Subspace Outlier Detection in Linear Time with Randomized Hashing

Saket Sathe
Research Staff Member
IBM T.J. Watson Research Center

Charu C. Aggarwal
Distinguished Research Staff Member
IBM T.J. Watson Research Center

Motivation – Fast and simple subspace outlier detection

- **Simplicity:** Simple to implement $\Longrightarrow$ requires few lines of code to implement
- **Accuracy:** Accurate for high-dimensional data sets
- **Linear Complexity:** Linear in the number of points $\Longrightarrow$ orders of magnitude speedup
- **Constant Memory:** Constant memory $\Longrightarrow$ memory requirement does not change with the number of points
- **Stream Compatibility:** Works for data streams $\Longrightarrow$ on-the-fly outlier scoring is possible

RS-Hash: Randomized Subspace Hashing

RS-Hash is ensemble centric $\Longrightarrow$

- It is comprised of several weak outlier detectors
- Each individual detector is extremely weak, but the overall performance is extraordinarily good.

How does it work?

- **Input:** Data set D with n points and d dimensions. The ith row is a d-dimensional vector denoted by $\overline{X}_i = (x_{i1}...x_{id})$.
- **Output:** Outlier score for each point in D.

Steps:

1. Sample s points from the data set D.
2. Learn a training model on these s points
3. Score all the data points using this training model
4. Perform steps 1-3 for m times and report the average score

RS-Hash: Randomized Subspace Hashing

Steps for A Single Ensemble Component

1. Select a value of the locality parameter f randomly between $(1/\sqrt{s}, 1 - 1/\sqrt{s})$.

2. Generate a random vector $(\alpha_1 \ldots \alpha_d)$, where α_i is drawn randomly from $(0, f)$. α_i is referred to as the shift parameter.

3. Select an integer r randomly between $1 + 0.5 \cdot [\log_{\max\{2,1/f\}}(s)]$ and $\log_{\max\{2,1/f\}}(s)$. Select r dimensions and denote this set by V.

4. Draw a training sample S of s points and determine the min and max values (min_j and max_j) for each dimension j

5. Normalize each $\overline{X_i}$ in the sample to $\overline{X_i'}$ as follows:

$$x'_{ij} \Leftarrow \frac{x_{ij} - min_j}{max_j - min_j} \tag{1}$$

RS-Hash: Randomized Subspace Hashing

6. For each $\overline{X'_i}$ create its *integer representation* $\overline{Y}_i$ such that:

- if $j \notin V$ set y_{ij} to -1
- if $j \in V$ set y_{ij} to the integer value of $\lfloor (x'_{ij} + \alpha_j)/f \rfloor$.

8. Apply w different hash functions $h_1(\overline{Y}_i) \ldots h_w(\overline{Y}_i)$ to $\overline{Y}_i$. Increment the $h_k(\overline{Y}_i)$th element of the kth hash table by 1.

RS-Hash: Randomized Subspace Hashing

6. For each $\overline{X'_i}$ create its *integer representation* $\overline{Y}_i$ such that:
   - if $j \notin V$ set y_{ij} to -1
   - if $j \in V$ set y_{ij} to the integer value of $\lfloor (x'_{ij} + \alpha_j)/f \rfloor$.

8. Apply w different hash functions $h_1(\overline{Y}_i) \ldots h_w(\overline{Y}_i)$ to $\overline{Y}_i$. Increment the $h_k(\overline{Y}_i)$th element of the kth hash table by 1.

Scoring step:

9. Transform each point $\overline{X} \in D$ to $\overline{Y}$ using the same steps as before. Apply the same w hash function to points $\overline{Y}$.

10. Let the value of the $h_k(\overline{Y})$th cell in the kth hash table be c_k
    - If $\overline{Y} \in S$ report $\log_2(\min\{c_1 \ldots c_w\})$ as the outlier score;
    - Otherwise, report $\log_2(\min\{c_1 \ldots c_w\}+1)$

Implementing the Hash Functions

RS-Hash(S) $\rightarrow$ Streaming Variant $\rightarrow$ Count-min Sketch

- Insert: $\overline{Y}$ is hashed w times. Each hashed output maps to a bucket ID between $(0, p-1)$, whose count is incremented by 1.
- p is the size of hash table and is known as the hash range and is selected based on memory availability
- Lookup: $\overline{Y}$ is again hashed w times. All counts are retrieved and the minimum is returned.
- Typical values of p and w are $p = 10,000$ and $w = 4$
- Collisions occur with very low probability, and (theoretically) any number of elements can be inserted

RS-Hash(E) $\rightarrow$ Static Variant $\rightarrow$ Hash Table with Linear Probing

- This is the standard hash table with linear probing.

Why does the Approach Work?

$$\text{Log-Fit}(\overline{Y}) = \log(\rho(\overline{Y})) = \log\left(\frac{1+\min\{c_1 \ldots c_w\}}{1+n}\right)$$

- Overall score of *RS-Hash* can be written as

$$\text{Score}(\overline{Y}) = \text{Const.} + \text{AVG}_{[\text{All } V]} \log\left(1+\min\{c_1 \ldots c_w\}\right)$$

- *RS-Hash* computes ensemble-centric average log-likelihood fits over different base components

Logic of Parameter Choices

- $f \in [1/\sqrt{s}, (1-1/\sqrt{s})]$. Ensures that larger grids are explored when there are fewer points
- The dimensionality r of the subspace explored is at least 2. This is because the value of r is at least $1 + 0.5\log_{\max\{2,1/f\}}(s) \geq 1 + 0.5\log_{\sqrt{s}}(s) = 2$.
- The dimensionality selected is such that the expected number of points in the local region corresponding to each test point varies between 1 and $\sqrt{s}$.

Experiments – Data sets

Setup: All data sets are from the UCI Machine Learning Repository. Ensemble components $m = 300$, hash tables $w = 4$, hash range $p = 10,000$

Data set	Points	Attribute	Percent Outliers (%)
Lympho	148	18	3.4
Ecoli	336	7	2.7
Yeast	1,364	8	4.8
Cardio	1,831	21	9.6
Musk	3,062	166	3.1
Waveform	3,509	21	4.7
Optdigits	5,216	64	2.9
KDDCup99	25,000	41	0.7

Baselines: LoF [1], FastABOD [2], HiCS [3], iForest [4], and AvgKNN [5]

[1] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. LOF: identifying density-based local outliers. SIGMOD 2000.

[2] H.-P. Kriegel, M. Schubert, and A. Zimek. Angle-based Outlier Detection in High-Dimensional Data. KDD 2008.

[3] F. Keller, E. Muller, K. Bohm. HiCS: High-Contrast Subspaces for Density-based Outlier Ranking. ICDE 2012.

[4] Liu, F. T., Ting, K. M., Zhou, Z. H. Isolation forest. ICDM 2008.

[5] Angiulli, F., Pizzuti, C. Fast outlier detection in high dimensional spaces. PKDD 2002.

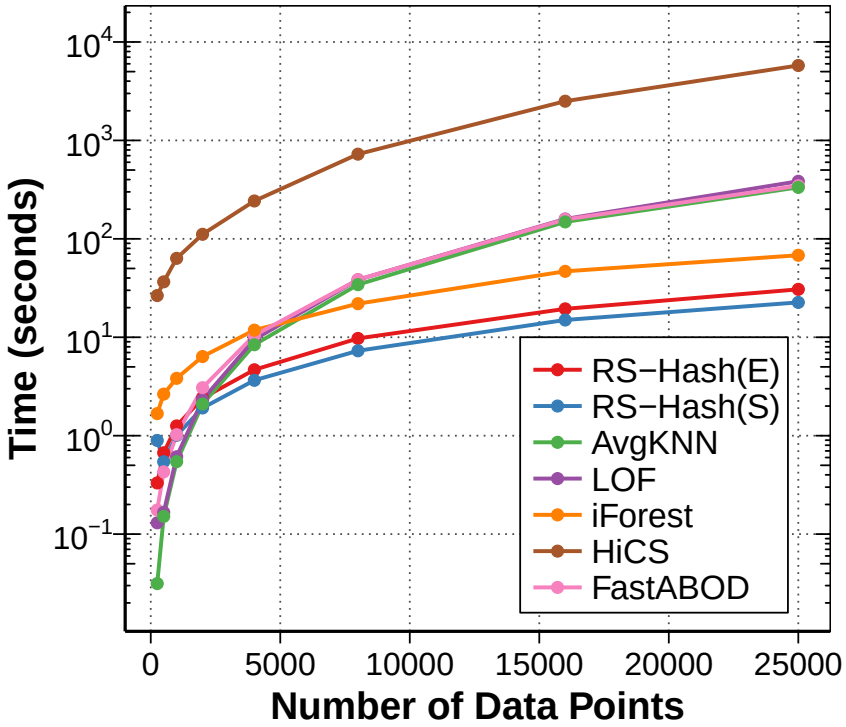
Experiments – Area under curve (AUC)

Data set	RS-Hash(E)	RS-Hash(S)	AvgKNN	LOF	iForest	HiCS	FastABOD
Lympho	**100.0**	**99.85**	98.16	97.18	99.71	86.78	52.32
Ecoli	**88.41**	**88.44**	87.62	86.29	85.31	73.72	84.66
Cardio	**91.61**	**91.78**	70.47	59.67	91.52	53.04	58.74
Musk	**100.0**	**100.0**	24.48	39.99	**100.0**	47.77	23.95
Optdigits	**76.04**	**76.14**	39.59	61.54	72.66	39.05	72.07
Yeast	**80.87**	**80.80**	66.47	55.06	79.44	61.23	66.91
Waveform	**74.42**	**73.85**	66.98	61.08	72.47	62.61	58.91
KDDCup99	**99.97**	**99.98**	13.6	46.43	**99.98**	79.94	13.71

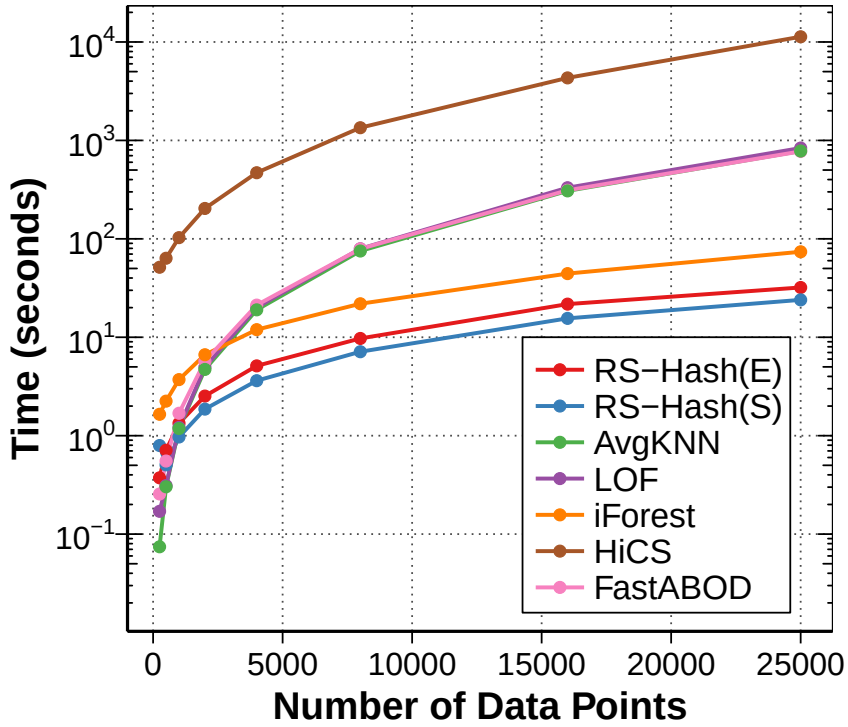
* top-2 methods are shown in boldface

- Highest average improvement of 53% is observed in Musk
- Musk is a very high dimensional data set, which demonstrates the effectiveness of randomized feature selection strategy of *RS-Hash*.
- *RS-Hash* consistently improves on the baselines

Experiments – Efficiency Analysis



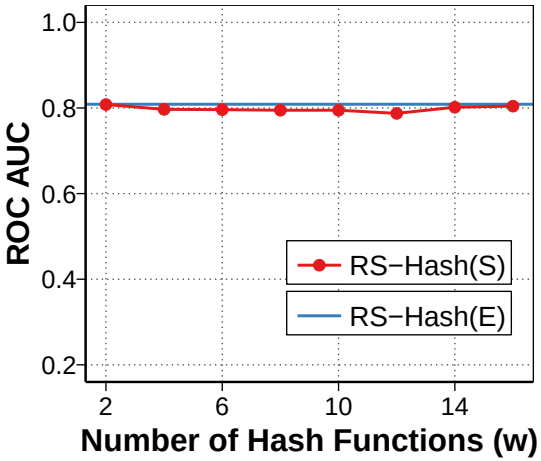
(a) KDDCup99



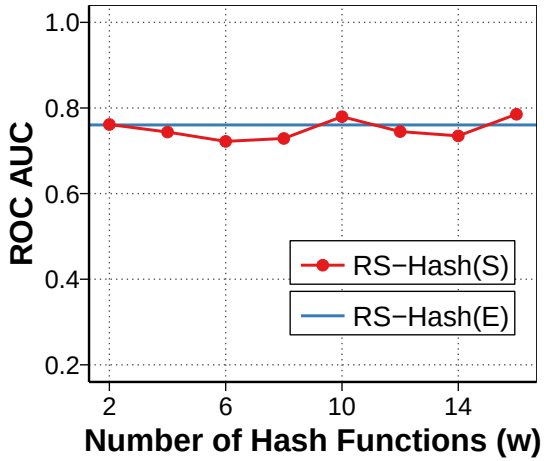
(b) NORMAL(0,1)

- *RS-Hash(S)* is between 20 and 100 times faster than LOF for KDDCup99
- *RS-Hash* is nearly 400 times faster than HiCS at this data size.
- The difference between *RS-Hash* and baselines only increases with data size

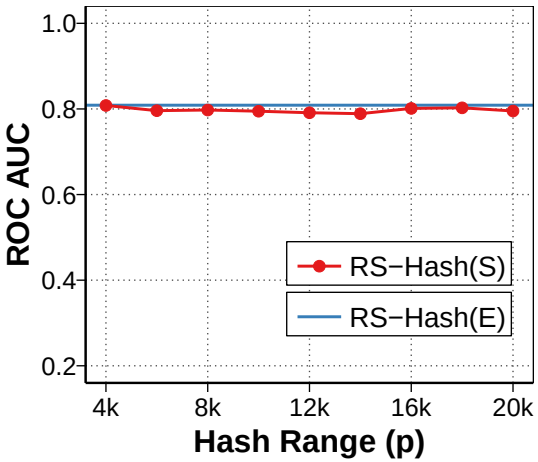
Experiments – Parameter Sensitivity



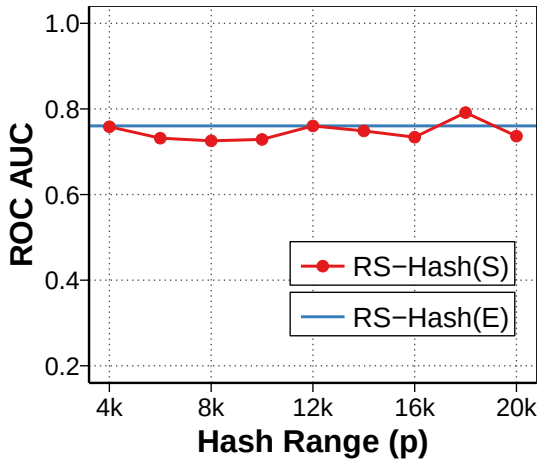
(a) YEAST



(b) OPTDIGITS



(c) YEAST



(d) OPTDIGITS

Additional experiments on parameter sensitivity analysis and the streaming version *RS-Hash(S)* can be found in the paper.

Summary and Conclusion

- We discussed the randomized hashing-based outlier detector
- The discussed methods are simple and easy to implement and only rely on hash tables
- We discussed how to effectively use the min-hash data structure with *RS-Hash*
- Demonstrated accuracy and efficiency of *RS-Hash* with extensive experiments on real data sets

Summary and Conclusion

- We discussed the randomized hashing-based outlier detector
- The discussed methods are simple and easy to implement and only rely on hash tables
- We discussed how to effectively use the min-hash data structure with *RS-Hash*
- Demonstrated accuracy and efficiency of *RS-Hash* with extensive experiments on real data sets

Thank You.

We are hiring!

Backup

Extensions to Data Streams

Instead initializing each hash function one after another, we initialize them simultaneously.

- Time-decayed scores: The weight of each point is reduced after the arrival of an additional data point
- It becomes $2^{-\lambda t}$ after t points have arrived $\implies$ half-life $= 1/\lambda$
- The values of min_j and max_j are initialized on using an initial sample of data
- Since the data set size is not fixed, we determine $s = \max\{1000, \sum_i 2^{-\lambda i}\} = \max 1000, \frac{1}{1-2^{-\lambda}}$

Extensions to Data Streams

Instead initializing each hash function one after another, we initialize them simultaneously.

- Time-decayed scores: The weight of each point is reduced after the arrival of an additional data point
- It becomes $2^{-\lambda t}$ after t points have arrived $\implies$ half-life $= 1/\lambda$
- The values of min_j and max_j are initialized on using an initial sample of data
- Since the data set size is not fixed, we determine $s = \max\{1000, \sum_i 2^{-\lambda i}\} = \max 1000, \frac{1}{1-2^{-\lambda}}$
- Access: count $* 2^{-\lambda(t_c - t_l)}$
- Update: new count $= 2^{-\lambda(t_c - t_l)} *$ old count $+ 1$
- t_c current time stamp, t_l last time stamp