JMLR: Workshop and Conference Proceedings 41:65–80, 2015 BIGMINE 2015

Random Decision Hashing for Massive Data Learning

Xiatian Zhang XIATIAN.ZHANG@TENDCLOUD.COM
TendCloud Ltd., Beijing, China

Wei Fan WEIFAN@GMAIL.COM
Baidu Research Big Data Labs, Sunnyvale, CA., USA

Nan Du DUNAN@GATECH.EDU
Georgia Institute of Technology, Atlanta, GA., USA

Editors: Wei Fan, Albert Bifet, Qiang Yang and Philip Yu

Abstract

In the era of Big Data, the iterative nature of most traditional learning algorithms renders them increasingly inefficient to address large learning problems. Random decision trees algorithm is an efficient and decent learning algorithm, but the complexity of tree structure makes the algorithm inefficient for the big data problems. Inspired by the theoretical analyses of random decision trees, we propose a randomized algorithm for big data classification tasks based on unsupervised locality sensitive hashing. Our algorithm is essentially non-iterative, very flexible to deploy over clusters of machines, and thus able to handle large datasets efficiently. Experiments on real datasets demonstrate that the proposed algorithm can easily scale up to millions of data samples and features while still achieves at most 17% and 800% improvement in accuracy and efficiency respectively with moderate memory consumption over existing algorithms.

Keywords: Classification, Randomization

1. Introduction

Classification is a critical task of machine learning, there are many algorithms proposed to solve the problem. In the Big Data era, the new Moore's law Economist (2010) indicates that the amount of digital information grows 10 times every 5-year. As a consequence, mainstream approaches pervasively deployed in practice, such as logistic regression, decision tree, support vector machine, etc., have been facing increasing challenges than ever before, for that their computations often involve successive iterations with super-linear time complexity, which renders them difficult to scale in a parallelized and distributed setting. Such iterative essence of computation is in fact the "Achilles heel" of these classic algorithms when applied to large-scale data. As the volume of the dataset becomes large, the in-memory iterations are limited by the available memory space, while the out-memory iterations are rather inefficient to sustain.

Random Decision Trees (RDT) Fan et al. (2003) is a fast randomized algorithm adaptable to many different applications Fan et al. (2006b); Zhang and Fan (2008); Zhang et al. (2008). By simply building trees to randomly partition the data instances into leaf nodes and memorizing the respective class frequencies, RDT has robust performance and runs much faster than classic methods Fan et al. (2003); Zhang et al. (2008). However, the complexity of maintaining tree structures greatly restricts the scalability of RDT to be directly applied to large data. More specifically, a popular way to implement RDT is to build several empty trees randomly Fan et al. (2003) and then use the training data to prune the trees by removing the branches without enough training samples. The problem is that the empty trees should keep all the possible branches before being pruned, so the whole tree structure has to expand dramatically as the depth of the tree increases, which can consume huge amount of memory, and significantly restrict the possible maximum depth of the tree for the large scale data. Another mainstream method of RDT builds the tree according to the training dataset level by level Fan (2004), that is, the method prunes the tree as the tree grows. Unfortunately, in this way, we have to iterate the training dataset multiple times, which significantly slows down the training process for large scale problems where the training dataset is often too big to be loaded into the memory. Besides, even if the second method can be straightforwardly implemented on modern computational platforms, such as Hadoop Hadoop and Spark Spark, the tree structure is still difficult to maintain among the cluster nodes.

Inspired by the randomness design and the theoretical properties of RDT, we reveal that the 'black art' of RDT essentially does not lie in the tree structure but the underlying mechanism of random shattering. If we treat the tree in RDT as a way to shatter the data instances into different partitions, we can generalize the learning algorithm as follows:

- Repeatedly shatter the data instances into a set of random partitions.
- Within each partition, calculate the empirical distribution of class labels independently for each subset of the data.
- Average the class distributions of all the subsets that a data instance belong to from all the random partitions to make final class estimation

Essentially, the proposed random decision hashing is a kind of non-parametric estimation method, which could be more flexible compared to parametric models. Therefore, in order to design an efficient learning algorithm, the tree structures should be replaced with other simpler and lightweight shattering methods. Locality Sensitive Hashing (LSH) is widely used to solve large-scale problems in nearest neighborhood search Gionis et al. (1999); Broder (1997); Indyk and Motwani (1998), data indexing Bai et al. (2009) and dimension reduction Indyk and Motwani (1998). The key idea of LSH is simply to map similar data samples into the same hash code with high probability. As a consequence, in our proposed algorithm RDH, we apply LSH as an independent module to replace the tree-building component from RDT. However, one has to note that because many LSH methods Bergamo et al. (2011); Kulis and Darrell (2009); Liu et al. (2012); Norouzi et al. (2012); Norouzi and Fleet (2011); Salakhutdinov and Hinton (2009); Shakhnarovich et al. (2003); Weiss et al. (2008) must learn the hash functions to preserve 'semantic similarity', which makes extra computation costs, not all the LSH methods are our ideal candidates. Similar to the fact that RDT does not rely on a single tree with strong predictive power, RDH also does not depend on the 'semantic similarity' preservation capability of a single hash function. Instead, our only requirement is to roughly shatter the data instances evenly into the respective subspaces of a random partition. Therefore, we can simply turn to the classic unsupervised LSH Andoni and Indyk (2008); Broder (1997); Broder et al. (1998); Charikar

(2002); Immorlica et al. (2004); Gionis et al. (1999); Jain et al. (2010); Indyk and Motwani (1998) to meet our purpose. To achieve the best efficiency, we develop two variants to deal with the dense and sparse data based on the hyperplane hashing Charikar (2002) and SimHash Sadowski and Levin, respectively.

2. Theoretical Analyses of RDT

Random Decision Trees (RDT) algorithm is interesting in that it does not optimize any objective function, but instead, simply builds different trees randomly. That is out of the common sense of the conventional decision tree algorithm. Therefore, revealing the magic behind the good learning performance of RDT is first important to have a better understanding of the algorithm. In the original paper Fan et al. (2003) of RDT, Wei et al., discusses several heuristics to choose tree depth as well as the number of trees, based on the diversity and the probability of testing the relevant features in the worst case. Then Wei Fan (2004); Fan et al. (2005a) conducts large amount of empirical evaluations to investigate the accuracy under the 0-1 loss function and the cost-sensitive lost function. There are also some works to compare RDT with other randomized methods, such as Random Forest, Bagging, and Boosting, etc., by experiments Fan et al. (2005b, 2006a). Although these are all related work to verify the performance of RDT algorithm, there is still lack of a formal rigorous analysis, except that Zhang et al. (2008) tries to prove that the upper bound of the error rate of RDT will be stable, and the lower bound will decrease as the number of trees increases.

In this section, we analyze the estimation risk of RDT and reveal that the decision tree is not the key ingredient to enable the learning capability of random decision trees algorithm. We focus on the problem in continuous feature space. The analysis starts from univariate probability estimation problem, and then will be generalized to multivariate case. Before diving into the details, we first give the following formulations. Let x be the data instance, and y be the class. For a binary classification problem, if we can find the posterior distribution $P(y = +|x)$, the best decision can be made. However, the real probability distribution function is always unknown, so we can only to estimate the function $\widehat{P}(y = +|x)$. For simplicity, we replace $P(y = +|x)$, and $\widehat{P}(y = +|x)$ by $P_+(x)$, and $\widehat{P}_+(x)$, respectively. Our purpose is to study the risk of $\widehat{P}_+(x)$ defined by RDT to approximate $P_+(x)$. Mean square error (MSE) is the criteria to measure such risk, which is given by,

$$MSE(\widehat{P}_+(x)) = E\{[\widehat{P}_+(x) - P_+(x)]^2\} = \int [\widehat{P}_+(x) - P_+(x)]^2 dx$$

In the following sections, we will investigate how two parameters of RDT, that is, the number of trees and the minimal number of instances in the leaf nodes, can affect the MSE performance of RDT.

2.1. Univariate Probability Estimation

In the case of univariate probability estimation, RDT splits the data space (a line) into multiple bins. Each bin contains several data instances. Except that the bin-widths are not uniform, RDT is similar to the histogram, which is a nonparametric estimation method. In statistics, nonparametric estimation is more robust for different data distributions than parametric estimation methods which assume the data fit some parametric models. For RDT, a single tree can be treated as a random histogram. In the single dimensional case, we have

$$\widehat{P}_+(x) = \frac{v_{+j}}{v_j}, x \in B_j,$$

where B_j is a bin of the histogram, h_j is the width of B_j, and v_j is the number of instances in B_j. We then average the estimated probabilities by several trees to get the final estimation of the probabilities of being the positive class for a given data instance x by

$$\widehat{P}_+(x) = \sum_{i=1}^{m} \frac{v_{+j_i}}{v_{j_i}}.$$

Then, we can carry out the bias-variance analysis.

Theorem 1 shows that the MSE between the estimated and true probabilities depends on the width of bin h as well as the number of instances in the bin v. When h becomes narrow, the bias αh^4 will be reduced. But, the variance $\frac{\beta}{v}$ decreases with larger values of v. However, the decrease in h and increase of v can't happen at the same time, for that $h \propto v$, that is, the larger width of the bin, the more instances fall into the bin, so the value of either h or v can be adjusted to minimize MSE of the model. In practice, controlling h is more difficult for multi-dimensional problems, especially when there are lots of discrete and categorical features. Therefore, v is used as the control parameter of the model.

Theorem 1 *The* MSE *(mean square error) of a bin is as follows:*

$$\text{MSE} \approx \alpha h^4 + \frac{\beta}{v} \tag{1}$$

where α and β are two constants, αh^4 is the bias, and $\frac{\beta}{v}$ is the variance.

As follows is the proof of Theorem 1:
Proof $P_+(x)$ is the probability function. For simplicity, consider estimating $P_+(0)$. Let h be a small positive number. Define

$$p_+^h \equiv \mathbb{P}(-\frac{h}{2} < x < \frac{h}{2}) = \frac{1}{h}\int_{-h/2}^{h/2} P_+(x)dx \approx P_+(0),$$

so we have

$$P_+(0) \approx p_+^h.$$

Let v_+ be the number of observed instances belonging to class + in the range $(-h/2, h/2)$, and v be the number of instances in the range $(-h/2, h/2)$. An estimation to $P_+(0)$ is

$$\widehat{P}_+(0) = \widehat{p}_+^h = \frac{v_+}{v}. \tag{2}$$

Let

$$P_+(x) \approx P_+(0) + xP'_+(0) + \frac{x^2}{2}P''_+,$$

so,

$$p_+^h = \frac{1}{h}\int_{-h/2}^{h/2} P_+(x)dx \approx \frac{1}{h}\int_{-h/2}^{h/2}[P_+(0) + xP'_+(0) + \frac{x^2}{2}P''_+]dx = P_+(0) + \frac{P_+(0)''h^2}{24}.$$

As a result, by Equation 2, we can derive that

$$\mathbb{E}(\widehat{P}_+(0)) = \frac{\mathbb{E}(v_+)}{v} = \frac{\mathbb{E}(vp_+^h)}{v} = p_+^h \approx P_+(0) + \frac{P''_+h^2(0)}{24}.$$

Therefore, the bias is

$$\text{bias} = \mathbb{E}(\widehat{P}_+(0)) - P_+(0) \approx \frac{P''_+(0)h^2}{24}.$$

To calculate variance, based on the fact that $\mathbb{V}(\widehat{P}_+(0)) = vp_+^h(1-p_+^h)$ Wasserman (2009), we can derive

$$\mathbb{V}(\widehat{P}_+(0)) = \frac{\mathbb{V}(v_+)}{v^2} = \frac{p_+^h(1-p_+^h)}{v},$$

so the variance is given by

$$\mathbb{V}(\widehat{P}_+(0)) = \frac{1}{v}[P_+(0) + \frac{P''_+(0)h^2}{24}][1 - P_+(0) - \frac{P''_+(0)h^2}{24}] \approx \frac{1}{v}P_+(0)[1-P_+(0)].$$

Finally, we can have the MSE by

$$\text{MSE} = bias^2 + \mathbb{V}(\widehat{P}_+(0)) \approx \frac{P_+(0)''h^4}{576} + \frac{1}{v}P_+(0)[1-P_+(0)] \equiv \alpha h^4 + \frac{\beta}{v}$$

■

Following the study on the estimation risk within one bin, we then investigate why averaging the estimated probabilities from multiple trees would have less risks compared to a single tree. Given x and the number of trees m, by Equation 3, the $P_+(x)$ is

$$\widehat{P}_+(x) = \frac{1}{m}\sum_{i=1}^{m} p_(h_i)(x) \approx \frac{1}{m}[\sum_{i=1}^{m} P_+(t_i) + \sum_{i=1}^{m}\frac{P''_+(t_i)h_i^2}{24}],$$

where h_i indicates the width of the i-th bin, and t_i is the middle point of the i-th bin. When $m \to \infty$, the range of t_i is $[x - a/2, x + a/2]$. Let $h = \max(h_1, \dots, h_m)$, we have

$$\widehat{P}_+(x) \approx \frac{1}{a}[\int_{x-a/2}^{x+a/2} P_+(t)dt + \frac{h^2}{24}\int_{x-a/2}^{x+a/2} P''_+(t)dt].$$

According to Equation 3, we can further derive that

$$\widehat{P}_+(x) \approx P_+(x) + \frac{h^2P''_+(x)}{24} + \frac{h^2}{24a}\int_{x-a/2}^{x+a/2} P''_+(t)dt = P_+(x) + \frac{h^2P''_+(x)}{24} + \frac{h^2}{24a}[P'_+(x+a/2) - P'_+(x-a/2)]$$

This means $\widehat{P}_+(x)$ will get close to $P_+(x)$ as m increases. It explains as more trees are used, the estimation risk by RDT is decreased.

2.2. Multivariate Probability Estimation

Now, we are ready to generalize the analysis to the multivariate case. For simplicity, we treat the partitioned subspace as a hypercube in the feature space. Suppose the dimension of data is d, the data sample is thus $\mathbf{x} = \{x_1, \ldots, x_d\}$. Then the estimated probability is

$$\widehat{P}_+(\mathbf{x}) = \frac{v_{+j}}{v_j}, \mathbf{x} \in C_j, \tag{3}$$

where C_j is the hypercube, $C_j = B_{j1} \times B_{j2} \ldots B_{jd}$, and B_{ji} is a bin or range on the feature i. Like the bias-variance analysis for the univariate case, based on the Taylor expansion of $P_+(\mathbf{x})$ on the zero point $0 = (0, \ldots, 0)$:

$$P_+(\mathbf{x}) \approx P_+(0) + \sum_i^d x_i \frac{\partial P_+(0)}{\partial x_i} + \sum_{1 \le i,j \le d} \frac{x_i x_j}{2} \frac{\partial^2 P_+(0)}{\partial x_i x_j} + \mathrm{O}(0^3),$$

the MSE of multivariate probability of zero point $0 = (0_1, \ldots, 0_d)$ by a hypercube is

$$\mathrm{MSE} \approx \frac{P_+(0)'' \sum_{i=1}^d h_i^4}{576} + \frac{1}{v} P_+(0)[1 - P_+(0)]$$

where h_i is the width of the range of the i-th feature. Then we have the following theorem,

Theorem 2 *The* MSE *of a hypercube is given by :*

$$\mathrm{MSE} \approx \alpha \sum_{i=1}^d h_i^4 + \frac{\beta}{v}. \tag{4}$$

Equation 4 indicates that the narrower the range of features h_i becomes, the smaller the bias of the estimation will be. On the other hand, the larger the number of instances in the hypercube v gets, the smaller the variance will be. In addition, the bias of MSE increases with d linearly. If h_i is reduced, the bias part can be reduced, but the number of instances in the hypercube will also be less.

The contradiction between bias and variance of multivariate problem are more significant than that of univariate problem, which also explains why for the real-world data mining problem containing many features, a smaller v (such as 3 or 4) works better than larger v. The reason is that a larger v with respect to larger h_i can increase the bias significantly.

Similar to the analysis on the relation between the estimation risk and the number of trees m for univariate problems, one can also derive that

$$\widehat{P}_+(x) \approx P_+(x) + \frac{dh^2 P_+''(x)}{24} + \frac{dh^2}{24a^d}[P_+'(x + a/2) - P_+'(x - a/2)]$$

where $m \to \infty$, $a = (a, \ldots, a)$, $h = \max(h_1^1, \ldots, h_1^d, \ldots, h_m^d)$, and h_i^j is the width of the j-th dimension of the hypercube i. Thus,as m increases, $\widehat{P}_+(x)$ gets closer to $P_+(x)$.

2.3. Insights

The above bias-variance decomposition analysis reveals the deep mechanism of RDT. It explains that although RDT doesn't optimize any global (or local) objective functions or decision boundary explicitly, it can still achieve competitive and robust performance over different data sets. The analysis also confirms our assumption that the crucial part of RDT is not the tree structure but rather the random shattering mechanism. That indicates a more general principle to design an even better algorithm to overcome the weakness of RDT for big data problem, which is summarized in the Section 1.

3. Random Decision Hashing

The analysis of RDT suggests that the 'tree' is simply one kind of random partition method. Because of the complexity of maintaining tree structures, it becomes difficult to scale across multiple computing nodes. Even more worse, it makes RDT either build huge amount of empty tree structures or iterate through the large training data for several times. In order to address these two issues, we first present the basic idea of our algorithm, referred to as Random Decision Shattering Hashing (RDH) and then propose two variants adaptable to both dense and sparse data, respectively.

3.1. Basic Idea

On the high level, the basic idea is simply to replace the random tree of RDT with the unsupervised LSH functions to shatter the data space randomly. In the training stage, RDH uses unsupervised LSH functions to hash the training data instances so that similar items are mapped to the same buckets (hash codes) with high probability. Then, the algorithm counts the frequencies of classes in the buckets. For scoring, RDH uses LSH to hash the given data instance to a bucket and scores the instance with the frequency of the class. To achieve better accuracy, we further apply multiple LSH functions and averages the scores returned by each of them.

Similarly to RDT which requires dozens of random trees to achieve certain accuracy, we may use several hash functions in RDH. As a result, the multiple hashing operations will slow down the training process. To overcome this challenging, we instead first apply a single hash function to map the input data instance to a binary code. Then, in order to get multiple different hash codes, we choose to use several randomly generated bit masks to do logical **And** operation on the single hash code to get a group of different new hash codes. Apparently, the number of the **1**s in the bit masks is the key parameter to control the number of instances in the shattered subspaces. According to the aforementioned analysis, this number in turn controls the estimation risk. The overall algorithm of RDH is presented in Algorithm 1.

Different LSH methods may get different accuracy and efficiency over different datasets. Designing appropriate LSH functions is crucial to achieve good performance. An ideal LSH for RDH should be unsupervised, shattering evenly, simple and adaptable over different data distributions. Although there is a bunch of LSH functions, we take the two classic LSH methods, Hyperplane Hashing(or random projection) Charikar (2002) and its variant SimHash Sadowski and Levin, as the prototypes to handle the sparse and dense data,

```
Input: training set $X = \{(\mathbf{x}, y)\}$, $y$ is the class, hash function $h(x)$, the number of masks
       $m$
Output: $model$ is a counter, recording the positive instance frequencies with the same hash
        code of each mask
generate $mask_j, j = 1, ..., m$  for $i = 1; i \leq |X|; ++i$ do
    $code = h(\mathbf{x}_i)$  for $j = 1; j \leq m; ++j$ do
        $b = mask_j$ AND $code$  update $model$ with the$j$, $b$ and $y_i$
    end
    return $model$
end
```

Algorithm 1: RDH Algorithm

respectively. The main reason to choose the two hash methods is that the two LSH methods is simple enough and make the algorithm be easy to parallelize with high efficiency. In the following two sections, we present the specifically designed two methods to handle the different characteristics of the sparse and dense datasets, respectively.

3.2. Hash for the Dense Data

Hyperplane Hashing is a good prototype for RDH to handle the dense datasets. The method chooses a random hyperplane to hash input vectors. Given an input vector $\mathbf{x}$ and a hyperplane defined by $\mathbf{r}$, we let $h(\mathbf{x}) = sgn(\mathbf{x} \cdot \mathbf{r})$. That is, depending on which side of the hyperplane $sgn(\mathbf{x})$ lies, we have $h(\mathbf{x}) = \pm 1$. In order to generate p-bit hash code, we can randomly generate p hyper planes.

Although the hyperplane hashing is simple, the computational cost is still high. If the feature number is d, to generate a p-bit hash code for a data instance, there are pd multiplication operations, and $p(d-1)$ addition operations for a single one hash function. Even worse, we may need multiple hash codes for a data instance. If the number of masks is m, the number of total multiplication and addition operations will become pmd and $pm(d-1)$. In order to reduce the computational cost, we propose the following heuristics :

- Limit the coefficients of each hyper plane only be 0, 1 and -1
- Limit the number of nonzero coefficients of a hyper plane to be $\lceil \frac{d}{mp} \rceil$

The first heuristic reduce the possible space of hyper plane, but for the problems with more features, the limited possible space is still huge. The second heuristic is to make the coefficients of hyper planes as spare as possible, and at the same time, all the features can be covered. Consequently, the coefficients are only 0, 1 and -1, we can remove all the multiplication operations. Moreover, since there are only d nonzero coefficients of all the hyper planes, the addition operations also can be reduced to d. Then the training computing complexity is about $O(nd)$, where n is the number of the training data instances in total.

3.3. Hash for the Sparse Data

The hashing method for the dense datasets can be directly reused for sparse datasets. First, the hyper plane coefficients for the dense data is sparse. When both the hyper planes and the data instances become sparse, the inner product between the two tends to be 0 quite often, making most hash codes the same, so the data cannot be shattered evenly, and thus reduce the accuracy. Secondly, for the sparse datasets with large number of features, the dimension of the hyper plane is also large, which incurs too much memory consumption. Therefore, we turn to the SimHash Sadowski and Levin, which is good at working on the sparse datasets. Although the mathematical principle of SimHash is similar to hyperplane hashing, SimHash doesn't need to generate hyperplanes. Instead, the identities of features and the feature hash function determine these hyperplanes implicitly. The description of SimHash is listed in Algorithm 2.

```
Input: sparse instance $\mathbf{x} = (f_i, v_i)...$, $f_i$ is the feature index, and $v_i$ is the value
Output: hash code $S$
initialize vector $q$-dimension $r$ with 0   initialize a $q$-bit bitmap $S$ with 0   for $(f, v) \in \mathbf{x}$ do
    $b =$ FastHash($f$)   for $i = 1; i \leq q; ++i$ do
        if $b[i] == 1$ then
            $r[i] += v$
        else
            $r[i] -= v$
        end
    end
end
for $i = 1; i \leq q; ++i$ do
    if $r[i] > 0$ then
        $S[i] = 1$
    else
        $S[i] = 0$
    end
end
return $S$     Function FastHash(64 bit integer: $f$)
    $f \wedge = f \gg 23$   $f* = 2127599bf4325c37$(Hex)   return $f \wedge = f \gg 47$
```
Algorithm 2: SimHash with FastHash

The hash function used in the original SimHash is to hash string values. But for most classification datasets, the features are labelled with integer numbers. Naturally, we need a hash function to map integer numbers to bitmaps. Here, we apply the fast hash FastHash technique which consists of two **Xor**-shifts and one multiplication. The pseudo code of the fast hash is described in Algorithm 2 as a module procedure.

The original hash code produced from fast hash is encoded by 64-bit. However, the 64-bit hash function is not enough to generate dozens of hash codes by bit masks with much diversity. Assuming, there are 50 bit masks with 10 bits over the 64 bit hash code, each bit will be hit by about 8 bit masks. To ensure the diversities of the bit masks, we should get longer bit hash code from the hash function. To get 128-bit hash code, we merge the hash codes $h(i)$, and $h(-i)$, where h is the mix fast hash, i is the feature index; to get 192-bit hash code, merge $h(i)$, $h(-i)$ and $h(i+d)$; to get 256-bit hash code, merge $h(i)$, $h(-i)$, $h(i+d)$ and $h(-i-d)$; and so on. Obliviously, the more bits we use, the more computing cost we have to pay, but the accuracy will be improved. In the current work, the max length of hash code is 512, which merges $h(i)$, $h(-i)$, $h(i+d)$, $h(-i-d)$, $h(i+2d)$, $h(-i-2d)$, $h(i+3d)$ and $h(-i-3d)$. As the result, the length of hash code is limited in 64, 128, 192, 256, 320, 384, 448, 512. To determine the correct length of the hash code, we choose the minimum number which is greater or equal to mp. However, when $mp > 512$, the length of hash code is still 512. While $mp <= 512$, we generate m bit masks without concurrent bits. Otherwise, there will be some unavoidable concurrent bits across the masks.

```
Input: Data partitions $X_i$,$i = 1, ..., k$ , standalone RDH $RDH(X_i)$
Output: $model$
Function Map($X_i$)
  | return $model_i = RDH(X_i)$
Function Reduce($model_i$)
  | return $model = \sum model_i$
```

Algorithm 3: MapReduce RDH Algorithm

3.4. MapReduce Implementation

Our goal is to handle large-scale datasets. Nowadays, clusters of massive computational nodes become the de facto standard as the big data processing infrastructure. One of the most popular programming paradigms for the cluster settings is MapReduce. In fact, the logic of the MapReduce version RDH is simple: (i)In the map stage, each mapper executes the standalone RDH algorithm on the designated data partition, which records the numbers of classes of examples over appeared buckets(hash codes of bit masks) for each data partition; (ii) In the reduce stage, the sole reducer gather all the number records of the buckets and sum up the numbers of classes every buckets, and calculate the class frequencies of the buckets. We formally describe the algorithm in Algorithm 3.

4. Experiments

In this section, we evaluate the performance of RDH on a group of diverse benchmarks. Due to the limited scalability of RDT, we first demonstrate the advantage of RDH over RDT on eight relatively small datasets running on a standalone machine. Then, we compare our method with the mainstream Logistic Regression (LR), Support Vector Machine (SVM) and Decision Tree (DT) on the Spark 1.2 Spark platform over six large-scale datasets. In all cases, we show that RDH significantly outperforms the other competitors in terms of prediction accuracy, memory efficiency.

4.1. Experimental Settings

For the small datasets, the experiments are carried out over a single desktop with Intel I5 3.0GHz CPU, 4GB memory, and JDK 1.6. For the large-scale datasets, we use a Spark 1.2 cluster. Spark is a fast and general engine for large-scale data processing, which is the implementation of Resilient Distributed Datasets. Spark is more suitable for processing iterative machine learning algorithms than Hadoop Hadoop. The official announcement of Spark claims that it runs programs up to 100 times faster than Hadoop MapReduce in memory, or 10times faster on disk.

Our experimental cluster consists of eight computing nodes, each of which has 64GB memory and Intel(R) Xeon(R) E5-2620 @ 2.00GHz CPU with 6 cores. However, the resource management configuration restricts the maximum number of workers with 1 executor for a job to be 12, and the maximum memory for the master node and the worker node to be 6GB and 8GB, respectively. Although Spark can persist the datasets in the memory to accelerate the learning process, limited by the capability of our experimental cluster, not all the these large datasets can be loaded into the memory. In practice, we also can't assume that the computing cluster can hold the datasets in the memory always, because the datasets may much larger than these experimental datasets. As the results, we don't persist the datasets in the memory to examining the efficiency of RDH and the competitors for the big data problems which is too big to be loaded into memory. Besides, all our competitors LR, SVM and DT are the default optimized implementations in the MLLib library provided by Spark 1.2.

For the small dataset experiment, we implement RDH by Java according to Algorithm 1 and 2. For the large-scale datasets, we implement RDH by Scala for Spark 1.2 according to Section 3.4. Both versions support the two variants of RDH to handle the dense and sparse datasets, respectively. Because of the simplicity of RDH, our algorithm only includes two parameters to tune, namely, the number of masks m and the number of bits of the hash code p.

In all experiments, we use the Area Under ROC Curve (AUC), for that other measurements, such as the error rate, recall and F score, may vary by different classification thresholds. AUC score is relatively more robust and comprehensive to test the accuracy of classification algorithms since it is independent of such magic thresholds.

4.2. Competitors

RDT. Because RDT is hard to scale over a parallel computing platform, we instead compare RDH with RDT on the standalone machine to validate the accuracy and efficiency. In the experiments, we apply the Java implementation of RDT in the DICE Dice library. One parameter to set is the number of trees. Because the number of trees corresponds to the respective number of the masks of RDH, we also use m to present this parameter. Another parameter of RDT is the minimal number of instances in a leaf node q, which is also tuned by cross-validation.

Logistic Regression and SVM MLLib of Spark 1.2 includes the implementations of LR and linear SVM algorithm. The training process of LR uses the stochastic gradient descent (SGD) Bottou (2010) method, and takes as input a regularization parameter (λ) along with the parameters associated with the stochastic gradient descent (s, I). s is the learning rate, and I is the number of maximum iterations. Because there are several datasets having much more features than the data instances, we choose the L2 regularized variant of LR to control the over-fitting risk. Here, we do not use the L1 regularized variant simply for that it will lead to a very sparse LR model. If the test dataset is also sparse, for most cases, the predictions will be the same, since the dot product between two sparse vectors (feature vector and coefficient vector) will be 0 with high probability.

Decision Tree MLLib of Spark 1.2 also supports the Decision Tree algorithm. There are two node-impurity measurements Gini and Entropy for the DT algorithm. We choose the Entropy in the test. Another parameter is the maximum-depth. According to the source code of the algorithm, to increase the depth of a tree by one, it needs at least one iteration over the training data. In our experiments, the DT algorithm can only run on the two datasets with less features. For the datasets with large number of features, DT is caused to fail by the "out of memory" exception. Even if the memory is not limited, DT cannot train in reasonable time in these datasets for that its time complexity is $O(d^2n)$, where $d = \sum(f_i - 1)$ and f_i is the size of values of i-th feature. When the number of features is more than one million, the complexity becomes dramatically large.

Naive Bayes is also in the MLLib, which is a non-iterative algorithm as well. However, the implementation of Naive Bayes only supports the document classification, which requires that the features have to be the frequencies of the terms. The features of all the benchmark datasets simply do not meet this restricted requirement. We test the algorithm over these datasets, but the algorithm just achieves about 0.5 AUC on two datasets, and fails on all the other datasets because of the "out of memory" exception. As a result, we do not take the Naive Bayes as an effective competitor.

4.3. Results on Small Datasets

We first compare RDH with RDT in classification accuracy and training speed on eight small datasets from Libsvm LIBSVM. The statistics of these datasets are summarized in Table 1.

Data	Statistic				RDH ($m = 30$)			RDT($m = 30, q = 4$)	
	Type	#Feature	#Train	#Test	q	AUC	Tr. Time	AUC	Tr. Time
a9a	dense	123	32561	16281	12	**0.894**	**4.125**	0.890	24.171
mushrooms	dense	112	7124	1000	10	1.000	**0.606**	1.000	3.608
w8a	dense	300	49749	14951	12	0.982	**13.739**	**0.997**	33.759
splice	dense	60	1000	2175	6	**0.966**	0.416	0.909	**0.387**
cod-rna	dense	8	59535	271617	12	**0.971**	**6.473**	0.969	7.799
covtype	dense	54	481082	100000	16	**0.770**	**42.027**	0.768	240.392
gisette	dense	5000	6000	1000	6	0.922	13.281	**0.934**	**11.488**
rcv1	sparse	47236	20242	677399	12	0.939	**9.362**	**0.981**	5438.532

Table 1: Statistics and Parameter Settings of the Small Datasets

On these small datasets, both the number of the bit masks (RDH) and that of the trees (RDT) are fixed to be 30. To ensure an appropriate number of instances in each bucket, we change parameter p of RDH from 2 to 20 by a unit of 2. For the RDT, we tune q from 1 to 10, and find that "4" is the best one for all the datasets. The AUC score and the training time of the two algorithms over the eight datasets are listed in Table 1.

We can see that RDH is more accurate than RDT on the following 4 datasets, *a9a*, *splice*, *splice*, *cod-rna*, and *covtype*. On the other four datasets, the AUC of RDH is also close to RDT. Overall, RDH is as good as RDT in terms of prediction accuracy. Furthermore, with respect to the training time, RDH runs significantly faster than RDT on six datasets, *a9a*, *mushrooms*, *w8a*, *cod-rna*, *covtype*, and *rcv1*. On the other two datasets, the runtime of RDT varies greatly. In particular, on the *rcv1* dataset, RDT costs more than 1.5 hours. The reason for the inefficiency of RDT is that the distribution of the feature values is very skewed. Therefore, RDT suffers from building many deep trees to partition the instances

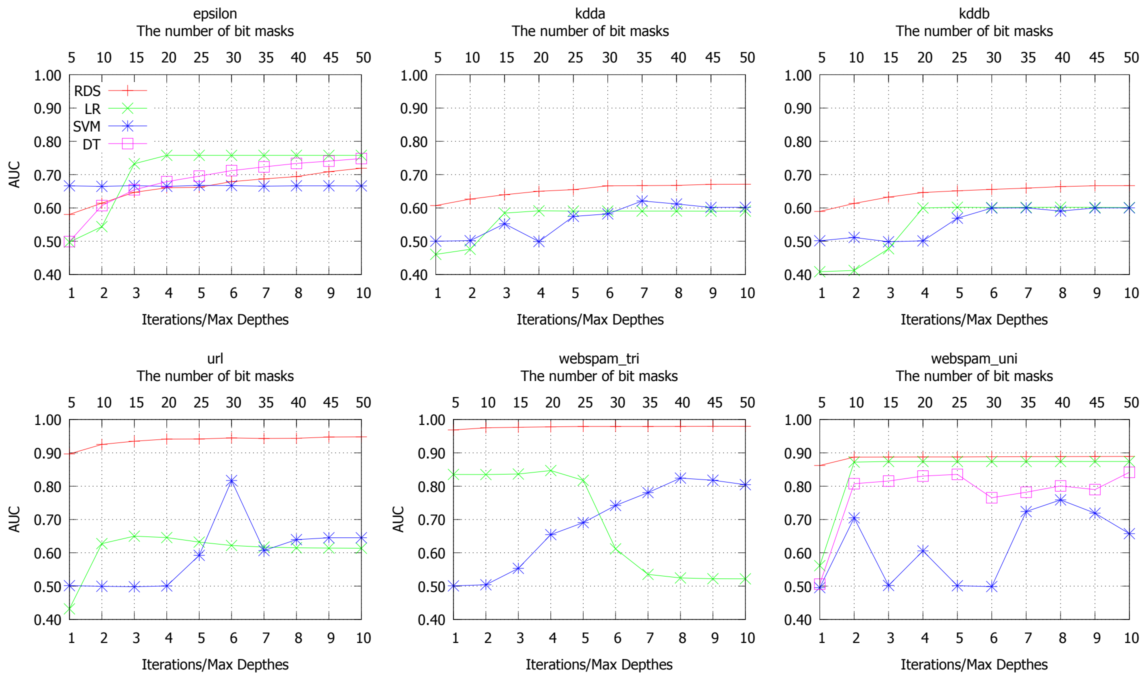

Figure 1: The comparison of AUC Scores among RDH, LR, SVM and DT.

evenly. In contrast, RDH does not suffer from this issue. In addition, these experimental results also confirm that the tree building process of RDT is just one way to partition the data space, and it can be replaced by other more efficient alternatives.

Data	Type	#Feature	#Train	#Test	Tr. Vol.	RDH		LR			SVM			DT
						Mem.	p	Mem.	s	λ	Mem.	s	λ	Mem.
epsilon	dense	2000	400000	100000	11.3G	2G/1G	10	2G/1G	10	0.8	2G/1G	1	0.2	2G/2G
kdda	sparse	20216830	8407752	510302	2.49G	2G/1G	12	6G/8G	8	1.1	6G/8G	1	1	-
kddb	sparse	29890095	19264097	748401	4.78G	2G/1G	12	6G/8G	8	1.2	6G/8G	1	1	-
url	sparse	3231961	2000000	396130	344M	2G/1G	16	4G/4G	1	0.9	4G/4G	1	0.3	-
ws_tri	sparse	16609143	300000	50000	20G	2G/1G	12	6G/8G	5	1.5	6G/8G	1	1	-
ws_uni	dense	254	300000	50000	327M	2G/1G	12	2G/1G	9	1.8	2G/1G	10	0.8	2G/2G

Table 2: Statistics and Parameter Settings of the Big Datasets

4.4. Results on Large Datasets

The six big datasets are also selected from the Libsvm binary classification datasets LIBSVM, including *epsilon*, *kdda*, *kddb*, *url*(combine), *webspam_tri* Wang et al. (2012), and *webspam_uni* Wang et al. (2012). The only criteria of selecting these datasets is that they have to be "big". The volume of the datasets varies from several hundreds MB to about 20 GB. The number of features is in the range from 254 to almost 30 millions. The minimum number of training instances is 400,000, and the maximum number is up to 20 millions. Overall, two datasets are dense, and the other four is sparse. Since the *url*, *webspam_uni*

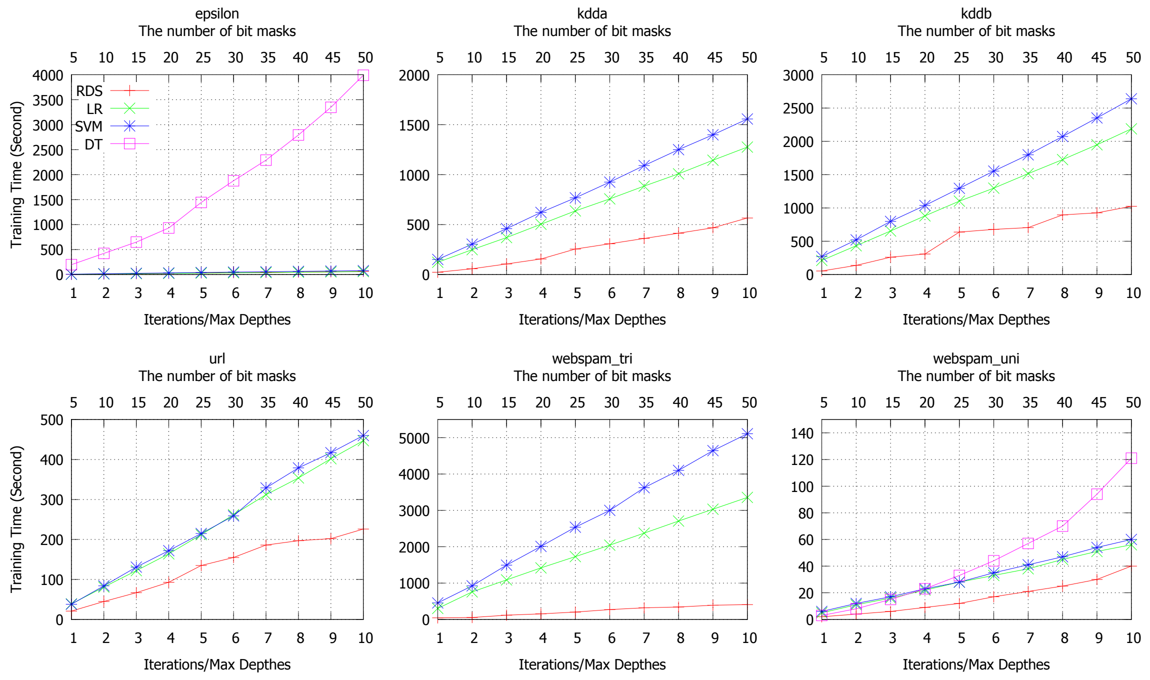

Figure 2: The comparison of training time among RDH, LR, SVM and DT.

and *webspam_tri* do not include the test datasets, we take the first 396,130 data instances as the test dataset for *url*, and the first 50,000 data instances as the test for both *webspam_tri* and *webspam_uni*. The statistics of these datasets are summarized in Table 2.

After demonstrating the superiority of the RDH over RDT, we then evaluate the performance of RDH on the big datasets by comparing it to LR, SVM and DT. We claim RDH is more suitable for the distributed computing setting in previous sections. In order to verify this claim, we conduct the experiments on the Spark 1.2 cluster. We first compare their prediction performance on the held-out testing data. Then, we show the respective training speed of these algorithms in the cases when the training data can be loaded in the cluster memory and when they have to be kept on disks. In the end, we examine the scalability of RDH with respect to different numbers of running nodes in the cluster.

4.4.1. Parameter and Memory Settings

For RDH, we run it with the number of bit masks from 5 to 50 every 5-mask. For LR, SVM and DT, we run with the iterations or maximum depth from 1 to 10. For RDH, we tune the p parameter from 2 to 20 every 2. For LR and SVM, we first fix the regularization parameter λ as 0, and tune the s parameter from 1 to 10 every 1. After the s has been determined, we tune the λ from 0 to 2 every 0.1. These tuned parameter settings on the 6 datasets are shown in Table 2, which also contains the memory settings of the algorithms on Spark. Note, in the field "Mem.", "2G/1G" means that the up bounds of memory of the master and worker are 2G and 1G respectively.

For all these 6 datasets with different numbers of features, instances and varied sparsity, RDH simply needs 2GB and 1GB memory for the master and each worker, respectively. In contrast, LR and SVM have to cost much more memory on the *kdda*, *kddb*, *url* and *webspam_tri* sparse datasets with millions features. The memory cost of DT exceeds the upper bound(6G/8G) of the system, and thus it fails on these 4 datasets. Actually, the memory cost of the trained model of RDH on the sparse datasets only depends on the parameter m and p, which determines the number of buckets. However, the model of LR and SVM heavily depends on the number of features. By comparison, the memory consumption of RDH is much more less than the others across all different datasets.

4.4.2. Accuracy of Prediction

The AUC scores of the algorithms with different numbers of bit masks, iterations and maximum depth over 6 datasets are plotted in Figure 1. Except on the *epsilon* datasets, the accuracy performance of RDH beats all the other 3 algorithms. The most significant improvement that RDH achieves compared to the other best algorithm is 17% on the *url* dataset. On the test data of the *url* dataset, the AUC of RDH exceeds 0.94 when the number of bit masks is greater than or equal to 20, and the best AUC of SVM (the best other algorithm) achieves 0.8169 when the number of iterations is 6. Even on the *epsilon* that RDH doesn't achieve the best accuracy, the AUC of RDH is still close to the other algorithms. Overall, we can see that even if we trade the accuracy of the other algorithms for speed by restricting the number of bit masks to be less than 10, RDH still has the best performance. Moreover, the AUC of RDH increases stably on all the datasets. When m becomes larger than 20, the AUC is improved very slowly, which is the same as RDT. In contrast, the AUC score of LR and SVM varies greatly with respect to the number of iterations.

It is also important to note that LR, SVM and DT algorithms may iterate datasets much more times than 10 in usual case. The limitation of the iterations in the experiments seems unfair to the competitors, which may achieve better accuracy with more iterations. However, for the big data problems, we should balance the accuracy and speed. If we don't limit the number of iterations, the training process of these algorithms may not finish in reasonable time compared to RDH that scans the datasets only one pass.

4.4.3. Time Cost of Training

We plot the training time of RDH, LR, SVM and DT on the 6 datasets with increasing number of bit masks, iterations, and maximum depth, respectively in Figure 2. Obviously, RDH costs significantly less time than the other algorithms over the 6 datasets. For the best case (*webspam_tri*), the second fastest algorithm (LR) spends 1418 seconds to achieve converge by 4 iterations, and RDH spends 153 seconds by 20 bit masks. LR costs 9.27 times of training time than RDH. The converge means that the improvement of AUC is within 0.1% for one more bit masks, iteration, or depth. Overall the more the features and instances of the datasets, the more efficient RDH compared to competitors. For example, *kdda*, *kddb*, *url*, and *webspam_tri* which are with millions of features and instances(except *url*), is an order of magnitude faster than other competitors roughly. We also show that the training time of RDH increases linearly with m. On the 4 sparse datasets, this linear trend of RDH has some fluctuations, since the number of bits of the hash codes increases every 64-bit. The time cost of DT is not only the worst, but also raises super-linearly with the maximum tree depth. In sum, implementation of RDH on Spark is more efficient than other main stream competitors for the large scale problems.

5. Conclusion and Future Work

We propose a new randomized algorithm RDH for large-scale classification problem inspired from the formal analysis of Random Decision Trees. Essentially being an iteration-free algorithm, RDH is highly suitable for distributed computing setting. To achieve the best speedup, we design different variants for dealing with the dense and sparse datasets, respectively. On real datasets of tens of millions of items and features, we show that RDH can achieve significantly better prediction accuracy, train speed and memory consumptions compared to other mainstream competitors.

Limited by the space and time, there are still many interesting works of RDH cannot be included in this paper. In the future, Like the RDT, RDH also can be applied to regression Fan et al. (2006b), and multi-label classification Zhang et al. (2008) problems.

References

A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1): 117–122, 2008.

B. Bai, J. Weston, D. Grangier, R. Collobert, K. Sadamasa, Y. Qi, O. Chapelle, and K. Weinberger. Supervised semantic indexing. *CIKM*, pages 187–196, 2009.

A. Bergamo, L. Torresani, and A. Fitzgibbon. Picodes:learning a compact code for novel-category recognition. *NIPS*, 2011.

L. Bottou. Large-scale machine learning with stochastic gradient descent. *International Conference on Computational Statistics*, 2010.

A. Z. Broder. On the resemblance and containment of documents. *Proceedings of Compression and Complexity of Sequences*, 1997.

A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *STOC*, 1998.

M. Charikar. Similarity estimation techniques from rounding algorithms. *STOC*, 2002.

Dice. http://www.dice4dm.com.

The Economist. Data, data everywhere. 25th, February 2010.

W. Fan. On the optimality of probability estimation by random decision trees. *AAAI*, 2004.

W. Fan, H. Wang, P. S. Yu., and S. Ma. Is random model better? on its accuracy and efficiency. *ICDM*, 2003.

W. Fan, E. Greengrass, J. McCloskey, and P. S. Yu. Effective estimation of posterior probabilities: Explaining the accuracy of randomized decision tree approaches. *ICDM*, 2005a.

W. Fan, J. Mathuria, and C. Lu. Making data mining models useful to model non-paying customers of exchange carriers. *SDM*, 2005b.

W. Fan, J. McClosky, and P. S. Yu. Making data mining models useful to model non-paying customers of exchange carriers. *KDD*, 2006a.

W. Fan, J. McClosky, and P. S. Yu. A general framework for accuracy and fast regression by data summarization in random decision trees. *KDD*, 2006b.

FastHash. https://code.google.com/p/fast-hash/.

A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. *VLDB*, 1999.

Hadoop. http://hadoop.apache.org/.

M. Immorlica, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. *STOC*, 2004.

P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. *STOC*, 1998.

P. Jain, S. Vijayanarasimhan, and K. Grauman. Hashing hyperplane queries to near points with applications to large-scale active learning. *NIPS*, 2010.

B. Kulis and T. Darrell. Learning to hash with binary reconstructive embeddings. *NIPS*, 2009.

LIBSVM. http://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/binary.html.

W. Liu, J. Wang, R. Ji, Y. Jiang, and S. Chang. Supervised hashing with kernels. *CVPR*, 2012.

M. Norouzi and D. J. Fleet. Minimal loss hashing for compact binary codes. *ICML*, pages 353–360, 2011.

M. Norouzi, D. J. Fleet, and R. Salakhutdinov. Hamming distance metric learning. *NIPS*, pages 1070–1078, 2012.

C. Sadowski and G. Levin. Simihash: Hash-based similarity detection. *Technical Report UCSC-SOE-11-07*.

R. Salakhutdinov and G. Hinton. Semantic hashing. *Approximate Reasoning*, 2009.

G. Shakhnarovich, P. A. Viola, and T. Darrell. Fast pose estimation with parameter-sensitive hashing. *ICCV*, 2003.

Spark. http://spark.apache.org.

D. Wang, D. Irani, and C. Pu. Evolutionary study of web spam: Webb spam corpus 2011 versus webb spam corpus 2006. *CollaborateCom*, 2012.

L. Wasserman. All of nonparametric statistics. *Springer-Verlag New York*, 2009.

Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. *NIPS*, 2008.

K. Zhang and W. Fan. Forecasting skewed biased stochastic ozone days: analyses, solutions and beyond. *Knowl. Inf. Syst. 14(3)*, pages 507–527, 2008.

X. Zhang, Q. Yuan, S. Zhao, W. Fan, W. Zheng, and Z. Wang. Multi-label classification without the multi-label cost. *Knowl. Inf. Syst. 14(3)*, pages 507–527, 2008.