Minwise-Independent Permutations with Insertion and Deletion of Features

Rameshwar Pratap[1(✉)] and Raghav Kulkarni[2]

[1] IIT Hyderabad, Telangana, India
rameshwar@cse.iith.ac.in
[2] Chennai Mathematical Institute, Chennai, Tamil Nadu, India
kulraghav@gmail.com

Abstract. The seminal work of Broder *et al.* [5] introduces the minHash algorithm that computes a low-dimensional sketch of high-dimensional binary data that closely approximates pairwise Jaccard similarity. Since its invention, minHash has been commonly used by practitioners in various big data applications. In many real-life scenarios, the data is dynamic and their feature sets evolve over time. We consider the case when features are dynamically inserted and deleted in the dataset. A naive solution to this problem is to repeatedly recompute minHash with respect to the updated dimension. However, this is an expensive task as it requires generating fresh random permutations. To the best of our knowledge, no systematic study of minHash is recorded in the context of dynamic insertion and deletion of features. In this work, we initiate this study and suggest algorithms that make the minHash sketches adaptable to the dynamic insertion and deletion of features. We show a rigorous theoretical analysis of our algorithms and complement it with supporting experiments on several real-world datasets. Empirically we observe a significant speed-up in the running time while simultaneously offering comparable performance with respect to running minHash from scratch. Our proposal is efficient, accurate, and easy to implement in practice.

Keywords: Sketching algorithms · Jaccard similarity estimation · Streaming algorithms · Locality sensitive hashing (LSH)

1 Introduction

Sets are one of the popular ways to embed data points, and their pairwise similarities are captured using Jaccard similarity. For a pair of sets $U, V \subseteq [d]$, their Jaccard similarity is defined as $|U \cap V|/|U \cup V|$. The seminal work of Broder *et al.* [5] suggests the minHash algorithm that computes a low-dimensional representation (or *sketch*) of the high-dimensional binary data that closely approximates the underlying pairwise Jaccard similarity. We discuss it as follows:[1]

[1] We note that binary vectors and sets give two equivalent representations of the same data object. Let the data elements be a subset of a fixed universe. In the

O. Pedreira and V. Estivill-Castro (Eds.): SISAP 2023, LNCS 14289, pp. 171–184, 2023.
https://doi.org/10.1007/978-3-031-46994-7_15

Definition 1 (Minwise Independent Permutations [5]). Let S_d be the set of all permutations on $[d]$. We say that $F \subseteq S_d$ (the symmetric group) is min-wise independent if for any set $U \subseteq [d]$ and any $u \in U$, when π is chosen at random in F, we have

$$\Pr[\min\{\pi(U)\} = \pi(u)] = 1/|U|. \tag{1}$$

For a permutation $\pi \in F$ chosen at random and a set $U \subseteq [d]$, Broder *et al.* [5] define minHash as follows $\text{minHash}_\pi(U) = \arg\min_u \pi(u)$ for $u \in U$. For a pair of points, $U, V \subseteq [d]$, and π is chosen at random in F, we have the following

$$\Pr[\text{minHash}_\pi(U) = \text{minHash}_\pi(V)] = |U \cap V|/|U \cup V|. \tag{2}$$

The above characteristic demonstrates the locality-sensitive nature (LSH) [13] of minHash, and as a consequence, it can be effectively used for the approximate nearest neighbour search problem. minHash is successfully applied in several real-life applications such as computing document similarity [3], item-set mining [2], faster de-duplication [4], all-pair similarity search [1], document clustering [6], building recommendation engine [11], near-duplicate image detection [9], web-crawling [12,18].

This work considers the scenario where features are dynamically inserted and/or deleted from the input. We emphasize that this natural setting may arise in many applications. Consider the *"Bag-of-Word" (BoW)* representation of text, where first, a dictionary is created using the important words present in the corpus such that each word present in the dictionary corresponds to a feature in the representation. Consequently, the embedding of each document is generated using this dictionary based on the frequency of the words present. Consider the downstream applications where the task is to compute pairwise Jaccard similarities between these documents, and the dimensionality of the *BoW* representation is high due to the large dictionary size. We can use minHash to compute the low-dimensional sketch of input documents. It is natural to assume that the dictionary is evolving; new words are inserted, and unused words are deleted. One evident approach to handle such a dynamic scenario is to run the minHash from scratch on the updated dictionary, which is expensive since it involves generating fresh min-wise independent (random) permutations. Note that during the insertion/deletion of features in the dataset, we consider inserting/deleting the same features in all the data points. To clarify this further, let $\mathcal{D} = \{X_i\}_{i=1}^n$ be our dataset, where $X_i \in \{0,1\}^d$. Considering the addition/removal of the j-th feature, the j-th feature gets inserted/deleted in the point X_i. Similarly, the corresponding j-th feature is inserted/deleted in all the remaining points in $\mathcal{D}$. Note that we don't consider the case when data points are dynamically inserted or deleted in the dataset.

corresponding binary representation, we generate a vector whose dimension is the size of the universe, where for each possible element of the universe, a feature position is designated. To represent a set into a binary vector, we label each element's location with 1 if it is present in the set and 0 otherwise.

Problem statement: minHash **for dynamic insertion and deletion of features:** In this work, we focus on making minHash adaptable to dynamic feature insertions and deletions of features. We note that the insertion/deletion of features dynamically leads to the expansion/shrinkage of the data dimension.

We note that in practice a d dimensional permutation required for minHash is generated via the universal hash function $h_d(i) = ((ai + b) \mod p) \mod d$, where p is a large prime number and a, b are randomly sampled from $\{0, 1, \ldots p-1\}$; typically $((ai + b) \mod p) > d$[2]. This hash function generates permutations via mapping each index $i \in [d]$ to another index $[d]$ that can be used to compute the minHash sketch. We note that in the case of dynamic insertions/deletion of features, even using universal hash functions to compute the minHash sketch doesn't give an efficient solution. We illustrate it as follows. Suppose we have a minHash sketch of data points using the hash function $h_d(.)$. Consider the case of feature insertion, where the dimension d increases to $d + 1$, and therefore, we require a hash function $h_{d+1}(.)$ to generate a $(d + 1)$-dimensional permutation. Note that the permutation generated via $h_{d+1}(.)$ can potentially be different on several values of $i \in [d + 1]$. Therefore, just computing $h_{d+1}(d + 1)$, taking the corresponding input feature, and taking the minimum of this quantity with the previous minHash would not suffice to compute minHash after feature insertion. If implemented naively, this re-computation step takes $O(d)$ in the worst case. A similar argument also holds in the case of feature deletion.

1.1 Our Contribution

In this work, we consider the problem of making minHash adaptable to dynamic insertions and deletions of features. We focus on cases where features are inserted/deleted at randomly chosen positions from 1 to d. We argue that this is a natural assumption that commonly occurs in practice. For example, in the context of *BoW*, a word's position in the dictionary is determined via a random hash function that randomly maps it to a position from 1 to d. Therefore, when a new word is added to the dictionary, its final position in the representation appears as a random position (from 1 to d). A similar argument is also applicable to feature deletion. We summarize our contributions as follows:

◇ **Contribution 1:** We present algorithms (Sect. 2) that makes minHash sketch adaptable to single/multiple feature insertions. Our algorithm takes the current permutation and the corresponding minHash sketch; values and positions of the inserted features as input and outputs the minHash sketch corresponding to the updated dimension.
◇ **Contribution 2:** We also suggest algorithms (deferred to the full version of the paper [21] due to the space limit, discussed in Section 4 of [21]) that makes minHash sketch adaptable for single/multiple feature deletions. It takes the data points, current sketch, and permutations used to generate the same; positions of the deleted features and outputs the minHash sketch corresponding to the updated dimension.

[2] These hash functions are called universal hash functions (see Chapter 11 of [10].

Our work leaves the possibility of some interesting open questions: to propose algorithms when features are inserted or deleted adversarially (rather than uniformly at random from 1 to d, as considered in this work). We hope that our techniques can be extended to handle this situation.

Our Techniques and Their Advantages: A major benefit of our results is that they do not require generating fresh random permutations corresponding to the updated dimension (after feature insertions/deletions) to compute the updated sketch. We implicitly generate a new permutation (required to compute the sketch after feature insertion/deletion) using the old d-dimensional permutation and also show that it satisfies the min-wise independence property (Definition 1). We further give simple and efficient update rules that take the value and position of inserted/deleted features, and output the updated minHash sketch. To show the correctness of our result, we prove that the sketch obtained via our update rule is the same as obtained via computing minHash from scratch using the implicitly generated permutation mentioned above. For both insertions and deletion cases, our algorithms give significant speedups in dimensionality reduction time while offering almost comparable accuracy with respect to running minHash from scratch. We validate this by running extensive experiments on several real-world datasets (Sect. 3 and Table 3). We want to emphasize that our algorithms can also be easily implemented when permutations are generated via random hash functions.

Applicability of our Result in Other Sketching Algorithms for Jaccard Similarity: We note that there are several improved variants of minHash are known such as one-permutation hashing [15,22], b-bit minwise hashing [14,16], oddsketch [20] that offer space/time efficient sketches. We would like to highlight that our algorithms can be easily adapt to these improved variants of minHash, in case of dynamic insertion and deletion of features. One permutation hashing divides the permuted columns evenly into k bins. For each data point, the sketch is computed by picking the smallest nonzero feature location in each bin. In the case of dynamic settings, our algorithms can be applied in the bin where features are getting inserted/deleted. Both $b-$bit minwise hashing [14] and oddsketch [20] are two-step sketching algorithms. In their first step, the minHash sketch of the data points is computed. In the second step of b-bit minwise hashing, the last b-bits (in the binary representation) of each minHash signature is computed. In contrast, in the second step of oddsketch, one bit of each minHash sketch is computed using their proposed hashing algorithm. As both results compute the minHash sketch in their first step, we can apply our algorithms to compute the minHash sketch in case of feature insertion/deletion. This will make their algorithms adaptable to dynamic feature insertions and deletions.

Recently, some hashing algorithms have been proposed that closely estimate the pairwise Jaccard similarity [7,8,19] without computing their minHash sketch. However, to the best of our knowledge, their dynamic versions (that can handle dynamic insertions/deletions of features) are unknown. Several improvements of the LSH algorithm [23] have been proposed that are adaptable to the dynamic/streaming framework. However, a significant difference is in the under-

lying problem statement. These results aim to handle dynamic insertion and deletions of data points, whereas we focus on dynamic insertions and deletions of the features (Table 1).

2 Algorithm for Feature Insertion

Table 1. Notations

Data dimension	d	Input data point $\{0,1\}^d$ or input set	X
Set $\{1,\ldots,d\}$	$[d]$	Data point after feature insertion $\{0,1\}^{d+1}$	X'
Position of the inserted feature	m	Original d-dim. permutation $(a_1,\ldots,a_d)$ $s.t.$ $a_i \in [d]$	π
Value of the inserted feature	b	Lifted $(d+1)$-dim. permutation $(a'_1,\ldots a'_{d+1})$ $s.t.$ $a'_i \in [d+1]$	π'_m
No. of 1′s in X	$\lvert X\rvert$	Set of non-zero indices of X, $i.e.$, $\{i \lvert x_i = 1\}$	J
Size of the set J	$\lvert J\rvert$	minHash of X with π, $i.e.$, $\text{minHash}_\pi(X)$	h_{old}

We first give our algorithm for a single feature insertion.

2.1 One Feature Insertion at a Time – liftHash

The liftHash (Algorithm 2) is our main algorithm for updating the sketch of data points consisting of binary features. It takes a d dimensional permutation π and the corresponding minHash sketch h_{old} π as input. In addition, it takes an index m and a bit value b, corresponding to the position and the value of the binary feature, to be inserted, respectively, and outputs updated hash value h_{new}. We show that h_{new} corresponds to a minHash sketch of the updated feature vector. To show this, we use liftPerm (Algorithm 1), which extends the original permutation π to a $(d+1)$ dimensional min-wise independent permutation. Note that the liftPerm algorithm is used solely for the proof and not required in the liftHash algorithm.

The main intuition of our algorithm is that we can (implicitly) generate a new $(d+1)$-dimensional permutation by reusing the old d-dimensional permutation (Algorithm 1), and can update the corresponding minHash $w.r.t.$ the new $(d+1)$-dimensional permutation via a simple update rule (Algorithm 2). Consider a d dimensional input vector $X = (x_1, x_2, \ldots, x_d)$. A permutation π of $\{1,2,\ldots,d\}$ can be thought of as imposing the following ordering on the indices of X: $\pi(1), \pi(2), \ldots, \pi(d)$. After feature insertion, we want the (implicit) liftPerm algorithm to generate a new permutation π' of $\{1,2,\ldots,d+1\}$ that still maintains the ordering that was imposed by π. We show that such an extension is achievable with high probability assuming (i) feature insertion is happening at a random position and (ii) our binary feature vector is sparse. This helps us guarantee (with high probability) that π' is min-wise independent if π is min-wise independent (see Theorem 2). Finally, we show that the sketch obtained by

the liftHash algorithm is the same one produced by applying the minHash with respect to the output π' of the liftPerm algorithm (see Theorem 3).

Algorithm 1: liftPerm(π, r).

Input: d-dim permutation π, a number r.
Output: $(d+1)$-dim. permutation π'.
for $i \in \{1, \ldots, d+1\}$ **do**
  if $i \leq r$ **then**
    $\pi'(i) = \pi(i)$
  else
    $\pi'(i) = \pi(i-1)$
  end
end
for $i \in \{1, \ldots, d+1\}/\{r\}$ **do**
  if $\pi'(i) \geq \pi'(r)$ **then**
    $\pi'(i) = \pi'(i) + 1$
  end
end
return π'

Algorithm 2: liftHash(π, m, b, h_{old}).

Input: $h_{old} := \text{minHash}_\pi(X)$, π, $m \in [d]$, $b \in \{0, 1\}$.
Output: $h_{new} := \text{liftHash}(\pi, m, b, h_{old})$.
Denote $a_m = \pi(m)$.
`/* m is the position of the inserted feature */`
if $h_{old} < a_m$ **then**
  $h_{new} = h_{old}$
else
  if $b = 1$ **then**
    $h_{new} = a_m$
  end
  if $b = 0$ **then**
    $h_{new} = h_{old} + 1$
  end
end
return h_{new}

We illustrate our algorithm with the following example and then state its proof of correctness.

Example 1 We illustrate our Algorithms using the following example. We assume that the index count starts with 1. Let $X = [1, 0, 0, 1, 0, 1, 0]$ be the data point, and $\pi = [6, 3, 1, 7, 2, 5, 4]$ be the original permutation. Then $\text{minHash}_\pi(X)$ is 5. Further, let us assume that we insert the value $b = 1$ at the index $m = 2$. Therefore $a_m = \pi(m) = 3$. The updated value $X' = [1, 1, 0, 0, 1, 0, 1, 0]$ and due to Algorithm 1 by setting $r = m = 2$, we obtain $\pi'_m = [7, 3, 4, 1, 8, 2, 6, 5]$. We

calculate the value of h_{new} outputted by Algorithm 2: as $h_{old} = 5 > a_m = 3$ and $b = 1$, then we have $h_{new} = \text{liftHash}(\pi, m, b, h_{old}) = a_m = 3$. Further, $\text{minHash}_{\pi'_m}(X') = 3$. Therefore, we have $h_{new} = \text{minHash}_{\pi'_m}(X')$.

[3]The following theorem gives proof of the correctness of Algorithm 1, and shows that the permutation π' outputted by the algorithms satisfies the minwise independent property (Definition 1), with high probability. At a high-level proof of Theorem 2 relies on showing the bijection between the ordering on the indices of X by the original d-dimensional permutation π, and $(d+1)$-dimensional permutation π'. We show that this bijection holds with probability 1 when inserted feature value $b = 0$, and holds with a high probability when $b = 1$.

Theorem 2 *Let $\pi = (a_1 \ldots, a_d)$ be a minwise independent permutation, where $a_i \in [d]$, and r be a random number from $[d]$. Let π and r be the input to Algorithm 1. Then for any $X \in \{0,1\}^d$ with $|X| \leq k$, the permutation $\pi' = (a'_1 \ldots, a'_{d+1})$, where $a'_i \in [d+1]$, obtained from Algorithm 1 satisfies the condition stated in Equation (1) of Definition 1, with probability at least $1 - O(k/d)$.*

Theorem 3 gives proof of the correctness of Algorithm 2. We show that the sketch outputted by Algorithm 2 is the same as obtained by running minHash using the $(d+1)$-dimensional permutation obtained by Algorithm 1 on the updated data point after one feature insertion.

Theorem 3 *Let π'_m be the $(d+1)$-dimensional permutation outputted by Algorithm 1 by setting $r = m$. Then, the sketch obtained from Algorithm 2 is exactly the same as the sketch obtained with the permutation π'_m on X', that is, $h_{new} := \text{liftHash}(\pi, m, b, h_{old}) = \text{minHash}_{\pi'_m}(X')$.*

Remark 1 We remark that in order to compute the minHash sketch of X', Algorithm 2 requires only h_{old}, b, m, the value of $\pi(m)$. Whereas *vanilla* minHash requires a fresh $(d+1)$ dimensional permutation to compute the same.

Remark 2 We can extend our results for multiple feature insertion by repeatedly applying Theorem 2, and Theorem 3 along with the probability union bound. However, the time complexity of the algorithm obtained by sequentially inserting n features will grow linearly in n as observed in the empirical results (Fig. 1, Sect. 3). In the next subsection, we present an algorithm that performs multiple insertions in parallel, which helps us achieve much better speedups.

2.2 Algorithm for Multiple Feature Insertions – multipleLiftHash

Results presented in this subsection are extensions to that of Subsect. 2.1. The intuition of our proposal is that we can (implicitly) generate a new $(d+n)$-dimensional permutation (n is the number of inserted features), using the old

[3] We defer the proofs of Theorems 2, 3, 5, 6, to the full version of this paper [21] due to space limit.

Table 2. Notations

No. of inserted features	n	Position of inserted features $\{m_i\}_{i=1}^{n}$, $m_i \in [d+1]$	M
X after n features insertion $\{0,1\}^{d+n}$	X'	Set of inserted bits $\{b_1, \ldots, b_n\}$ with $b_i \in \{0,1\}$	B
multipleLiftHash(M, π, B, h_{old})	h_{new}	Lifted $(d+n)$-dim. permutation	π'_M

d-dimensional permutation. By exploiting the sparsity of input and the fact that inserted bits are random positions, we show that the updated permutation satisfies the min-wise independent property with high probability. Further, we suggest a simple update rule aggregating the existing minHash sketch and the minHash restricted to inserted position and outputs the updated sketch.

Algorithm 3: partialMinHash(π, M, B)

Input: Permutation π, a sorted set of indices $M = \{m_1, ...m_n\}$, and set of inserted bits $B = \{b_1, \ldots, b_n\}$
Output: The min value of π (with appropriate shift) restricted to only those indices m_i of M that correspond to non-zero b_i.
$\pi_{M,B} = \{\pi(m_i) \mid i \in \{1, \ldots, n\}$ and $b_i = 1\}$
return $\min\{\pi_{M,B}(k) + 1\}$

Algorithm 4: multipleLiftPerm(π, R).

Input: Permutation π; R with $|R| = n$.
Output: $(d+n)$-dim. permutation π'.
$R \leftarrow$ sorted(R) `/* sorting array` R `in ascending order */`
for $i \in \{1, 2, \ldots n\}$ **do**
  $R[i] = R[i] + i - 1$
end
$\pi' = \pi$ `/* Initialization */`
for $i \in \{1, \ldots n\}$ **do**
  $\pi' =$ liftPerm$(\pi', R[i])$ `/* Calling Algorithm 1 with` $\pi = \pi'$ `and` $r = R[i]$ `*/`
end
return π'

Algorithm 5: multipleLiftHash(M, π, B, h_{old}).

Input: $h_{old} :=$ minHash$_\pi(X)$, permutation π, M and B.
Output: $h_{new} :=$ multipleLiftHash(M, π, B, h_{old}).
Let $\pi_M := \{\pi(m) : m \in M\}$.
$a_M =$ partialMinHash(π, M, B)
$h_{new} = \min\left(h_{old} + |\{x \mid x \in \pi_M \text{ and } x \leq h_{old}\}|, a_M\right)$ `/* Picking the minimum between` partialMinHash `and shifted value of` h_{old}. `*/`
return h_{new}

Algorithm 5 takes h_{old}, M, B, and π as input, and outputs the updated sketch h_{new}. Algorithm 5 uses Algorithm 3 to obtain the value of partialMinHash – minimum π value restricted to the inserted indices only with inserted bit value 1, from which it obtains multipleLiftHash for the updated input. Algorithm 4 is implicit and is used to prove the correctness of Algorithm 5. Algorithm 4 takes the permutation π and M as input, and outputs a $(d+n)$-dimensional permutation π'_M which satisfies the condition stated in Equation (1) for X, with $|X| \leq k$. We show this in Theorem 5. Then in Theorem 6, we show that $h_{new} = \text{minHash}_{\pi'_M}(X')$. As π'_M satisfies, the condition stated in Equation (1) for sparse X, then due to Equation (2) and [5] the sketch of data points obtained from Algorithm 5 approximates the Jaccard similarity.

Example 4 Suppose $X = [1,0,0,1,0,1,0]$ and $\pi = [6,3,1,7,2,5,4]$ are input point and original permutation, respectively. Then the value of h_{old} is 5. Let $M = [2,4]$ and $B = [0,1]$. Thus, in this case $\pi'_M = [7,3,4,1,8,9,2,6,5]$ and $X' = [1,0,0,0,1,1,0,1,0]$. Consequently we have, $\text{partialMinHash}(\pi, M, B) = 2 < h_{old} + |\{x \mid x \in \pi_M \text{ and } x \leq h_{old}\}| = 5+1 = 6$. Therefore, $\text{minHash}_{\pi'_M}(X') = 2$.

We have the following theorems for the correctness of the algorithms presented in this subsection. A proof of the Theorem 5 follows similarly to the proof of Theorem 2 along with the probability union bound, and the proof of Theorem 6 is a generalization of proof of Theorem 3.

Theorem 5 *Let π be a minwise independent permutation. Let $M = \{m_1, \ldots, m_n\}$ such that m_i is chosen uniformly at random from $\{1, \ldots, d\}$. Then for any $X \in \{0,1\}^d$ with $|X| \leq k$, the permutation π'_M obtained from Algorithm 4 satisfies the condition stated in Equation (1) of Definition 1, with probability $1 - O(kn/d)$.*

Theorem 6 *Let π'_M be the $(d+n)$-dimensional permutation outputted by Algorithm 4, if we set $R = M$. Then, the sketch obtained from Algorithm 5 is exactly the same as the sketch obtained with the permutation π'_M on X', that is,* $\text{multipleLiftHash}(\pi, M, B, h_{old}) = \text{minHash}_{\pi'_M}(X')$.

Along similar lines, we give algorithms for single and multiple-feature deletions. Due to space limit, we defer it to Section 4 of the full version of this paper [21].

3 Experiments

Hardware Description: CPU model name: Intel(R) Xeon(R) CPU @ 2.20 GHz; RAM:12.72 GB; Model name: Google Colab.

Datasets and Baselines: We perform our experiments on *"Bag-of-Words"* representations of text documents [17]. We use the following datasets: NYTimes news articles (number of points = 300000, dimension = 102660), Enron emails

(number of points = 39861, dimension= 28102), and KOS blog entries (number of points = 3430, dimension = 6960).

We consider the binary version of the data, where we focus on the presence/absence of a word in the document. For our experiments, we considered a random sample of 500 points from the NYTimes and 2000 points for Enron and KOS.

We compare the performance of our algorithms multipleLiftHash and multipleDropHash with respect to running minHash from scratch on the updated dimension, and we refer to it as vanilla minHash. We also note the performance of sequential versions of single feature insertion/deletion algorithms – liftHash and dropHash, respectively. We give implementation details of the baseline algorithms as the following link https://tinyurl.com/y98yh6k3.

Table 3. Speedup of our algorithms *w.r.t* their vanilla minHash version

Experiment	Method	NYTimes		Enron		KOS	
		Max.	Avg.	Max.	Avg.	Max.	Avg.
Feature Insertions	multipleLiftHash	54.91×	51.96×	9.61×	9.17×	24.4×	23.11×
	liftHash	91.23×	87.38×	13.96×	12.66×	35.00×	35.50×
Feature Deletions	multipleDropHash	109.5×	105.31×	18.6×	17.01×	46.02×	43.94×
	dropHash	78.34×	72.79×	15.95×	14.89×	38.24×	35.71×

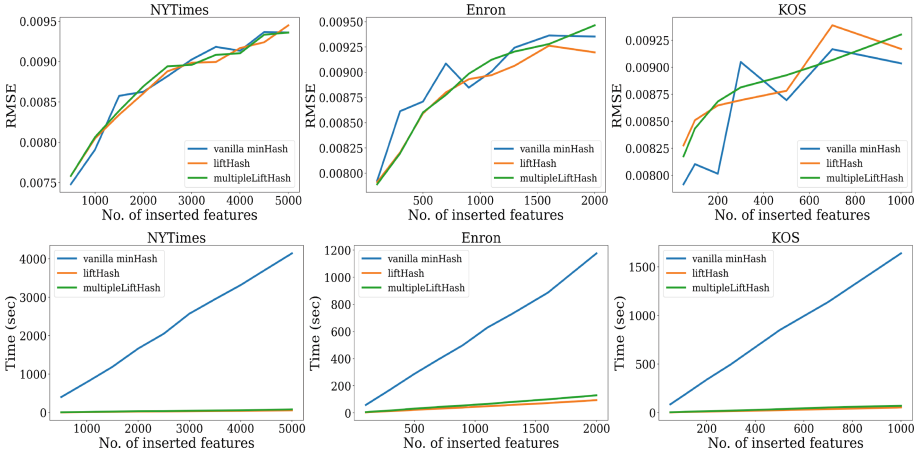

Fig. 1. Comparison among liftHash, multipleLiftHash, and vanilla minHash on the task of feature insertions. Vanilla minHash corresponds to computing minHash on the updated dimension. We iteratively run liftHash n times, where n is the number of inserted features

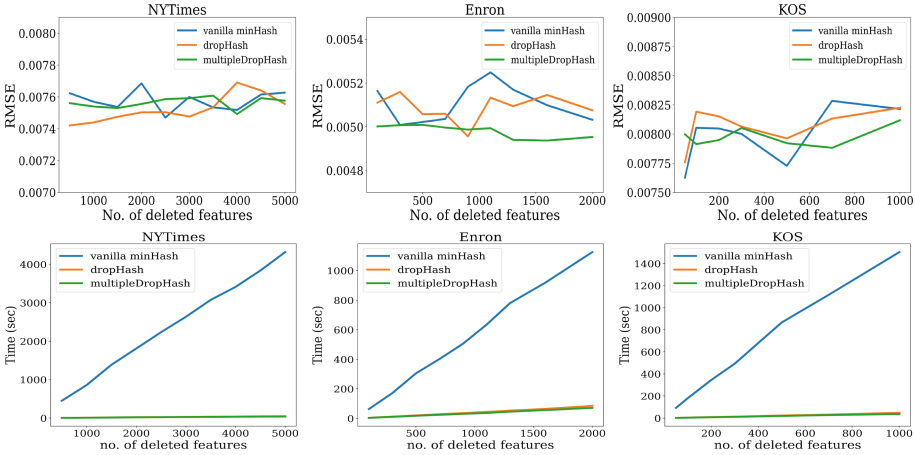

Fig. 2. Comparison among dropHash, multipleDropHash, and vanilla minHash on feature deletions. We iteratively run dropHash n times, where n is the number of deleted features

3.1 Experiments for Feature Insertions

We use two metrics for evaluation: a) RMSE: to examine the quality of the sketch, and b) running time: to measure the efficiency. We first create a 500 dimensional minHash sketch for each dataset using 500 independently generated permutations. Consider that we have a set of n random indices representing the locations where features need to be inserted. For each position, we insert the bit 1 with probability 0.1 and 0 with probability 0.9. We then run the liftHash algorithm (Algorithm 2) after each feature insertion; we repeat this step until n feature insertions are done. This gives a minHash sketch corresponding to the liftHash algorithm. We again run our multipleLiftHash algorithm (Algorithm 5) on the initial 500 dimensional sketch with the parameter n. We compare our methods with vanilla minHash by generating a 500 dimensional sketch corresponding to the updated datasets after feature insertions.

For computing the RMSE, our ground truth is the pairwise Jaccard similarity on the original full-dimensional data. We measure it by computing the square root of the mean (over all pairs of sketches) of the square of the difference between the pairwise ground truth similarity and the corresponding similarity estimated from the sketch. A lower RMSE is an indication of better performance. We compare the RMSE of our methods with that of vanilla minHash by generating a fresh 500 dimensional sketch. We summarise our results in Fig. 1.

Insights: Both of our algorithms offer comparable performance (under RMSE) with respect to running minHash from scratch on the updated dimension. That is, our estimate of the Jaccard similarity is as accurate as the one obtained by computing minHash from scratch on the updated dimension. Simultaneously, we

obtain significant speedups in running time compared to running minHash from scratch. In particular, the speedup for multipleLiftHash is noteworthy (Table 3).

3.2 Experiments for Feature Deletion

We use the same metric as feature insertion experiments – RMSE and running time. We first create a 500 dimensional minHash sketch for each dataset using minHash. Suppose we have a list of n indices that denote the position where features need to be deleted. After each feature deletion, we run the dropHash algorithm (discussed in Section 4.1 of [21] - full version of this paper). We repeat this step n times. This gives a minHash sketch corresponding to the dropHash algorithm. We again run our multipleDropHash algorithm (discussed in Section 4.2 of [21] - full version of this paper) on the initial 500 dimensional sketch with the parameter n. We compare our results with vanilla minHash by generating a fresh 500 dimensional sketch on the updated dataset. We note the RMSE and running time as above. We summarise our results in Fig. 2.

Insights: Again, both our algorithms offer comparable performance (under RMSE) with respect to running minHash from scratch. Similar to the previous case, we obtained a significant speedup in running time *w.r.t.* computing minHash from scratch. In particular, the speedup obtained in multipleDropHash is quite prominent. We summarise a numerical speedup in Table 3.

Remark 3 Our current implementation of multipleLiftHash makes multiple passes over indices to be inserted, whereas multipleDropHash makes only one pass over the deleted indices. This is reflected in higher speedup values for multipleDropHash in Table 3. We believe an optimized implementation for multipleLiftHash would further improve the speedup.

4 Conclusion and Open Questions

We present algorithms that make minHash adaptable to dynamic feature insertions and deletions of features. Our proposals' advantage is that they do not require generating fresh permutations to compute the updated sketch. Our algorithms take the current permutation (or its representation using universal hash function [10]), minHash sketch, position, and the corresponding values of inserted/deleted features and output updated sketch. The running time of our algorithms remains linear in the number of inserted/deleted features. We comprehensively analyse our proposals and complement them with supporting experiments on several real-world datasets. Our algorithms are simple, efficient, and accurately estimate the underlying pairwise Jaccard similarity. Our work leaves the possibility of several interesting open questions: (i) extending our results for dense datasets in the case of feature insertions; (ii) extending our algorithms for the case when features are inserted/deleted adversely; (iii) improving our algorithms when we have prior information about the distribution of features; for example, features distribution follows *Zipf's law* etc.; (iv) improving theoretical guarantees and obtaining further speedups by optimizing our algorithms.

Acknowledgments. We sincerely thank Biswadeep Sen for providing their valuable input on the initial draft of the paper.

References

1. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: Proceedings of the 16th International Conference on World Wide Web, WWW 2007, pp. 131–140. Association for Computing Machinery , New York, NY, USA (2007)
2. Bera, D., Pratap, R.: Frequent-itemset mining using locality-sensitive hashing. In: Dinh, T.N., Thai, M.T. (eds.) COCOON 2016. LNCS, vol. 9797, pp. 143–155. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42634-1_12
3. Broder, A.Z.: On the resemblance and containment of documents. In: . Proceedings of Compression and Complexity of Sequences 1997, pp. 21–29. IEEE (1997)
4. Broder, A.Z.: Identifying and filtering near-duplicate documents. In: Giancarlo, R., Sankoff, D. (eds.) CPM 2000. LNCS, vol. 1848, pp. 1–10. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45123-4_1
5. Broder, A.Z., Charikar, M., Frieze, A.M., Mitzenmacher, M.: Min-wise independent permutations (extended abstract). In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, STOC 1998, pp. 327–336. Association for Computing Machinery, New York, NY, USA (1998)
6. Broder, A.Z., Glassman, S.C., Nelson, C.G., Manasse, M.S., Zweig, G.G.: Method for clustering closely resembling data objects, September 12 2000. US Patent 6,119,124
7. Christiani, T., Pagh, R.: Set similarity search beyond minhash. In: Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, pp. 1094–1107. Association for Computing Machinery, New York, NY, USA, (2017)
8. Christiani, T., Pagh, R., Sivertsen, J.: Scalable and robust set similarity join. In: 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16–19, 2018, pp. 1240–1243. IEEE Computer Society (2018)
9. Chum, O., Philbin, J., Zisserman, A.: Near duplicate image detection: min-hash and TF-IDF weighting. In: Everingham, M., Needham, C.J., Fraile, R. (Eds.), Proceedings of the British Machine Vision Conference 2008, Leeds, UK, September 2008, pp. 1–10. British Machine Vision Association (2008)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
11. Das, A.S., Datar, M., Garg, A., Rajaram, S.: Google news personalization: scalable online collaborative filtering. In WWW 2007: Proceedings of the 16th international conference on World Wide Web, pp. 271–280. ACM, New York, NY, USA (2007)
12. Henzinger, M.: Finding near-duplicate web pages: a large-scale evaluation of algorithms. In: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2006, pp. 284–291. Association for Computing Machinery, New York, NY, USA (2006)
13. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23–26, 1998, pp. 604–613 (1998)
14. Li, P., König, A.C.: Theory and applications of b-bit minwise hashing. Commun. ACM **54**(8), 101–109 (2011)

15. Li, P., Owen, A.B., Zhang, C.-H.: One permutation hashing. In: Bartlett, P.L., Pereira, F.C.N., Burges, Léon Bottou, C.J.C., Weinberger, K.Q., (Eds.), Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3–6, 2012, Lake Tahoe, Nevada, United States, pp. 3122–3130 (2012)
16. Li, P., Shrivastava, A., König, A.C.: B-bit minwise hashing in practice. In: Proceedings of the 5th Asia-Pacific Symposium on Internetware, Internetware 2013, New York, NY, USA. Association for Computing Machinery (2013)
17. Lichman, M.: UCI machine learning repository (2013)
18. Singh Manku, G., Jain, A., Sarma, A.D.: Detecting near-duplicates for web crawling. In: Proceedings of the 16th International Conference on World Wide Web, WWW 2007, pp. 141–150. Association for Computing Machinery, New York, NY, USA (2007)
19. McCauley, S., Mikkelsen, J.W., Pagh, R.: Set similarity search for skewed data. In Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, SIGMOD/PODS 2018, pap.63–74, New York, NY, USA, 2018. Association for Computing Machinery (2018)
20. Mitzenmacher, M., Pagh, R. Pham, ,N.: Efficient estimation for high similarities using odd sketches. In: Proceedings of the 23rd International Conference on World Wide Web, WWW 2014, p–118. Association for Computing Machinery, New York, NY, USA, 2014
21. Pratap,R ., Kulkarni, R.: Minwise-independent permutations with insertion and deletion of features. arxiv.org/abs/2308.11240 (2023)
22. Shrivastava, A., Li, P.: Improved densification of one permutation hashing. In: Proceedings of the Thirtieth Conference On Uncertainty In Artificial Intelligence, UAI 2014, pp. 732–741. AUAI Press, Arlington, Virginia, USA, (2014)
23. Sundaram, N., et al.: Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. Proc. VLDB Endow. **6**(14), 1930–1941 (2013)