ex écution pour mesurer  l inf érence
    infer_time = time.time() - t1

    return acc, train_time, infer_time
```

A.9 Exécution métriques - DataFrame

```
# Exécute le pipeline DF NB et affiche les métriques principales
acc, ttrain, tinfer = df_run(train_norm, test_norm)
print(f"[DataFrame NB] accuracy={acc:.4f} | train_time={ttrain:.3f}s |
infer_time={tinfer:.3f}s")
```

A.10 Baseline Spark ML NaiveBayes

```
# 1) Vectoriser les f* en une colonne "features" --------------------------------
# Récupère toutes les colonnes de features (f0, f1, ..., fN), triées par index numérique.
feature_cols = sorted(
    [c for c in train_df.columns if c.startswith("f")],
    key=lambda x: int(x[1:]) if x[1:].isdigit() else 10**9
)

# Assemble les colonnes f* en un unique vecteur dense "features".
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")

# Applique  l assembleur  aux splits et ne garde que (features, label).
train_ml = assembler.transform(train_df).select("features", "label")
test_ml  = assembler.transform(test_df ).select("features", "label")

# 2) Entraîner et évaluer le NaiveBayes Spark ML
t0 = time.time()
nb = NaiveBayes(
    modelType="multinomial",  # Adapté aux features non négatives.
    smoothing=1.0,            # Lissage de Laplace (   = 1).
    featuresCol="features",
    labelCol="label",
    predictionCol="prediction"
)
model = nb.fit(train_ml)             # Entraînement
train_time = time.time() - t0        # Temps d'entraînement (s)

# Inférence sur le set de test
t1 = time.time()
```

```
pred = model.transform(test_ml).select("prediction", "label").cache()
pred.count()
infer_time = time.time() - t1        # Temps d'inférence (s)

# Accuracy
acc = MulticlassClassificationEvaluator(
    labelCol="label",
    predictionCol="prediction",
    metricName="accuracy"
).evaluate(pred)

print(f"[Spark␣ML␣NB]␣accuracy={acc:.4f}␣|␣train_time={train_time:.3f}s␣|␣
infer_time={infer_time:.3f}s")
```

A.11 Expérience Scalabilité

```
# === Grille d'expériences ============================================

fractions = [0.25, 0.5, 1.0, 2.0]   # redimensionne l'ensemble d'entraînement : 25%, 50%,
100%, 200% (200% = échantillonnage avec remise)
base_parts = max(train_df.rdd.getNumPartitions(), 2)
parts_grid = [base_parts, base_parts * 2, base_parts * 4]   # différents niveaux de
parallélisme (nb de partitions / shuffles)

rows = []
for fr in fractions:
    tr = scale_dataset(train_norm, fr)  # redimensionne l'entraînement (<=1 sous-échant., >1
    sur-échant.)
    for p in parts_grid:
        # Régler le parallélisme (nombre de partitions dans les shuffles)
        spark.conf.set("spark.sql.shuffle.partitions", p)

        # Harmoniser le partitionnement des datasets et matérialiser
        tr_p = tr.repartition(p).cache();  tr_p.count()
        te_p = test_norm.repartition(p).cache(); te_p.count()

        # --- Variante RDD ----
        acc, ttrain, tinfer = rdd_run(tr_p, te_p)
        rows.append(("RDD", fr, p, tr_p.count(), te_p.count(), acc, ttrain, tinfer))

        # --- Variante DataFrame ----
        acc, ttrain, tinfer = df_run(tr_p, te_p)
        rows.append(("DataFrame", fr, p, tr_p.count(), te_p.count(), acc, ttrain, tinfer))

        # Libérer le cache pour l'itération suivante
        tr_p.unpersist(); te_p.unpersist()

# Résultats agrégés (un DF pour affichage)
schema = ["approach","train_fraction","partitions","n_train","n_test",
          "accuracy","train_time_s","infer_time_s"]
res = spark.createDataFrame(rows, schema).orderBy("approach","train_fraction","partitions")
n = res.count()
res.show(n, truncate=False)
```

B Code source (k-NN)

B.1 Chargement et préparation des données.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, array, row_number, count, udf, spark_partition_id
from pyspark.sql.window import Window
import math
import time
import pandas as pd

# ------------------------------
# INITIALISATION SPARK
# ------------------------------
spark = SparkSession.builder.appName("kNN_Maillo_k_variation").getOrCreate()
sc = spark.sparkContext

# ------------------------------
# LOAD TRAIN AND TEST DATASET
# ------------------------------
cols = [f"f{i}" for i in range(10)] + ["label"]
n_train = 10000
n_test = int(0.2 * n_train)

train = spark.read.csv("poker-hand-training-true.data", header=False,
inferSchema=True).toDF(*cols).limit(n_train)
test = spark.read.csv("poker-hand-testing.data", header=False,
inferSchema=True).toDF(*cols).limit(n_test)

print(f"train size: {train.count()}, test size: {test.count()}")
```

B.2 Implémentation RDD.

```
import math
import time

# ============================================================
# VERSION RDD
# ============================================================
test_points = test.collect()
test_broadcast = sc.broadcast(test_points)

def euclidean_distance(p1, p2):
    return math.sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2)))

def map_knn_local(partition):
    """
    Phase Map : chaque partition calcule distances test/train
    et garde les k voisins LOCAUX pour chaque test point.
    """
    partition = list(partition)
    results = {i: [] for i in range(len(test_broadcast.value))}

    for idx, test_point in enumerate(test_broadcast.value):
        test_features = [test_point[f"f{i}"] for i in range(10)]
        local_neighbors = []

        for train_point in partition:
            train_features = [train_point[f"f{i}"] for i in range(10)]
```

```
            dist = euclidean_distance(test_features, train_features)
            local_neighbors.append((dist, train_point["label"]))

        # garder les k voisins locaux
        local_neighbors = sorted(local_neighbors, key=lambda x: x[0])[:k]
        results[idx] = local_neighbors

    for idx, neighbors in results.items():
        yield (idx, neighbors)

def reduce_knn_global(neighbors_lists):
    """
    Phase Reduce : fusionne toutes les listes locales
    et garde les k meilleurs globalement.
    """
    merged = []
    for lst in neighbors_lists:
        merged.extend(lst)
    return sorted(merged, key=lambda x: x[0])[:k]

def majority_vote(neighbors):
    labels = [lbl for _, lbl in neighbors]
    return max(set(labels), key=labels.count)

# --- Pipeline RDD
start_rdd = time.time()

mapped = train.rdd.mapPartitions(map_knn_local)
reduced = mapped.groupByKey().mapValues(reduce_knn_global)
predictions_rdd = reduced.mapValues(majority_vote)

# Évaluation précision
correct_rdd = predictions_rdd.filter(lambda x: test_broadcast.value[x[0]]["label"] ==
x[1]).count()
total_rdd = predictions_rdd.count()
accuracy_rdd = correct_rdd / total_rdd

time_rdd = time.time() - start_rdd
print(f"\nRDD (Maillo-style) => Accuracy: {accuracy_rdd:.4f}, Time: {time_rdd:.2f}s")
```

B.3 Implémentation DataFrame.

```
from pyspark.sql.functions import col, array, row_number, count, udf, spark_partition_id
from pyspark.sql.window import Window
import math
import time

# ============================================================
# VERSION DATAFRAME
# ============================================================
# Ajout features
train = train.withColumn("features", array(*[col(f"f{i}") for i in range(10)]))
test = test.withColumn("features", array(*[col(f"f{i}") for i in range(10)]))

# Distance UDF
def euclidean_distance_udf(f1, f2):
    return float(math.sqrt(sum((a - b) ** 2 for a, b in zip(f1,f2))))
distance_udf = udf(euclidean_distance_udf, "float")

start_df = time.time()
```

```
# Cross join train/test
df_cross = test.crossJoin(train.withColumnRenamed("features","train_features")
                                .withColumnRenamed("label","train_label"))

# Calcul distance
df_cross = df_cross.withColumn("distance", distance_udf(col("features"),
col("train_features")))

# --- MAP phase (gardons k voisins locaux par partition)
window_local = Window.partitionBy("features",
spark_partition_id()).orderBy(col("distance").asc())
df_local_knn = df_cross.withColumn("rn", row_number().over(window_local)).filter(col("rn") <=
k)

# --- REDUCE phase (fusion et garde k globaux)
window_global = Window.partitionBy("features").orderBy(col("distance").asc())
df_global_knn = df_local_knn.withColumn("rn_global",
row_number().over(window_global)).filter(col("rn_global") <= k)

# --- Vote majoritaire
window_vote = Window.partitionBy("features", "train_label")
df_votes = df_global_knn.withColumn("cnt", count("train_label").over(window_vote))

window_max = Window.partitionBy("features").orderBy(col("cnt").desc())
df_pred = df_votes.withColumn("rn_vote", row_number().over(window_max)).filter(col("rn_vote")
== 1)

# --- Évaluation
df_eval = df_pred.withColumn("correct", (col("label") == col("train_label")).cast("integer"))
accuracy_df = df_eval.agg({"correct":"avg"}).collect()[0][0]

time_df = time.time() - start_df
print(f"DataFrame (Maillo-style) => Accuracy: {accuracy_df:.4f}, Time: {time_df:.2f}s")

# --------------------------------
spark.stop()
```

B.4 Scalabilité.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, array, row_number, count, spark_partition_id
from pyspark.sql.window import Window
import math, time
import pandas as pd

# --------------------------------
# INITIALISATION SPARK
# --------------------------------
spark = SparkSession.builder.appName("kNN_Scalability").getOrCreate()
sc = spark.sparkContext

# --------------------------------
# PARAM TRES
# --------------------------------
cols = [f"f{i}" for i in range(10)] + ["label"]
k = 5
num_partitions = 6
sample_sizes = [500, 2000, 5000, 10000, 15000]  # tailles d'échantillons à tester
```

```
results = []

# ------------------------------
# BOUCLE SUR LES TAILLES
# ------------------------------
for n_train in sample_sizes:
    n_test = int(0.2 * n_train)

    print(f"\n=== Test avec n_train={n_train}, n_test={n_test} ===")

    # Charger données
    train = spark.read.csv("poker-hand-training-true.data", header=False, inferSchema=True) \
                    .toDF(*cols).limit(n_train).repartition(num_partitions)
    test = spark.read.csv("poker-hand-testing.data", header=False, inferSchema=True) \
                   .toDF(*cols).limit(n_test).repartition(num_partitions)

    # ------------------------------
    # VERSION RDD
    # ------------------------------
    test_points = test.collect()
    test_broadcast = sc.broadcast(test_points)

    start_rdd = time.time()
    mapped = train.rdd.mapPartitions(map_knn_local)
    reduced = mapped.groupByKey().mapValues(reduce_knn_global)
    predictions_rdd = reduced.mapValues(majority_vote)

    correct_rdd = predictions_rdd.filter(lambda x: test_broadcast.value[x[0]]["label"] ==
    x[1]).count()
    total_rdd = predictions_rdd.count()
    accuracy_rdd = correct_rdd / total_rdd
    time_rdd = time.time() - start_rdd

    results.append(("RDD", n_train, accuracy_rdd, time_rdd))

    # ------------------------------
    # VERSION DATAFRAME
    # ------------------------------
    train = train.withColumn("features", array(*[col(f"f{i}") for i in range(10)]))
    test = test.withColumn("features", array(*[col(f"f{i}") for i in range(10)]))

    start_df = time.time()
    df_cross = test.crossJoin(train.withColumnRenamed("features","train_features")
                                   .withColumnRenamed("label","train_label"))
    df_cross = df_cross.withColumn("distance", distance_udf(col("features"),
    col("train_features")))

    window_local = Window.partitionBy("features",
    spark_partition_id()).orderBy(col("distance").asc())
    df_local_knn = df_cross.withColumn("rn", row_number().over(window_local)).filter(col("rn")
    <= k)

    window_global = Window.partitionBy("features").orderBy(col("distance").asc())
    df_global_knn = df_local_knn.withColumn("rn_global",
    row_number().over(window_global)).filter(col("rn_global") <= k)

    window_vote = Window.partitionBy("features", "train_label")
    df_votes = df_global_knn.withColumn("cnt", count("train_label").over(window_vote))

    window_max = Window.partitionBy("features").orderBy(col("cnt").desc())
    df_pred = df_votes.withColumn("rn_vote",
    row_number().over(window_max)).filter(col("rn_vote") == 1)
```

```
    df_eval = df_pred.withColumn("correct", (col("label") ==
    col("train_label")).cast("integer"))
    accuracy_df = df_eval.agg({"correct":"avg"}).collect()[0][0]
    time_df = time.time() - start_df

    results.append(("DataFrame", n_train, accuracy_df, time_df))

# ------------------------------
# RÉSULTATS FINAUX
# ------------------------------
df_results = pd.DataFrame(results, columns=["Version", "TrainSize", "Accuracy", "Time"])
print("\n=== Résultats comparatifs ===")
print(df_results)

# ------------------------------
spark.stop()
```

B.5 Spark ML.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, array, row_number, count, udf, spark_partition_id
from pyspark.sql.window import Window
import math
import time
import pandas as pd

# ------------------------------
# INITIALISATION SPARK
# ------------------------------
spark = SparkSession.builder.appName("kNN_Maillo_k_variation").getOrCreate()
sc = spark.sparkContext

# ------------------------------
# LOAD TRAIN AND TEST DATASET
# ------------------------------
cols = [f"f{i}" for i in range(10)] + ["label"]
n_train = 10000
n_test = int(0.2 * n_train)

train = spark.read.csv("poker-hand-training-true.data", header=False,
inferSchema=True).toDF(*cols).limit(n_train)
test = spark.read.csv("poker-hand-testing.data", header=False,
inferSchema=True).toDF(*cols).limit(n_test)

print(f"train size: {train.count()}, test size: {test.count()}")

# ------------------------------
# Distance RDD/DataFrame
# ------------------------------
def euclidean_distance(p1, p2):
    return math.sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2)))

def map_knn_local(partition):
    partition = list(partition)
    results = {i: [] for i in range(len(test_broadcast.value))}
    for idx, test_point in enumerate(test_broadcast.value):
        test_features = [test_point[f"f{i}"] for i in range(10)]
        local_neighbors = []
        for train_point in partition:
```

```
            train_features = [train_point[f"f{i}"] for i in range(10)]
            dist = euclidean_distance(test_features, train_features)
            local_neighbors.append((dist, train_point["label"]))
        local_neighbors = sorted(local_neighbors, key=lambda x: x[0])[:current_k]
        results[idx] = local_neighbors
    for idx, neighbors in results.items():
        yield (idx, neighbors)

def reduce_knn_global(neighbors_lists):
    merged = []
    for lst in neighbors_lists:
        merged.extend(lst)
    return sorted(merged, key=lambda x: x[0])[:current_k]

def majority_vote(neighbors):
    labels = [lbl for _, lbl in neighbors]
    return max(set(labels), key=labels.count)

distance_udf = udf(lambda f1, f2: float(math.sqrt(sum((a-b)**2 for a,b in zip(f1,f2)))),
"float")

# --------------------------------
# Paramètres à tester
# --------------------------------
k_values = [3, 5, 7]      # Valeurs de k
partitions_list = [4, 6, 8]  # Nombre de partitions
results = []

# --------------------------------
# Broadcast des test points pour RDD
# --------------------------------
test_points = test.collect()
test_broadcast = sc.broadcast(test_points)

# --------------------------------
# Boucle sur toutes les combinaisons
# --------------------------------
for current_k in k_values:
    # ---------------- RDD ----------------
    for n_partitions in partitions_list:
        train_rdd = train.rdd.repartition(n_partitions)
        start_rdd = time.time()
        mapped = train_rdd.mapPartitions(map_knn_local)
        reduced = mapped.groupByKey().mapValues(reduce_knn_global)
        predictions_rdd = reduced.mapValues(majority_vote)
        correct_rdd = predictions_rdd.filter(lambda x: test_broadcast.value[x[0]]["label"] ==
        x[1]).count()
        total_rdd = predictions_rdd.count()
        accuracy_rdd = correct_rdd / total_rdd
        time_rdd = time.time() - start_rdd

        results.append({
            "version": "RDD",
            "k": current_k,
            "partitions": n_partitions,
            "accuracy": accuracy_rdd,
            "time": time_rdd
        })
        print(f"RDD - k={current_k}, Partitions={n_partitions} => Accuracy={accuracy_rdd:.4f},
        Time={time_rdd:.2f}s")

    # ---------------- DataFrame ----------------
```

```
    train_df = train.withColumn("features", array(*[col(f"f{i}") for i in range(10)]))
    test_df = test.withColumn("features", array(*[col(f"f{i}") for i in range(10)]))

    for n_partitions in partitions_list:
        spark.conf.set("spark.sql.shuffle.partitions", n_partitions)

        start_df = time.time()
        df_cross = test_df.crossJoin(train_df.withColumnRenamed("features","train_features")
                                    .withColumnRenamed("label","train_label"))
        df_cross = df_cross.withColumn("distance", distance_udf(col("features"),
        col("train_features")))

        # MAP phase
        window_local = Window.partitionBy("features",
        spark_partition_id()).orderBy(col("distance").asc())
        df_local_knn = df_cross.withColumn("rn",
        row_number().over(window_local)).filter(col("rn") <= current_k)

        # REDUCE phase
        window_global = Window.partitionBy("features").orderBy(col("distance").asc())
        df_global_knn = df_local_knn.withColumn("rn_global",
        row_number().over(window_global)).filter(col("rn_global") <= current_k)

        # Vote majoritaire
        window_vote = Window.partitionBy("features", "train_label")
        df_votes = df_global_knn.withColumn("cnt", count("train_label").over(window_vote))
        window_max = Window.partitionBy("features").orderBy(col("cnt").desc())
        df_pred = df_votes.withColumn("rn_vote",
        row_number().over(window_max)).filter(col("rn_vote") == 1)

        df_eval = df_pred.withColumn("correct", (col("label") ==
        col("train_label")).cast("integer"))
        accuracy_df = df_eval.agg({"correct":"avg"}).collect()[0][0]
        time_df = time.time() - start_df

        results.append({
            "version": "DataFrame",
            "k": current_k,
            "partitions": n_partitions,
            "accuracy": accuracy_df,
            "time": time_df
        })
        print(f"DataFrame - k={current_k}, Partitions={n_partitions} =>
        Accuracy={accuracy_df:.4f}, Time={time_df:.2f}s")

# ------------------------------
# Affichage tableau des résultats
# ------------------------------
df_results = pd.DataFrame(results)
print("\nTableau des résultats :")
print(df_results)

# ------------------------------
spark.stop()
```

B.6 Visualisation.

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
```

```
# --------------------------------
# Transformer les résultats en DataFrame
# --------------------------------
df_results = pd.DataFrame(results)

# --------------------------------
# Graphique Accuracy
# --------------------------------
plt.figure(figsize=(10, 5))
sns.lineplot(data=df_results, x="k", y="accuracy", hue="bucketLength", marker="o")
plt.title("Impact de k et bucketLength sur l'Accuracy")
plt.xlabel("k (nombre de voisins)")
plt.ylabel("Accuracy")
plt.grid(True)
plt.legend(title="bucketLength")
plt.show()

# --------------------------------
# Graphique Temps d'exécution
# --------------------------------
plt.figure(figsize=(10, 5))
sns.lineplot(data=df_results, x="k", y="time", hue="bucketLength", marker="o")
plt.title("Impact de k et bucketLength sur le Temps d'exécution")
plt.xlabel("k (nombre de voisins)")
plt.ylabel("Temps (s)")
plt.grid(True)
plt.legend(title="bucketLength")
plt.show()
```

B.7 Scalabilité : Accuracy RDD vs. DF

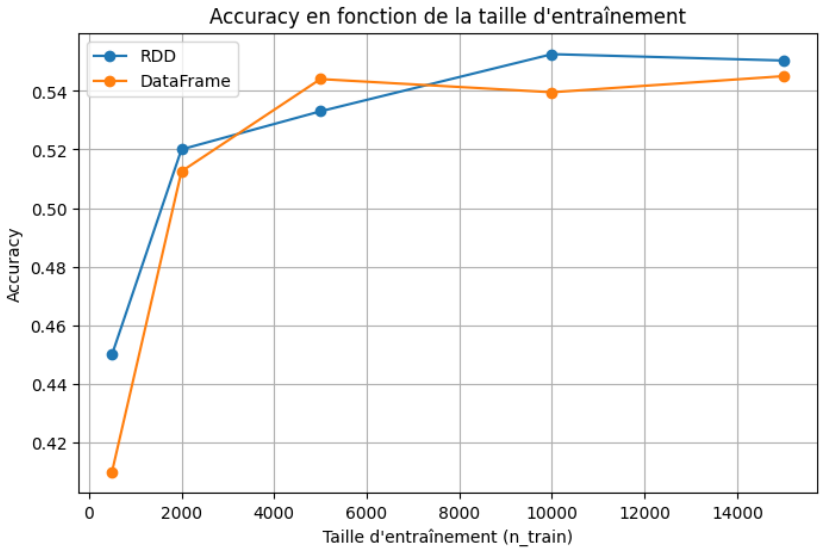

Figure 1: Enter Caption

B.8 Scalabilité : Temps RDD vs. DF

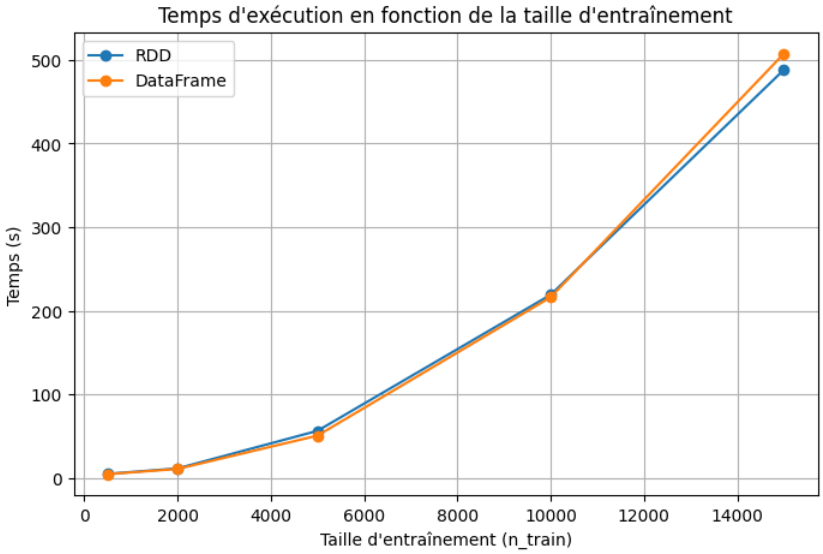

Figure 2: Enter Caption