Binned Consistent Weighted Sampling

No Author Given

No Institute Given

Abstract. Similarity computation between two weighted sets is widely used for various applications including similarity search over images and ECG recordings. Consistent Weighted Sampling (CWS) is popular since it reduces weighted sets into short signatures to speed up each similarity computation. Improved CWS (ICWS) is more efficient than CWS, but it becomes very slow when generating many hash values with a large universal set. We propose Binned CWS (BCWS) that partitions the universal set into bins of fixed size and generates a hash value per bin. Also, we suggest a simple and efficient densification method that uses rejection sampling to densify sparse signatures generated by BCWS. Experiments using two important tasks, the GJS estimation and the 1-NN classification, demonstrate that BCWS outperforms state-of-the-art methods such as ICWS, CCWS and PCWS.

Keywords: Weighted Sampling, Locality Sensitive Hashing, Generalized Jaccard Similarity, Approximate Similarity Search

1 INTRODUCTION

Locality Sensitive Hashing (LSH) [1, 2] is an efficient estimation method for similarity measures such as Jaccard similarity and cosine similarity. MinHash [3, 4] is an LSH method for Jaccard similarity between unweighted sets. It is widely used for various applications such as image recognition [7, 12], near duplicate web page detection [4], similar document identification [3].

Given a universal set $\Omega = \{0, 1, \dots, p-1\}$, a weighted set is represented as p weights. Weighted sets are used to model various types of complex objects such as images and ECG recordings. Suppose that there are two ECG recordings, $S = \{s_0, s_1, s_2, \dots, s_{p-1}\}$ and $T = \{t_0, t_1, t_2, \dots, t_{p-1}\}$, where s_i (or t_i) corresponds to the voltage level of the i-th sampling of S (or T). The Generalized Jaccard similarity (GJS) between S and T is defined in Eq. (1).

$$GJS(S,T) = \frac{\sum_{j=0}^{p-1} min(s_j, t_j)}{\sum_{j=0}^{p-1} max(s_j, t_j)} \tag{1}$$

In order to estimate GJS efficiently, weighted minwise hashing is suggested in [10]. It converts a weighted set of integers into an unweighted set by adding extra elements in proportion to each weight. However, these extra elements increase the size of the universal set, which in turn will increase the execution time. Later, [8, 9] extended weighted minwise hashing to deal with weighted sets of real numbers.

Consistent Weighted Sampling (CWS) [14] generates the signature of a weighted set directly without converting it into an unweighted set. However, it generates many pairs of active indices, $(w_{ik},\, z_{ik})$, for the i-th weight $(1 \leq i \leq p)$ while it uses only the final pair of each weight to generate a hash value. Improved CWS (ICWS) [11] is more efficient than CWS in that it generates only a pair of active indices, $(w_i,\, z_i)$, for the i-th weight to generate each coupled hash value, (j, w_j). ICWS is considered as the state-of-the-art method for approximating GJS.

There are some variants of ICWS suggested to improve the execution time of ICWS. The 0-bit CWS [13] showed through experiments that the impact of j on each coupled hash value, (j, w_j), is much higher than that of w_j, thus w_j can be discarded. Canonical CWS (CCWS) [18] did not use the logarithm transformation of each weight that ICWS used for better uniformity of CWS. Instead, CCWS directly generates active indices from weights. Although it is more efficient than ICWS, its approximation accuracy is worse since highly similar weights may be transformed into different hash values. Practical CWS (PCWS) [19] used four different uniformly distributed variables to produce a coupled hash value, which is less than ICWS. This enables PCWS to outperform ICWS in terms of execution time.

Notice that ICWS and its variants produce a coupled hash value by generating many random numbers, at least three times of the size of the universal set $(3|\Omega|)$. Therefore, for datasets with a large universal set, the execution time can be very high. Recently, a novel approach [15] is proposed to estimate GJS using Rejection Sampling [5], but it needs the upper bound of each weight in the universal set.

We propose Binned CWS (BCWS) method that produces less random numbers to generate hash values more efficiently for a given weighted set. It reduces the total number of random numbers by partitioning the universal set into smaller bins and generates a hash value per bin. Our contributions are the followings:

– The time complexity of BCWS is $O(|\Omega|)$ regardless of the number of hash values it generates.
– Our densification method improves the accuracy degradation that may be caused by sparse signatures of BCWS.
– Experiments on real datasets demonstrate the performance superiority of our method in terms of execution time while generating signatures of the same quality.

2 BACKGROUND

We give a brief explanation on the details of Improved Consistent Weighted Sampling (ICWS) and discuss on the execution time of ICWS.

Algorithm 1 : Improved Consistent Weighted Sampling (ICWS)

```
Input: weighted set $S = \{s_0, s_1, s_2, \dots s_{p-1}\}$, number of hashes, $nh$, random seeds, $seed$
Output: signature of $S$, $sig$
 1: for $i = 0, 1, \dots, nh - 1$ do
 2:     $set\ seed_i$
 3:     for $j = 0, 1, \dots p - 1$ $such\ that\ s_j > 0$ do
 4:         $r_j \sim Gamma(2, 1)$
 5:         $c_j \sim Gamma(2, 1)$
 6:         $u_j \sim Uniform(0, 1)$
 7:         $h_j = \lfloor ln(s_j)/r_j + u_j \rfloor$
 8:         $w_j = exp(r_j(h_j - u_j))$
 9:         $a_j = c_j/(w_j * exp(r_j))$
10:     end for
11:     $j^* = argmin_j\ a_j$
12:     $sig_{i,0} = j^*$
13:     $sig_{i,1} = w_{j^*}$
14: end for
15: return $sig$
```

ICWS [11] estimates GJS between two weighted sets, $S = \{s_0, s_1, s_2, \dots, s_{p-1}\}$ and $T = \{t_0, t_1, t_2, \dots, t_{p-1}\}$. For each weighted set, it generates a signature sig, a vector of coupled hash values. It generates consistent and uniform hash values (j^*, w_{j^*}) by directly sampling a pair of active indices, $(w_j,\ z_j)$, for the j-th weight, where $0 \leq j \leq p - 1$. Since ICWS does not generate any extra element other than two active indices per weight, the size of the universal set remains the same.

The active index w_j is the quantization of weight s_j, satisfying $w_j \leq s_j$. Another active index z_j is obtained by transforming the first active index w_j. z_j satisfies $z_j \geq s_j$. Therefore, these two active indices surround weight s_j. ICWS satisfies the LSH property i.e.

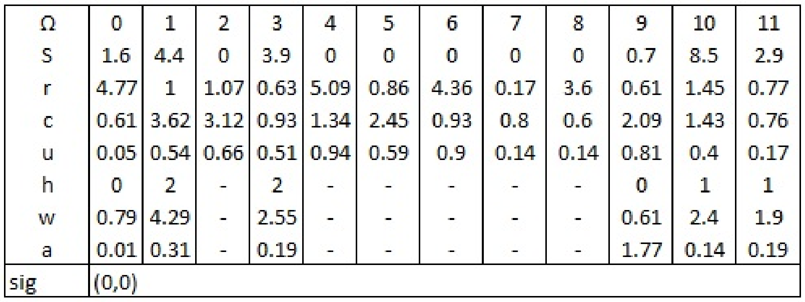
$$Pr(sig^S = sig^T) = GJS(S, T)$$

, where sig^S and sig^T are the signatures of S and T respectively. Eq. (2) is proposed to generate the first active index, w_j, uniformly at random in $[0, s_j]$.

$$\ln w_j = \ln s_j - r_j * u_j \tag{2}$$

, where $r \sim Gamma(2, 1)$ and $u \sim Uniform(0, 1)$. For consistency, instead of Eq. (2), ICWS samples w_j using the following equation: $\ln w_j = r_{j^*}(\lfloor \frac{\ln s_j}{r_j} + u_j \rfloor - u_j)$. The second active index z_j is generated using Eq. (3) uniformly at random in $[s_j, \infty)$. w_j and z_j are independent of each other since w_j and z_j are generated from uniform distributions of different intervals.

$$r_j = \ln z_j - \ln w_j \tag{3}$$



Ω	0	1	2	3	4	5	6	7	8	9	10	11
S	1.6	4.4	0	3.9	0	0	0	0	0	0.7	8.5	2.9
r	4.77	1	1.07	0.63	5.09	0.86	4.36	0.17	3.6	0.61	1.45	0.77
c	0.61	3.62	3.12	0.93	1.34	2.45	0.93	0.8	0.6	2.09	1.43	0.76
u	0.05	0.54	0.66	0.51	0.94	0.59	0.9	0.14	0.14	0.81	0.4	0.17
h	0	2	-	2	-	-	-	-	-	0	1	1
w	0.79	4.29	-	2.55	-	-	-	-	-	0.61	2.4	1.9
a	0.01	0.31	-	0.19	-	-	-	-	-	1.77	0.14	0.19
sig	(0,0)											

Fig. 1. An example that shows how ICWS generates a coupled hash value (0,0) for the input weighted set S of size 12.

In order to select j with a probability proportional to s_j, it uses the following equation: $a_j = c_j/z_j$. This equation can be expanded to Eq. (4).

$$a_j = \frac{c_j}{w_j * exp(r_j)} \tag{4}$$

, where $c \sim Gamma(2,1)$ and $a \sim Exp(1)$. This allows ICWS to exploit the important property of Exponential distribution, $P(a_{j^*} = min(a_0, a_1, \dots, a_{p-1})) = \frac{s_{j^*}}{\sum_j s_j}$, when selecting j^*. Finally, a coupled hash value, (j^*, w_{j^*}), is generated. When ICWS generates the next hash value, it needs to produce as many random numbers as it produced for the previous hash value. ICWS is summarized in Algorithm 1.

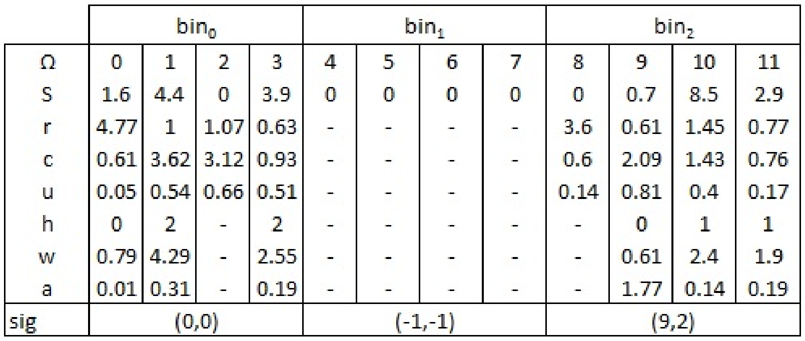
Estimated Generalized Jaccard Similarity ($EGJS$) between two weighted sets, S and T, is defined as the proportion of collisions between their signatures in Eq. (5).

$$EGJS(S,T) = \frac{\sum_{j=0}^{nh-1} \mathbb{1}(sig_j^S = sig_j^T)}{nh} \tag{5}$$

The example in Fig. 1 illustrates how ICWS generates a coupled hash value, (0,0). S denotes an input weighted set. To obtain (0,0), three random variables, r, c, and u are generated per weight, which means $3*p$ random numbers in total. If the universal set of a dataset is large, ICWS needs to generate a huge amount of random numbers, taking a long execution time.

3 BINNED CONSISTENT WEIGHTED SAMPLING (BCWS)

In order to improve the efficiency of signature generation for weighted sets, we suggest Binned CWS (BCWS), a novel method that generates a coupled hash value per bin.



	bin₀				bin₁				bin₂			
Ω	0	1	2	3	4	5	6	7	8	9	10	11
S	1.6	4.4	0	3.9	0	0	0	0	0	0.7	8.5	2.9
r	4.77	1	1.07	0.63	-	-	-	-	3.6	0.61	1.45	0.77
c	0.61	3.62	3.12	0.93	-	-	-	-	0.6	2.09	1.43	0.76
u	0.05	0.54	0.66	0.51	-	-	-	-	0.14	0.81	0.4	0.17
h	0	2	-	2	-	-	-	-	-	0	1	1
w	0.79	4.29	-	2.55	-	-	-	-	-	0.61	2.4	1.9
a	0.01	0.31	-	0.19	-	-	-	-	-	1.77	0.14	0.19
sig	(0,0)				(-1,-1)				(9,2)			

Fig. 2. An example that shows how BCWS splits Ω into 3 bins with $lbin = 4$ and generates a signature of size 3, $[(0,0),(-1,-1),(9,2)]$, for the input weighted set S of size 12. The bin_1 is an empty bin, so its hash value is initialized to (-1,-1).

3.1 BCWS

Let us consider a weighted set $S = \{s_0, s_1, s_2, \ldots s_{p-1}\}$ and the universal set $\Omega = \{0, 1, 2, \ldots p-1\}$ of size p. We create bins by partitioning Ω into $nbin$ number of bins, and the size of a bin, $lbin$, is equivalent to $\lfloor p/nbin \rfloor$. We generate one coupled hash value per bin.

In the i-th bin bin_i, three random numbers, r, c, and u, are generated per weight as described in Algorithm 2. Therefore, BCWS needs $3 * lbin$ random numbers to generate each hash value, whereas ICWS uses $3{*}|\Omega|$. After generating $3 {*} lbin$ random numbers, we compute w_j, z_j and a_j as described in Algorithm 2. Then, we select the index j^* that satisfies $j^* = argmin_j \ a_j$. Finally, BCWS returns $(j^* + binidx, w_{j^*})$ as the hash value for bin_i, where $binidx$ is the index of the first element of the corresponding bin. After generating hash values for all the bins, we combine them to get the signature vector of S, sig^S. Notice that the number of bins, $nbin$, is equivalent to the number of hash values, nh, since one hash value is generated per bin. Because of the uniqueness of hash values of each bin, our densification method in Section 3.2 can satisfy the LSH property.

Given a collection of weighted sets, it is very important to generate the i-th hash value from bin_i consistently across weighted sets to achieve high approximation accuracy. To ensure consistency, the same seed is used when generating random numbers for the i-th hash value for bin_i of every weighted set.

3.2 Densification of Sparse Signatures

In the example in Fig. 2, we can see that all of the weights of bin_1 of S are O's. For bins such as bin_1, h, w and a cannot be defined. We refer a bin that has no positive weight as an empty bin.

Empty bins have a negative impact on the approximation accuracy of similarity between two weighted sets if we assign the default hash value, $(-1,-1)$, to

Algorithm 2 : Binned Consistent Weighted Sampling

```
Input: weighted set $S = \{s_0, s_1, s_2, \ldots, s_{p-1}\}$, number of bins, $nbin$, random seeds, $seed$
Output: signature of $S$, $sig$
 1: $lbin = \lfloor p/nbin \rfloor$
 2: $binidx = 0$
 3: for $i = 0, 1, \ldots, nbin - 1$ do
 4:     $bin = [s_{binidx}, s_{binidx+1}, \ldots, s_{binidx+lbin-1}]$
 5:     if $empty\_bin(bin) = true$ then
 6:         $sig_{i,0} = -1$
 7:         $sig_{i,1} = -1$
 8:         $binidx = binidx + lbin$
 9:         $continue$
10:     end if
11:     $set\ seed_i$
12:     for $j = 0, 1, \ldots, lbin - 1$ $such\ that\ bin_j > 0$ do
13:         $r_j \sim Gamma(2, 1)$
14:         $c_j \sim Gamma(2, 1)$
15:         $u_j \sim Uniform(0, 1)$
16:         $h_j = \lfloor \ln(bin_j)/r_j + u_j \rfloor$
17:         $w_j = exp(r_j(h_j - u_j))$
18:         $a_j = c_j/(w_j * exp(r_j))$
19:     end for
20:     $j^* = argmin_j\ a_j$
21:     $sig_{i,0} = j^* + binidx$
22:     $sig_{i,1} = w_{j^*}$
23:     $binidx = binidx + lbin$
24: end for
25: $sig = densify(sig, nbin, seed)$ in Algorithm 3
26: return $sig$
```

each empty bin. Let us denote bin_i of S by bin_i^S. Suppose that bin_i^S and bin_i^T are empty. If both of them are assigned the default hash value of $(-1, -1)$, sig_i^S and sig_i^T always collide, increasing $EGJS(S, T)$ regardless of the actual similarity between S and T as discussed in [17, 16].

In order to assign hash values to empty bins more effectively, we propose a simple and efficient densification method that uses Rejection Sampling [5]. We generate a uniformly distributed discrete random number, x, in $[0, nbin)$. If bin_x is a non-empty bin, we assign sig_x as the hash value of the empty bin. If an empty bin is sampled, we reject and generate another discrete random number as x and sample another bin. We keep sampling bins until a non-empty bin is sampled.

For better approximation accuracy, this sampling also needs to be performed consistently. By using the same seed to densify bin_i's of different weighted sets, we maintain the cosistency of BCWS. Details of our densification method are given in Algorithm 3.

Algorithm 3 : Densification of Sparse Signatures of BCWS

```
Input: a sparse signature generated by BCWS, $sig$, number of bins $nbin$, random seeds, $seed$
Output: a densified signature, $sig$
 1: for $j = 0, 1, \ldots, nbin - 1$ do
 2:     if $sig_j$ is the default hash value then
 3:         $set\ seed_j$
 4:         while $true$ do
 5:             $x \sim DiscreteUniform(0, nbin - 1)$
 6:             if $sig_x$ is not the default hash value then
 7:                 $sig_{j,0} = sig_{x,0}$
 8:                 $sig_{j,1} = sig_{x,1}$
 9:                 $break$
10:             end if
11:         end while
12:     end if
13: end for
14: return $sig$
```

Claim. A densified signature of BCWS satisfies the LSH property.

Proof. The densification of sparse signatures is done consistently by using the same seed when we densify bin_i's of different weighted sets.

Let us suppose that both bin_i^S and bin_i^T are empty. Now, while densifying sig^S, bin_j is sampled to densify bin_i^S by Algorithm 3. By consistency, bin_j would be sampled to densify bin_i^S too. For a randomly selected bin_j, if both bin_j^S and bin_j^T are non-empty, the probability that $sig_j^S = sig_j^T$ is $GJS(S,T)$. If bin_j^S is empty and bin_j^T is non-empty, $sig_i^T \leftarrow sig_j^T$. To densify bin_i^S, Algorithm 3 keeps sampling bins until a non-empty bin bin_k^S is found, and $sig_i^S \leftarrow sig_k^S$ where $k \neq j$. Since hash values from different bins cannot collide, $sig_i^S \neq sig_i^T$. We can apply the same reasoning for the case in which bin_j^S is non-empty and bin_j^T is empty.

If bin_i^S is empty while bin_i^T is non-empty, we densify only bin_i^S. Let us assume that the non-empty bin sampled by Algorithm 3 for bin_i^S is bin_j^S where $i \neq j$. Then, sig_i^S and sig_i^T cannot collide since they are from different bins.

If both bin_i^S and bin_i^T are non-empty, their hash values satisfy the LSH property since BCWS complies with ICWS for non-empty bins and ICWS satisfies the LSH property.

Overall, the hash values of the i-th bin, sig_i^S and sig_i^T, satisfy the LSH property, i.e., $P(sig_i^S = sig_i^T) = GJS(S,T)$.

$$P(sig^S = sig^T) = \frac{\sum_{j=0}^{nh-1} \mathbb{1}(sig_j^S = sig_j^T)}{nh} = \sum_{j=0}^{nh-1} P(sig_i^S = sig_i^T) = GJS(S,T) \tag{6}$$

Therefore, $P(sig^S = sig^T) = GJS(S,T)$.

Table 1. Datasets for our experiments

Dataset	# Training data	# Test data	Dimension
News20	800	200	197415
MNIST	1000	500	784
StarLightCurves	900	300	1024
Kdd2010	800	200	8764
Rcv1	800	200	10635

3.3 Complexity

For both ICWS and BCWS, nh random seeds are stored to generate nh hash values. Although we use the same set of seeds for weighted sets, we also need to store random numbers to generate each hash value. Therefore, the space complexity of ICWS is $O(nh + 3 * |\Omega|)$, whereas BCWS uses $O(nh + 3 * lbin)$ space. Since $lbin << |\Omega|$, the space complexity of BCWS is much less than that of ICWS.

The time complexity of BCWS, $O(lbin * nh)$, is also much less than that of ICWS, $O(|\Omega| * nh)$, since $lbin << |\Omega|$. In fact, since $|\Omega| \approx lbin * nbin$ and $nbin = nh$, the time complexity of BCWS is $O(|\Omega|)$. This means that the execution time of BCWS remains the same even if it needs to generate more hash values.

The time complexity of our densification method in Algorithm 3 depends on the number of empty bins.

Therefore, our approach can generate signatures of weighted sets more efficiently than ICWS. We also want to mention that BCWS is naturally parallelizable, and thus can be executed much faster using GPUs or multicore processors.

4 EXPERIMENTAL EVALUATION

We compare our method, BCWS, with three state-of-the-art CWS algorithms, ICWS [11], PCWS [19] and CCWS [18] using two important tasks, the GJS estimation and the 1-NN classification on five real datasets. The comparison results using the GJS estimation task is presented in Section 4.1. In Section 4.2, we compare BCWS with other methods using the 1-NN classification task.

Experimental Setup Experiments were performed with $nbin \in \{32, 64, 128, 256, 512\}$. We store $nbin$ seeds to generate nh hash values, where $nbin = nh$. The information on the datasets used for our experiments is provided in Table 1. Since other methods are too slow to generate signatures for a large dataset, we reduced the size of test data by random selection. The dimension represents the size of the universal set of each dataset. Experiments were performed on a desktop computer with Intel i7 3.60GHz CPU and 32GB RAM.

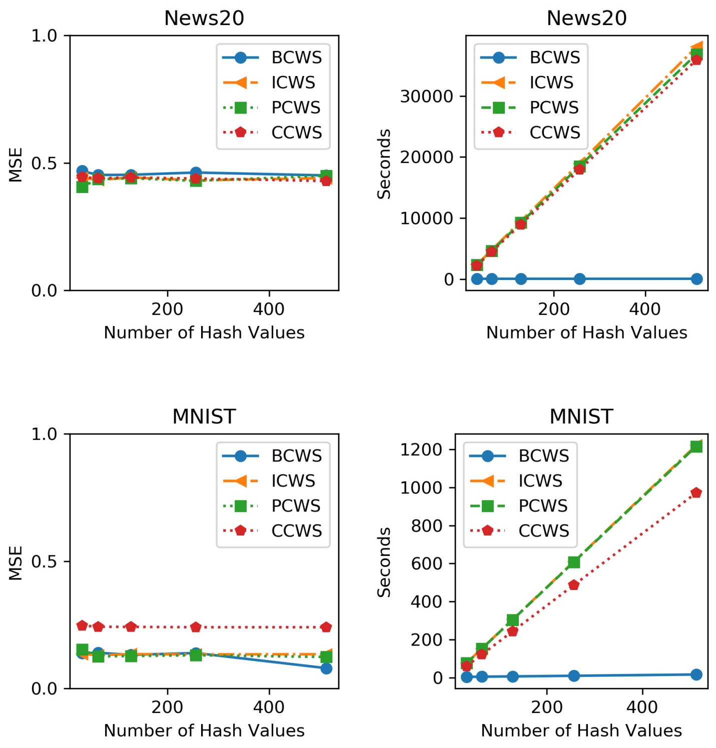


Fig. 3. The average MSE (left column) and the average execution time for signature generation (right column) based on 5 executions.

4.1 Comparison using GJS Estimation Task

We present the average approximation accuracy and the average execution time for hash generation using two real datasets, News20[1] and MNIST[2]. MSE (Mean Squared Error) measures the difference between EGJS in Eq. (5) and the actual GJS defined in Eq. (1).

Discussion on Results: In Fig. 3, all four methods show similar MSE values on both datasets. This means that the signatures generated by BCWS are as good as those by other methods in approximating the actual GJS between weighted sets. We can see that on MNIST dataset, the MSE of CCWS is higher than others. It is because the active indices generated by CCWS are very sensitive to a slight change of weights and thus highly similar weights may be transformed into different hash values.

[1] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
[2] http://yann.lecun.com/exdb/mnist/

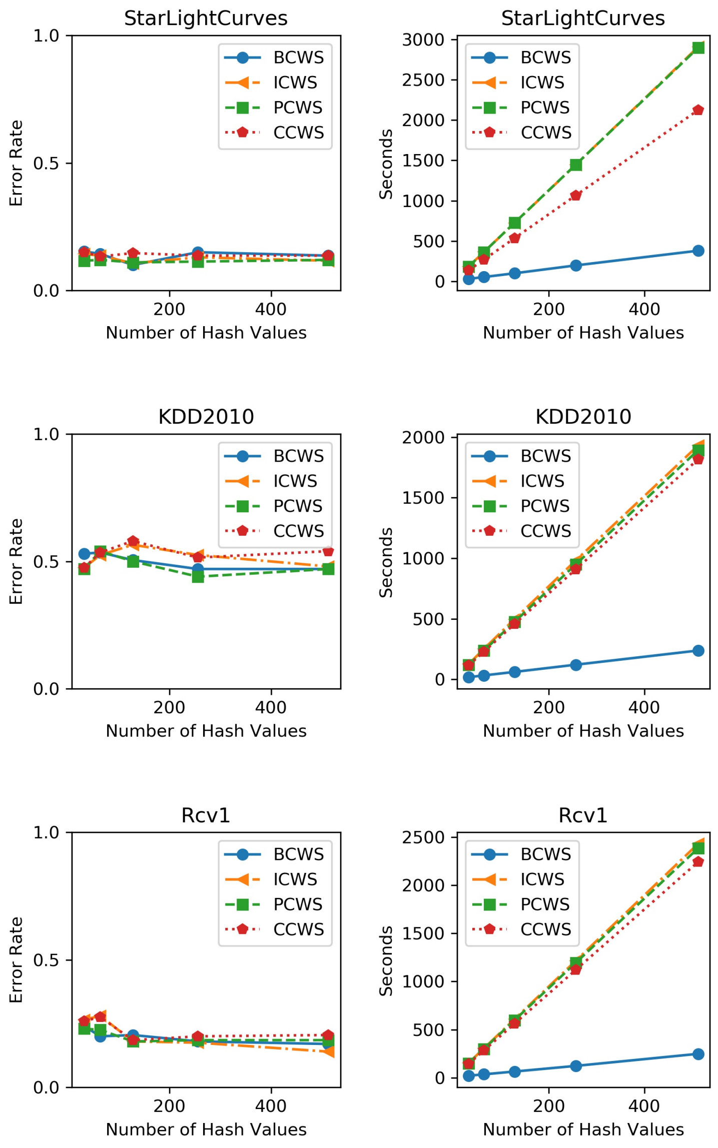


Fig. 4. The average classification error rate (left column) and the average runtime (right column) with the number of hash values of three real datasets based on 5 executions.

In terms of execution time for hash generation, BCWS clearly outperforms all the other methods on both datasets. Especially, when generating signatures of size $nh = 512$ on News20 dataset, BCWS is almost 700 times faster than the other methods. Furthermore, the execution time of BCWS remains the same when it generates more hash values. It is because the time complexity of BCWS is $O(|\Omega|)$, where $|\Omega|$ is the size of the universal set.

4.2 Comparison using 1-NN Classification Task

We present the average classification accuracy and the average total execution time for hash generation and 1-NN classification using three real datasets, Rcv1[3], StarLightCurves [6] and KDD2010[4]. Among given training instances, we find the most similar weighted set for a given query weighted set in test data. Then, we classify the query as the class of its 1-NN and compute the classification error rate as the proportion of the misclassified weighted sets in test data.

Discussion on the Results: In Fig. 4, all the four methods show similar classification accuracy on all the three datasets just as we observed in the results of the GJS estimation. Therefore, the signatures generated by BCWS is as good as those by other methods in performing the 1-NN classification of weighted sets.

In terms of execution time for hash generation and 1-NN classification, BCWS clearly outperforms all the other methods on all the datasets. We can see that the overall execution time of BCWS is increasing as the number of hash values grows. This is because while the hash generation time of BCWS is just $O(|\Omega|)$, it computes EGJS between signatures of a query and a weighted set in training data to find the 1-NN of the query. The execution time for such similarity search is proportional to the dimension of signatures, i.e., the number of hash values.

5 CONCLUSION

We proposed BCWS, an efficient method for generating signatures for weighted sets by partitioning the universal set into bins and densifying sparse signatures. BCWS generates consistent and uniform hash values by using less random numbers than existing methods.

We evaluated the performance of BCWS through comparison experiments using two tasks, the GJS estimation and the 1-NN classification, on five real datasets. Experimental results showed that BCWS outperformed existing methods in terms of execution time while maintaining the approximation accuracy. In particular, the execution time for BCWS remains constant for different number of hash values. Therefore, our approach can be used for efficient analysis of large-scale high dimensional datasets of weighted sets.

References

1. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on. pp. 459–468. IEEE (2006)

[3] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
[4] https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/

2. Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I., Schmidt, L.: Practical and optimal lsh for angular distance. In: Advances in Neural Information Processing Systems. pp. 1225–1233 (2015)
3. Broder, A.Z.: On the resemblance and containment of documents. In: Compression and Complexity of Sequences 1997. Proceedings. pp. 21–29. IEEE (1997)
4. Broder, A.Z., Charikar, M., Frieze, A.M., Mitzenmacher, M.: Min-wise independent permutations. Journal of Computer and System Sciences **60**(3), 630–659 (2000)
5. Casella, G., Robert, C.P., Wells, M.T.: Generalized accept-reject sampling schemes. Lecture Notes-Monograph Series pp. 342–347 (2004)
6. Chen, Y., Keogh, E., Hu, B., Begum, N., Bagnall, A., Mueen, A., Batista, G.: The ucr time series classification archive (July 2015), `www.cs.ucr.edu/~eamonn/time_series_data/`
7. Chum, O., Philbin, J., Zisserman, A., et al.: Near duplicate image detection: min-hash and tf-idf weighting. In: BMVC. vol. 810, pp. 812–815 (2008)
8. Gollapudi, S., Panigrahy, R.: Exploiting asymmetry in hierarchical topic extraction. In: Proceedings of the 15th ACM international conference on Information and knowledge management. pp. 475–482. ACM (2006)
9. Haeupler, B., Manasse, M., Talwar, K.: Consistent weighted sampling made fast, small, and easy. arXiv preprint arXiv:1410.4266 (2014)
10. HAVELIWALA, T.: Scalable techniques for clustering the web. In: Proc. of the Third International Workshop on the Web and Databases (Informal Proceedings), 2000 (2000)
11. Ioffe, S.: Improved consistent sampling, weighted minhash and l1 sketching. In: Data Mining (ICDM), 2010 IEEE 10th International Conference on. pp. 246–255. IEEE (2010)
12. Ke, Q., Isard, M.A., Lee, D.C.: Partition min-hash for partial-duplicate image determination (May 28 2013), uS Patent 8,452,106
13. Li, P.: 0-bit consistent weighted sampling. In: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 665–674. ACM (2015)
14. Manasse, M., McSherry, F., Talwar, K.: Consistent weighted sampling. Unpublished technical report) http://research. microsoft. com/en-us/people/manasse **2** (2010)
15. Shrivastava, A.: Simple and efficient weighted minwise hashing. In: Advances in Neural Information Processing Systems. pp. 1498–1506 (2016)
16. Shrivastava, A.: Optimal densification for fast and accurate minwise hashing. arXiv preprint arXiv:1703.04664 (2017)
17. Shrivastava, A., Li, P.: Improved densification of one permutation hashing. arXiv preprint arXiv:1406.4784 (2014)
18. Wu, W., Li, B., Chen, L., Zhang, C.: Canonical consistent weighted sampling for real-value weighted min-hash. In: Data Mining (ICDM), 2016 IEEE 16th International Conference on. pp. 1287–1292. IEEE (2016)
19. Wu, W., Li, B., Chen, L., Zhang, C.: Consistent weighted sampling made more practical. In: Proceedings of the 26th International Conference on World Wide Web. pp. 1035–1043. International World Wide Web Conferences Steering Committee (2017)