$O(m \log m)$ instance selection algorithms—RR-DROPs

1st Marek Orliński
Department of Informatics
Nicolaus Copernicus University
Toruń, Poland
morlinski@is.umk.pl

2nd Norbert Jankowski
Department of Informatics
Nicolaus Copernicus University
Toruń, Poland
norbert@is.umk.pl

***Abstract*—This paper is focused on an instance selection algorithm for classification purposes. We propose a new fast version of DROP algorithms with complexity reduced to $O(m \log m)$, while the original complexity was $O(m^3)$. The new RR-DROP algorithms use random region hashing forests and jungle, and several other data structures to keep the computational complexity as low as possible. The proposed algorithms can be used for huge datasets, with classification remaining unchanged, as proven by a statistical analysis on several datasets.**

***Index Terms*—instance selection, prototype-based classifiers, instance-based learning, k nearest neighbors**

I. Introduction

In this article we tackle instance selection applied to supervised classification problems. Instance selection means that we are looking for a subset $S \subseteq D$ of learning data ($D = \{(\mathbf{x}_i, y_i) : i = 1, \ldots, m, \mathbf{x}_i \in R^n, y_i \in R\}$). The subset S should be enough to build a trustworthy classifier upon, for example, a k nearest neighbor (kNN) classifier [1]. The number of different instance selection algorithms is quite large—we would like to recommend articles and books devoted to prototype selection algorithms [2]–[5]. However, most of these algorithms are fit only for small or medium-sized datasets because of their high complexity [2], [3] (greater or equal to the square of the number of instances).

In general, the tactic of several of the algorithms is to remove inconsistent instances (filtering methods) or instances that are somewhat redundant (prototype selection). An instance being redundant typically means that eliminating it from the training set does not decrease classification accuracy. Existing reviews show that while all reviewed algorithms serve the same purpose, they differ strongly in their details and consequentially have very different speeds and accuracies. In [2] it was clearly stated that most accurate algorithms are mostly slow.

For kNN and some other learning machines, all training examples must be stored as part of the model. From that perspective, a significant reduction of the dataset is very attractive. Instance selection algorithms can be also used to construct classification rules basing on the selected prototypes [6]. Such rules are then easy to understand, not being based on intervals (as in decision trees), but on distances from selected prototypes. In such a case it is convenient for instance selection to have as high a reduction rate as possible (e.g. as the algorithm Explore from [7]). However, high reduction is not obligatory. Another good application of instance selection is to use prototypes in the construction of neural networks (RBFN [8] or ELM [9], [10]) [11]. In such a case, the selected prototypes are successfully used as positions of the gaussian neurons. The advantage of such a solution is the automatic selection of the size of the network, since it is determined by the number of prototypes.

The goal of this paper is to construct algorithms of complexity $O(m \log m)$ on the basis of DROP algorithms proposed in [12]. In the next section we describe the original DROP algorithms. In the following, we present a special variant of hashing trees which is a basis for proposed fast version of the DROP algorithms. Construction of the new algorithms also requires utilization several supporting data structures. Moreover hashing has to be used for a few of subgoals. The last section compares learning time and complexity of the original and new DROP algorithms. We also analyze the average accuracies of the new and original algorithms on a wide range of small and medium-sized datasets. Finnaly we move our focus to instance selection for large datasets. The new algorithms are compared with other algorithms LSH-IS of complexity $O(m \log m)$ [13] and PSC [14] in terms of accuracy, reduction and speed.

II. DROP algorithms

Not all instance selection algorithms can be easily sped up. Upon an analysis of DROP algorithms proposed in [12], we have found a way to speed them up by applying locality sensitive hashing methods and additional data structures, which leads to more sophisticated procedures, compared to the original DROPs. The main concept of DROP2 is to delete all vectors whose removal does not change the classification results on the remainder of the set $\mathcal{D}$.

This idea produces the definition of the set $\mathcal{A}$, which simplifies the testing of the changes in classification to the elements of $\mathcal{A}$, in contrast to testing on the whole of $\mathcal{D}$:

$$\mathcal{A}(\mathbf{x}, k) = \{\mathbf{x}' : \mathbf{x} \in N^k(\mathbf{x}')\}, \tag{1}$$

where $N^k(\mathbf{x}')$ is the set of the k nearest neighbors of $\mathbf{x}'$.

The 'OrderByNearestEnemy' below defines an ascending order of instances (in $\mathcal{D}$) with respect to the distance to their nearest enemy (the nearest instance from an opposite class). The scheme of the algorithm is presented in Alg. 1.

Algorithm 1: Drop2(D)

```
   Data: $D = [instance_1, \dots, instance_m]$
   Result: vectors selected from $D$
1  begin
2      order = Reverse(OrderByNearestEnemy($D$))
3      do
4          changes = false
5          for $i \in order$ do
6              if not $D_i.pruned$ and Improvement(i) $\geq 0$
                then
7                  Prune(i)
8                  change = true
9      while changes;
10     return all $D_i$, where not $D_i.pruned$
```

The 'Improvement(i)' means the difference in the numbers of valid classifications without and with i-th instance. If the i-th instance is *redundant* it becomes pruned by 'Prune(i)'. To check the improvement the algorithm only has to visit the elements of $\mathcal{A}$, but if pruning is necessary, each $\mathcal{A}(\mathbf{x}_j, k)$ where $j \in \mathcal{A}(\mathbf{x}_i, k)$ must be updated and this forces several searches through all elements of $\mathcal{D}$. Those elements will be optimized in the next section, but the current cost is $O(m^2)$.

Authors of DROP2 proposed its modification, the DROP3, which starts the procedure with a *cleaning*, see Alg. 2. Cleaning is an elimination of inconsistent instances. An *inconsistent* instance is one whose neighbors are mostly from a different class. The test of inconsistency is performed for each instance.

Algorithm 2: Drop3(D)

```
   Data: $D = [instance_1, \dots, instance_m]$
   Result: vectors selected from $D$
1  begin
2      order = Shuffle($[1, \dots, m]$)
3      for $i \in order$ do
4          if $D_i.label \neq Classify(i)$ then
5              Prune(i)
6      Drop2($D$)
7      return $D_i$, where not $D_i.pruned$
```

Wilson et. al. also proposed the DROP4 which differs from the DROP3 in the strength of *cleaning*. In DROP4 an *inconsistent* instance is one whose neighbors are mostly from a different class, but additionally, the deletion of this instance would not decrease classification accuracy. Please see the Alg. 3 for details.

Algorithm 3: Drop4(D)

```
   Data: $D = [instance_1, \dots, instance_m]$
   Result: vectors selected from $D$
1  begin
2      order = Shuffle($[1, \dots, m]$)
3      for $i \in order$ do
4          if $D_i.label \neq Classify(i)$ then
5              if not $D_i.pruned$ and Improvement(i) $\geq 0$
                then
6                  Prune(i)
7      Drop2($D$)
8      return $D_i$, where not $D_i.pruned$
```

III. Fast DROP algorithms

To construct a faster version of the DROP algorithms, it is necessary to use much more sophisticated data structures. One of the main data structures is needed to facilitate fast computation of nearest neighbors. In the past, several concepts were introduced for calculating nearest neighbors like the r-trees [15], the vantage-point trees [16] or kd-trees [17], but those methods have (complexity) problems in multidimensional spaces as it was discussed in [18]. Additionally, fast search for nearest neighbors is not enough to construct fast versions of the DROP algorithms. This is why we decided to base on locality sensitive hashing (LSH) initially proposed in [19] and the forests of LSH for a more accurate approximation. LSH allows for computation of an approximation of nearest neighbors in an estimated time of $O(\log m)$ in contrary to $O(m)$. The forests or trees will be aided by hash sets or binary heaps to cope with multiple traversals through the trees.

In fact we used a modified version of LSH—the random regions tree (RRT). Moreover we construct two types of trees. One tree is a static tree which is not modified after construction. Another one is ready to have instances removed from, whilst maintaining all functionality. This is necessary to prune instances from $\mathcal{D}$ as in the DROP algorithms.

The first version of random regions tree was presented in [20] and now we made additional modifications, mostly to facilitate work with huge datasets, especially with multiple duplicates or with unordered attributes. Another reason for modifications was to make additional savings on execution time compared to our previous implementations—in the new version hashes can be shared by some RRTs which enables further optimizations.

LSH was proposed in [19]. The main idea of LSH is to independently and uniformly draw random hyperplanes which divide the space $\mathcal{R}^n$ into regions (bins) of similar objects. The bins contain a set of points and they serve as a source of potential neighbors of any point in the given bin. Sometimes a point may be situated near a border of its bin—in such cases not all of its actual nearest neighbors can be found in the given bin.

A. Random regions tree

In our algorithm, we construct the LSH tree in a slightly different way compared to [19]. First, we construct hashed data by addition of a bit of noise to data and multiplication with random hashes:

$$H = (D + \epsilon) \times R \qquad (2)$$

where ϵ is a bit of noise ($\epsilon_{ij} = rand([-10^{-4}, 10^{-4}]) * \sigma_j$), σ_j is the standard deviation of j-th attribute. R is $n \times h$ matrix of uniform random values in $[-1, 1)$ and $h = \lceil (1.2 \log(m) \rceil$ defines the maximal depth of the tree. The addition of noise yields the elimination of hash multi-duplication (in case of duplicated data instances).

Next, we start divisions into two subtrees with appropriate subsets of vectors. The divisions at the same depth use the same hashes (i.e. the same random directions), while the partition procedures set individual shifts of the hyperplanes. In every partition of a node, the division is shifted to keep a balance not worse than β (each branch has a fraction of at least β vectors in a node). Thanks to that the divisions are effective—always divide items of a node in good proportion.

The balanced version of partitioning resembles a typical partition3 operation. This strategy keeps the depth of the tree small. The tree is quite strongly balanced and there are no useless divisions, compared to [19]. Also, thanks to using the same hashes at all nodes of a given level construction of data multiplication by hashes is faster than in [20]. The split of a new node is continued, if their corresponding number of vectors is still too big compared to the desired number of neighbor candidates. The meta-code of the algorithm starts with the construction of random regions tree, Alg. 4.

The non-leaf nodes have their sub-nodes defined. All nodes have a defined interval of data and in case of a leaf it is a bin. Additionally, the pivot point is stored to define the final shift of the random hyperplane. This information is necessary to define a classification process that has to traverse the tree from the root node to an appropriate bin and then select nearest neighbors among the bin items.

Such more careful construction of the RRT is necessary because the tree will be used many times after construction in contrast to the quick sort algorithm where the sort finishes after all partitions.

Algorithm 4: RRTree($H, left, right, bCount, depth$)

```
Data: $H$ hashed data Eq. 2
    $[left, right)$ interval of the current split
    $h$ is maximal depth of a tree Eq. 2
    $depth$ the depth of current tree node
    $ids$ an array of vector indices
    $\beta = .25$ partition threshold of minimal balance
    $\alpha = 1 - (\beta + .5)/2$ threshold: whether to continue splitting
    $bCount$ minimal count of elements in a bin
Result: $ids$ array of bins defined by nodes, initially: $[1, \ldots, m]$
1  begin
2      $div = partition(ids, H, left, right, \beta)$;
3      if $depth + 1 < h \&\& (div - left) * \alpha > bCount$ then
4          $node.left = RRTree(H, left, div, bCount, depth + 1)$
5      else
6          $node.left = newnode(left, div)$;
7      if $depth + 1 < h \&\& (right - div) * \alpha > bCount$ then
8          $node.right = RRTree(H, div, right, depth + 1, bCount)$
9      else
10         $node.right = newnode(div, right)$;
11     return $node$
```

B. Forest of random regions trees and other ingredients of fast DROP algorithms

Because the trees are constructed on the base of sharp planes, it is highly recommended to use a forest of trees to overcome sharp approximation. First it was introduced in [18] and we also decide to use a forest in place of a single tree, indeed we use typically 8 trees in the forest.

However to keep the computational complexity possibly small, besides the trees in the forest we use additional data structures: an array of trees, an array of heaps, an array of hash sets, an array of sets of associate instances, and an array of arrays (stores the nearest neighbors for each instance). The heaps (one per instance) are used to accumulate the candidates found in trees while looking for neighbors. This serves as a cache, making the number of calls to tree searches smaller. The hash sets (one per instance) store the information on whether an instance was previously added to given instances neighbors heap from another tree. Those structures cooperate to keep the complexity possibly small. The utilization of those structures in most important parts of algorithms is presented below.

The initialization of both algorithms starts from building the forest (initiating heaps, sets of associate neighbors and nearest neighbors).

The construction of the forest was presented in Alg. 5. In the first step it constructs the trees, then the heaps are filled with neighbors candidates from leaves. Last part of the forest construction builds structures storing the neighbors of each instance and the associated instance sets ($\mathcal{A}$).

The RefillHeap (Alg. 6) tries to find neighbors from appropriate tree buckets (in each of the trees). Additionally it optimizes the structure of a tree if necessary. The instances from the trees are moved to the heap and an appropriate hash set.

In DROP2-4 algorithms, the instances are being pruned from time to time. Alg. 7 is responsible for pruning of a given instance from the forest (from each of the trees). A pruned instance is removed from the hash set of the instances added to the heap. Additionally, a pruned instance is removed from

Algorithm 5: ForestBuild($D, treecount, k, C$)

```
Data: $D = [instance_1, \ldots, instance_m]$
$treecount$ the number of trees in forest (default 8)
$C$ used to control the number of neighbor candidates
$bCount = C * k/treecount$ minimal number of elements in bin
1  begin
2      for $i \in [1, \ldots, treecount]$ do
3          $H_i = (D + \epsilon_i) \times R_i$;
           $trees[i] = RRTree(H_i, 1, m, bCount)$;
4      for $i \in [1, \ldots, m]$ do
5          RefillHeap(i)
6      for $i \in [1, \ldots, m]$ do
7          for $j \in [k+1, \ldots, 1]$ do
8              neighbor = $minheap_i$.Pop()
9              $neighbors_i$[j] = neighbor
10             $D_{neighbor}$.A.Add(i)
```

Algorithm 6: RefillHeap(id)

```
Data: $id$ - index of instance to refill for
1  begin
2      for $tree \in forest$ do
3          for $candidate \in tree.bucket[id]$ do
4              // bucket is moved up a tree if count in a
                 node is to small
5              // only not pruned are yielded from a bucket
6              if $idx \notin added_{id}$ then
7                  $minheap_{id}$.Push((candidate,
                     Distance(id, candidate)))
8                  $added_{id}$.Add(candidate)
```

an appropriate array of neighbors and a new neighbor has to be found, and the associated structures have to be adjusted as well (see Alg. 8). Those procedures of operating on these structures are similar to those employed by us in [20].

Tree pruning was presented in Algorithm 9. First, an appropriate leaf node is selected, and then conditions are

Algorithm 7: Prune(i)

```
Data: $id$ - index of instance to prune
1  begin
2      $D_{id}$.pruned = True
3      for $tree \in forest$ do
4          tree.Prune(id)
5      for $a \in D_{id}.A$ do
6          $added_a$.Remove(id)
7          ReplaceNeighbor(a, id)
```

Algorithm 8: ReplaceNeighbor

```
Data: $id$ - index of instance to replace neighbor for
$nid$ - index of instance to replace
1  begin
2      $neighbors_{id}$.Remove(nid)
3      do
4          if $minheap_{id}$.isSmall then
5              RefillHeap(id)
6          neighbor = $minheap_{id}$.Pop()
7          if not $D_{neighbor}$.pruned then
8              $neighbors_{id}$[0] = neighbor
9              $D_{neighbor}$.A.Add(id)
10     while $D_{neighbor}$.pruned;
```

Algorithm 9: TreePrune

```
Data: $id$ - index of the instance to be pruned
1  begin
2      node = tree.leaves[id]
3      decrement by one counts in a tree from node up
4      while node.parent has two small children do
5          node=node.parent
6      if node has two small children then
7          rewrite children to node without gaps
8      if node has small density then
9          rewrite node without gaps
```

tested on the path from the selected node to the root in order to optimize the tree structures if necessary (to keep the complexity low). Both the path and the representation of buckets can be compressed (the latter by removal of gaps in the buckets).

The usage of the forest of random regions trees—the nearest neighbors search, can be seen in Alg. 10. First, we collect candidate instances from each tree and then the nearest neighbors are selected among them. Because the number of trees and bin size is $O(1)$, the classification cost is $O(\log m)$ ($O(\log m)$ is the expected length of the longest path from the root node to a leaf).

Now, the computation of the badly-classified instance count, or the calculation of the *improvement* used in DROP2-4 algorithms, based on the fast neighbors search, is much faster.

The DROP2 algorithm needs to start the main loop in decreasing order of distances to the nearest enemy (opposite class instance). Plain calculation of distance between given instance and its nearest enemy is time consuming, but it is presented in Alg. 11 that such distances can be calculated quickly ($O(m \log m)$). The algorithm constructs a jungle as a set of forests: one forest to contain the instances of one class. Then the nearest enemies for all instances are calculated in a total time of $O(m \log m)$. The nearest enemy is the nearest one of nearest neighbors from all classes except the class of

Algorithm 10: NearestNeighbors($\mathbf{x}, trees, k, c$)

```
Data: $\mathbf{x}$ define whose neighbors have to be found
$tree[i], i = 1, \ldots, t$ an array of random trees, $tree[i]$
consists of the root node and $ids$ an array of vector
indices
$k$ desired number of neighbors
Result: $NN$ set of $k$ nearest neighbors
1 begin
2     foreach $T_i$ do
3         $I$ = items of the bin nearest to $\mathbf{x}$ in tree $tree[i]$
4         $N = N \cup I$
5     $NN$ = find $k$ nearest neighbors in $N$
```

Algorithm 11: OrderByNearestEnemy(p)

```
Data: $D = [instance_1, \ldots, instance_m]$
Result: order of $D$ by distance to nearest enemy
1 begin
2     jungle = LSHForest for each class
3     for $i \in [1, \ldots, m]$ do
4         for $forest \in jungle$ do
5             if $forest.class \neq D_i.label$ then
6                 distances[i] = min(distances[i],
                    forest.kNN(i, 1))
7     return Ordering(distances)
```

the given instance. Finally, the estimated complexity DROP algorithms is $O(m \log m)$ what can be seen in next section.

IV. Experimental analysis

We performed an analysis of several aspects of the new algorithms: accuracy, complexity and stability of behavior over different datasets from UCI Machine Learning Repository [21]. In all tests, we used 10-fold stratified cross-validation and all learning machines were learning on the same sets of data partitions.

The first portion of tests compares the original DROP algorithms with the new ones. This, however, enforces using small and medium datasets in this comparison. Next, the second portion of the test concentrates on the big datasets to test the behavior of algorithms with $O(m \log m)$ complexity.

To visualize the performance of all algorithms we present average accuracy for each benchmark dataset and for each learning machine. Additionally, we present the average reduction of dataset size in separate tables. ***Ranks*** are calculated for each machine for a given dataset $\mathcal{D}$. The ranks are calculated as follows: First, for a given benchmark dataset $\mathcal{D}$ the averaged accuracies of all learning machines are sorted in descending order. The machine with the highest average accuracy is ranked 1. Then, the following machines in the accuracy order whose accuracies are not statistically different from the result of the first machine are ranked 1, until a machine with a statistically different result is encountered. That machine starts the next rank group (2, 3, and so on), and an analogous process is repeated on the remaining (yet unranked) machines. Notice that each cell of the main part of Table I is in a form: $acc + std(rank)$, where acc is average accuracy (for a given data set and given learning machine), std is its standard deviation and $rank$ is the rank described just above. If a given cell of the table is in bold it means that this result is the best for given data set or not worse than the best one (rank 1 = winners).

It can be seen that the results of fast DROP are even slightly better than of the original algorithms. The mean rank for fast DROP3 is 1.4 and for original DROP3 1.56 (smaller=better), similarly the numbers of wins are 31 (fast) and 24 (original), and the fast DROP3 has 6 unique wins as well. Similar behavior can be observed on DROP4. The mean rank of the fast version is 1.42 and it is 1.53 for the original. The number of wins was greater for the fast version: 32 compared to 29. Concluding, the fast versions perform slightly better than original versions.

Figure 1 presents an analysis of learning time used by the fast and original DROP algorithms. We have tested the time for different numbers of instances of the MNIST8M dataset [22]. On the OX axis is the number of instances. The OY axis of the upper plot is time and on the lower plot OY is the proportion of time to the number of instances. Both plots clearly show that the LSH-based versions are much faster. Additionally, the bottom plot shows that the estimated complexity of LSH versions is $O(m \log m)$, because in the case of fast DROP algorithms the plot lines are straight (plot of the $\log$[1] with a log-scaled OX is a straight line). Meanwhile, looking at the upper plot, we see the computation times for the original DROP grow very quickly.

The next part of the analysis concentrates on selected big datasets. In three tables we present the analysis of the accuracy Table II, of the redundancy Table III and a combination of accuracy and redundancy Table IV. The performance of LSH-DROP3 and LSH-DROP4 is compared to a few other algorithms: LSH-IS-S, LSH-IS-F [13], PSC [14] and LSH-kNN (this is a kNN which uses our RRT forest for neighbor searching). The highest performance is achieved for LSH-kNN: mean rank is 1.3 and the number of wins is 7. Among the instance selection algorithms, the smallest mean rank of 2.2 was achieved by LSH-DROP3. The greatest number of wins (3) among instance selections was achieved by LSH-IS-S. PSC had the worst performance, with a mean rank of 4.1.

But it is interesting to compare the above results with reduction strength. The reduction is not very large in the case of LSH-IS algorithms. We have tested several different configurations of LSH-IS and only the best ones are presented. We have selected 4 best configurations to show how it changes, depending on parametrization. For the meaning of LSH-IS parameters we strongly recommend to read [13]. The difference is average reduction between LSH-IS and LSH-DROP algorithms is vast: the reduction of LSH-IS varies from

[1]log because we divide time by the number of instances.

TABLE I
ACCURACIES COMPARISON FOR DROP AND LSH-DROP ALGORITHMS.

	Drop3	LSHDrop3	Drop4	LSHDrop4
arrhythmia	**52.9±21(1)**	**53.5±20(1)**	**54.3±23(1)**	**51.7±22(1)**
autos	56.7±11(2)	57.6±11(2)	**64.9±12(1)**	**65.4±12(1)**
balance-scale	80.4±4.1(2)	**82.7±3.7(1)**	79.8±4.3(3)	81±4.6(2)
blood-transfusion	75.1±4.6(2)	**76.1±4.7(1)**	71.3±5.8(3)	72±5.9(3)
breast-cancer-diagnostic	93.6±2.8(2)	**94.4±3(1)**	93.4±3.2(2)	**94.1±2.6(1)**
breast-cancer-original	**95.2±2.5(1)**	**95.2±2.2(1)**	94.7±2.8(2)	**95.1±2.2(1)**
breast-cancer-prognostic	72.4±9.9(2)	**74.4±8.2(1)**	67±10(4)	68.9±9.4(3)
breast-tissue	63.2±14(2)	**64.2±13(1)**	**65.7±14(1)**	**64.3±14(1)**
car-evaluation	79.5±2.8(2)	**86.2±2.7(1)**	79.8±2.8(2)	**86.5±2.5(1)**
cardiotocography-1	**70.9±2.9(1)**	**70.7±3.1(1)**	**71.1±3(1)**	**71±3.2(1)**
cardiotocography-2	87.8±1.8(2)	**88.3±1.8(1)**	87.3±2(3)	**88±2.3(1)**
chess-rook-vs-pawn	90.5±1.6(2)	89.6±1.7(3)	**91±1.6(1)**	90±1.8(3)
cmc	**45.2±3.4(1)**	**45.1±3.9(1)**	43±3.9(2)	42.9±3.9(2)
congressional-voting	**91.1±5.5(1)**	89.3±6.2(2)	89.3±7.3(2)	89.1±6.8(2)
connectionist-bench-sonar	78.7±8.9(2)	78.4±9(2)	**80.5±7.9(1)**	**80.5±7.6(1)**
connectionist-bench-vowel	94.4±3.7(3)	93.8±4(3)	**96±3(1)**	95.3±3.5(2)
cylinder-bands	60.9±8.3(2)	61.5±8.4(2)	**62.7±8.6(1)**	**62.8±9(1)**
dermatology	**87.3±4.3(1)**	**87.6±5.6(1)**	**87.6±4.7(1)**	**87.7±5.1(1)**
ecoli	**84.8±5.2(1)**	**84.4±5.3(1)**	**84.1±4.8(1)**	**83.8±5.7(1)**
glass	**68.3±8.4(1)**	**67.4±8.6(1)**	**67.2±9.4(1)**	**67.9±9.7(1)**
habermans-survival	**69.5±7(1)**	**70.1±7.1(1)**	67.8±6.9(2)	**68.7±6.5(1)**
hepatitis	**81.4±13(1)**	**82.3±12(1)**	**82.4±13(1)**	**83.4±12(1)**
ionosphere	**84±5(1)**	**84.1±4.9(1)**	80.5±7.4(2)	**83.2±5.5(1)**
iris	**93.9±6.2(1)**	**94.1±6.1(1)**	**93.9±6.2(1)**	**94.3±5.6(1)**
libras-movement	76.6±6.2(3)	76.4±6.1(3)	**81.5±6.4(1)**	80.2±5.3(2)
liver-disorders	**59±8(1)**	**58.8±8.3(1)**	**60.3±8.2(1)**	**59.7±7.9(1)**
lymph	75.5±11(2)	**76.1±12(1)**	**77.2±11(1)**	**75.6±12(1)**
monks-problems-1	**94.7±2.9(1)**	**94.8±3(1)**	**94.6±2.9(1)**	**95.1±3.1(1)**
monks-problems-2	**58.7±5(1)**	55.7±6.2(3)	57.4±6.2(2)	53.7±6.6(4)
monks-problems-3	**93.4±3.5(1)**	**93.5±3.5(1)**	**93.4±3.5(1)**	**93.5±3.6(1)**
parkinsons	**86.9±7.6(1)**	86±7.7(2)	**87.8±7.4(1)**	**88.2±7.3(1)**
pima-indians-diabetes	**72.1±5.5(1)**	**72±5.1(1)**	70.4±5(2)	70.5±5.2(2)
sonar	78.7±8.9(2)	78.9±8.4(2)	**80.5±7.9(1)**	**79.8±8(1)**
spambase	88±1.8(2)	**88.4±1.7(1)**	87.6±1.6(3)	87.8±1.6(2)
spect-heart	75.5±8.4(2)	75.2±8.8(2)	**77.6±7.7(1)**	**77.6±7.5(1)**
spectf-heart	**69.5±7.8(1)**	**69.7±8.4(1)**	**68.2±8.9(1)**	**69.2±8.6(1)**
statlog-australian-credit	**77.5±4.9(1)**	**77.7±4.9(1)**	**77.7±5.7(1)**	**77.3±5.1(1)**
statlog-german-credit	68.1±4(2)	**69.4±4.1(1)**	66.2±4.4(3)	66.4±5.2(3)
statlog-heart	**76.4±7.6(1)**	**76.1±7.3(1)**	**75.8±7.5(1)**	**76.1±7.5(1)**
statlog-vehicle	**68.2±4.8(1)**	**67.8±4(1)**	**67.4±4.4(1)**	**67.4±4.5(1)**
teaching-assistant	40.9±11(3)	42.8±12(2)	**45.5±13(1)**	**45.8±12(1)**
thyroid-disease	90.7±2.3(3)	**93.6±0.79(1)**	90.7±1.5(3)	91.8±0.95(2)
vote	**90.8±6.4(1)**	**91.2±6(1)**	**90.9±5.5(1)**	**90±6.5(1)**
wine	**93.4±5.9(1)**	92.1±6.1(2)	**93.5±5.7(1)**	**92.6±6.1(1)**
zoo	45±13(2)	44.4±12(2)	**49.3±13(1)**	**48.3±12(1)**
Mean	76.4±6.5	76.7±6.5	76.6±6.7	76.8±6.6
Mean Rank	1.56±0.099	1.4±0.099	1.53±0.12	1.42±0.11
Wins[unique]	24[2]	31[6]	29[3]	32[0]

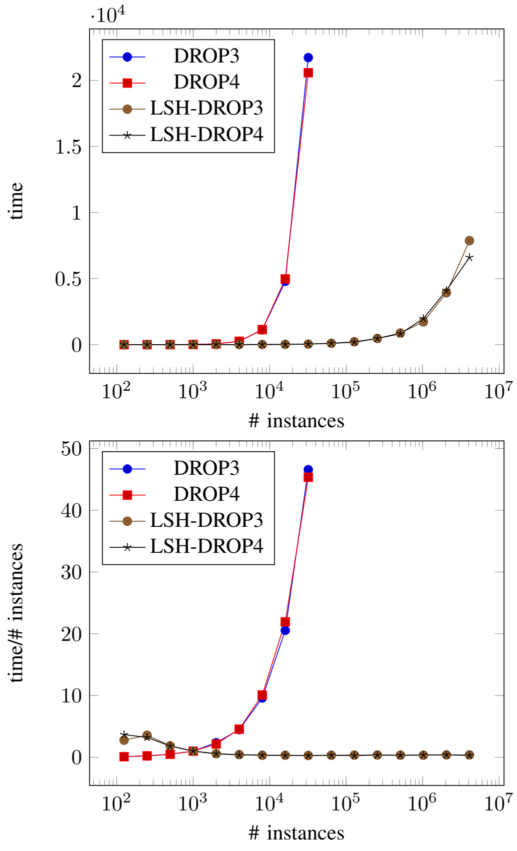

Fig. 1. $O(m \log m)$ time consumption of LSH versions of DROP algorithms.

0.45 to 0.52 and the reduction of LSH-DROP varies from 0.82 to 0.84—this is a huge difference. Of course it was possible to configure LSH-IS algorithms for higher reduction, but it would be at the expense of accuracy. In cases where LSH-IS have better accuracies than LSH-DROP algorithms, at the same time, LSH-IS has a far lower reduction than LSH-DROP—compare results for penbased and poker datasets. Similar behavior was observed in the case of small and middle datasets and we did not find a parametrization that kept both high accuracy and a good level of reduction for LSH-IS.

The average reduction of the PSC algorithm was a little smaller than those of LSH-DROP algorithms.

Table IV presents a combination of accuracy and reduction. The sequences of accuracies and reductions (for each dataset and for each learning machine separately) were multiplied and analyzed in the same way as accuracy and reduction. This test favors those machines which have high accuracy whilst maintaining a high reduction at the same time. In this case, LSH-DROP3 is a very clear winner. LSH-DROP3 has the number of wins equal to 9 (of 10) and the mean rank was 1.1, far superior to the rest of the algorithms.

In the last analysis we focus on complexity and reduction for various numbers of instances (spanning from 125 to 4M). The test was performed on the MNIST8M dataset [22]. Results are shown on Figure 2 The first sub-figure presents time relations of algorithms on a log-log scale. It is clear that the slowest is the PSC algorithm, except for very small datasets. In the middle are two LSH-DROP algorithms and the fastest are the LSH-IS-S and the LSH-IS-F algorithms. The LSH-IS algorithm is up to 15 times faster than DROP algorithms. In the middle sub-figure on the OY axis is the proportion of time to the number of instances. With this figure, it is easy to see whether the complexity of the algorithm is $O(m \log m)$,

TABLE II
ANALYSIS OF ACCURACIES FOR BIG DATASETS.

	LSH-IS-S 4OR	LSH-IS-F 4OR	LSH-IS-S 6OR	LSH-IS-F 5OR	**LSH-Drop3**	**LSH-Drop4**	PSC	LSHkNN
Penbased	**99.34±0.24(1)**	**99.34±0.22(1)**	**99.37±0.19(1)**	**99.35±0.23(1)**	98.44±0.41(2)	98.55±0.39(2)	95.85±0.61(3)	**99.41±0.24(1)**
Nursery	83.22±1.4(2)	82.96±1.1(2)	83.6±1.4(2)	82.61±1.3(2)	77.99±1.3(3)	78.03±1.1(3)	**86.62±2.5(1)**	82.71±0.75(2)
Magic	71.18±1.1(5)	72.65±1.4(4)	72.42±1.2(4)	73.35±1.2(3)	**81.31±1.1(1)**	80.28±1.1(2)	69.18±1(6)	**82±0.34(1)**
Letter	95.15±0.33(2)	94.16±0.44(4)	**95.51±0.43(1)**	94.44±0.37(3)	91.99±0.62(5)	91.9±0.52(5)	90.52±0.85(6)	95.25±0.48(2)
KR vs. K	51.32±0.55(5)	52.28±0.95(4)	51.33±0.6(5)	52.37±0.97(4)	54.65±0.49(2)	54.29±0.63(3)	**55.19±0.86(1)**	**55.4±1(1)**
Census	91.99±0.12(5)	92.33±0.13(3)	92.07±0.094(4)	92.36±0.13(3)	**93.61±0.091(1)**	92.42±0.19(3)	69.68±0.69(6)	93±0.14(2)
KDDCup99	99.88±0.015(4)	99.89±0.016(3)	99.89±0.02(3)	99.9±0.021(3)	99.91±0.017(2)	99.92±0.012(2)	98.82±0.94(5)	**99.96±0.0077(1)**
CovType	82.21±0.58(6)	82.7±0.59(5)	84.21±0.38(3)	83.75±0.43(4)	90.93±0.092(2)	90.9±0.13(2)	82.28±0.23(5)	**93.8±0.1(1)**
KDDCup99.1M	99.75±0.068(4)	99.77±0.058(4)	99.8±0.062(3)	99.81±0.051(3)	99.96±0.015(2)	99.95±0.014(2)	98.62±0.42(5)	**99.98±0.0032(1)**
Poker	**59.36±0.33(1)**	**59.32±0.2(1)**	**59.28±0.17(1)**	**59.3±0.36(1)**	56.53±0.23(2)	55.72±0.21(3)	55.3±1.5(3)	**59.21±0.3(1)**
Mean	83.34±0.47	83.54±0.51	83.75±0.45	83.72±0.51	84.53±0.44	84.2±0.43	80.21±0.96	86.07±0.34
Mean Rank	3.5±0.61	3.1±0.46	2.7±0.47	2.7±0.35	2.2±0.38	2.7±0.32	4.1±0.66	1.3±0.16
Wins[unique]	2[0]	2[0]	3[1]	2[0]	2[1]	0[0]	2[1]	7[3]

TABLE III
ANALYSIS OF REDUCTION FOR BIG DATASETS.

	LSH-IS-S 4OR	LSH-IS-F 4OR	LSH-IS-S 6OR	LSH-IS-F 5OR	**LSH-Drop3**	**LSH-Drop4**	PSC
Penbased	0.19±0.02(4)	0.2±0.02(3)	0.13±0.009(6)	0.16±0.01(5)	0.94±0.002(2)	0.93±0.001(2)	**0.94±0.002(1)**
Nursery	0±0(3)	0±0(3)	0±0(3)	0±0(3)	0.8±0.008(2)	0.79±0.006(2)	**0.9±0.006(1)**
Magic	0.82±0.01(3)	0.84±0.01(2)	0.79±0.01(4)	0.82±0.01(3)	**0.88±0.003(1)**	0.84±0.004(2)	0.75±0.003(5)
Letter	0.36±0.02(6)	0.49±0.02(4)	0.27±0.02(7)	0.44±0.02(5)	**0.81±0.003(1)**	0.81±0.003(2)	0.75±0.006(3)
KR vs. K	0.27±0.04(4)	0.47±0.03(2)	0.18±0.03(5)	0.41±0.02(3)	**0.5±0.002(1)**	0.49±0.002(2)	0.4±0.007(3)
Census	0.32±0.003(5)	0.32±0.003(4)	0.31±0.002(7)	0.32±0.003(6)	**0.96±0.0007(1)**	0.94±0.0004(2)	0.93±0.0002(3)
KDDCup99	0.98±0.001(4)	0.98±0.001(3)	0.97±0.001(6)	0.97±0.0008(5)	0.99±0.0002(2)	0.99±0.0003(2)	**1±0.0001(1)**
CovType	0.88±0.009(2)	**0.9±0.008(1)**	0.85±0.008(4)	0.88±0.007(3)	0.84±0.001(5)	0.82±0.001(6)	0.78±0.001(7)
KDDCup99.1M	0.98±0.0009(4)	0.98±0.0009(3)	0.98±0.001(6)	0.98±0.0007(5)	1±0.0001(2)	1±0.0002(2)	**1±0.0002(1)**
Poker	0.002±2E-05(4)	0.002±2E-05(4)	0.002±2E-05(4)	0.002±2E-05(4)	**0.65±0.0007(1)**	0.59±0.001(2)	0.46±0.004(3)
Mean	0.48±0.01	0.52±0.01	0.45±0.008	0.5±0.008	0.84±0.002	0.82±0.002	0.79±0.003
Mean Rank	3.9±0.4	2.9±0.3	5.2±0.5	4.2±0.4	1.8±0.4	2.4±0.4	2.8±0.7
Wins[unique]	0[0]	1[1]	0[0]	0[0]	5[5]	0[0]	4[4]

TABLE IV
ANALYSIS OF REDUCTION*ACCURACIES FOR BIG DATASETS.

	LSH-IS-S 4OR	LSH-IS-F 4OR	LSH-IS-S 6OR	LSH-IS-F 5OR	**LSH-Drop3**	**LSH-Drop4**	PSC
Penbased	0.19±0.02(4)	0.2±0.02(3)	0.13±0.009(6)	0.16±0.01(5)	**0.92±0.004(1)**	**0.92±0.004(1)**	0.91±0.006(2)
Nursery	0±0(3)	0±0(3)	0±0(3)	0±0(3)	0.62±0.01(2)	0.62±0.008(2)	**0.78±0.02(1)**
Magic	0.59±0.008(5)	0.61±0.01(3)	0.57±0.008(6)	0.6±0.007(4)	**0.72±0.01(1)**	0.67±0.009(2)	0.52±0.009(7)
Letter	0.34±0.02(6)	0.46±0.02(4)	0.26±0.02(7)	0.42±0.02(5)	**0.75±0.006(1)**	0.74±0.004(2)	0.68±0.006(3)
KR vs. K	0.14±0.02(5)	0.24±0.02(3)	0.091±0.01(6)	0.21±0.01(4)	**0.27±0.003(1)**	0.26±0.003(2)	0.22±0.003(4)
Census	0.3±0.003(5)	0.3±0.003(4)	0.29±0.002(7)	0.29±0.003(6)	**0.9±0.001(1)**	0.87±0.002(2)	0.65±0.006(3)
KDDCup99	0.97±0.001(3)	0.97±0.001(2)	0.97±0.001(5)	0.97±0.0008(4)	**0.99±0.0002(1)**	**0.99±0.0003(1)**	**0.99±0.009(1)**
CovType	0.73±0.002(5)	0.74±0.002(3)	0.72±0.004(6)	0.74±0.003(4)	**0.76±0.002(1)**	0.75±0.001(2)	0.64±0.002(7)
KDDCup99.1M	0.98±0.001(3)	0.98±0.001(3)	0.98±0.001(5)	0.98±0.0008(4)	**0.99±0.0002(1)**	**0.99±0.0002(1)**	0.98±0.004(2)
Poker	0.0012±1E-05(4)	0.0012±1E-05(4)	0.0012±1E-05(4)	0.0012±2E-05(4)	**0.37±0.001(1)**	0.33±0.001(2)	0.26±0.008(3)
Mean	0.42±0.007	0.45±0.008	0.4±0.006	0.44±0.006	0.73±0.004	0.72±0.003	0.66±0.008
Mean Rank	4.3±0.4	3.2±0.2	5.5±0.4	4.3±0.3	1.1±0.1	1.7±0.2	3.3±0.7
Wins[unique]	0[0]	0[0]	0[0]	0[0]	9[6]	3[0]	2[1]

because in the case of $O(m \log m)$ complexity the plot lines should be straight (should not grow faster than linearly). Looking at the plots, it is clear that the LSH-DROP3-4 and LSH-IS-F has $O(m \log m)$ complexity.

The third sub-figure of Figure 2 clearly shows a strange behavior of the LSH-IS algorithm in terms of reduction. This means that the configuration of LSH-IS should be tuned for every number of instances to keep an adequate reduction rate. The reductions of LSH-DROP3-4 are much greater.

V. SUMMARY

The new proposed versions of DROP algorithms are much faster than the existing ones. The complexity of the fast version is $O(m \log m)$, while the complexity of the original DROP algorithm is $O(m^3)$ (estimated complexity is over $O(n^2)$). The new, fast versions of DROP algorithms proved themselves, offering a good balance between high accuracy and a strong reduction of dataset volume. Experimental results clearly prove their attractiveness.

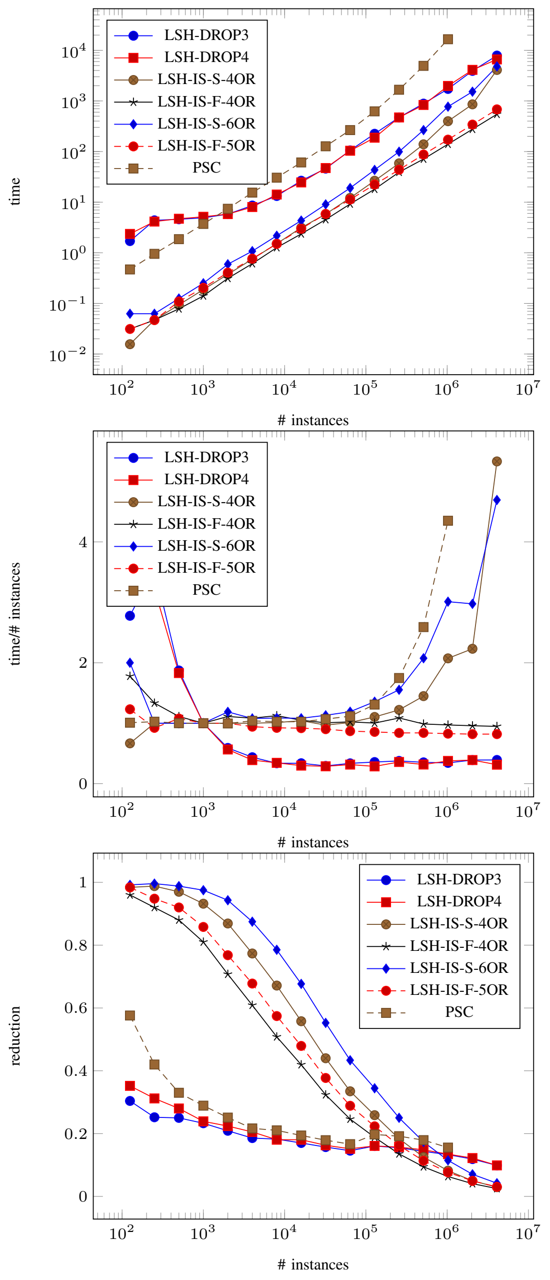

Fig. 2. Analysis of complexity and reduction.

The LSH-DROP algorithms are much more stable than LSH-IS algorithms, however LSH-DROP algorithms are slower. There is no tricky selection of configuration parameters in the case of LSH-DROP. The LSH-DROP algorithms have a good reduction factor while the accuracies remain very close to the accuracies of kNN.

References

[1] T. M. Cover and P. E. Hart, “Nearest neighbor pattern classification,” *Institute of Electrical and Electronics Engineers Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, Jan. 1967.

[2] S. Garcia, J. Derrac, J. Cano, and F. Herrera, “Prototype selection for nearest neighbor classification: Taxonomy and empirical study,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 3, pp. 417–435, 2012.

[3] N. Jankowski and M. Grochowski, “Comparison of instances selection algorithms: Ii. Algorithms survey,” in *Artificial Intelligence and Soft Computing*, ser. Lecture Notes in Computer Science, L. Rutkowski, J. H. Siekmann, R. Tadeusiewicz, and L. A. Zadeh, Eds. Poland, Zakopane: Springer-Verlag, 2004, vol. 3070, pp. 598–603. [Online]. Available: http://www.is.umk.pl/ norbert/publications/04-zakopane-NJMG.pdf

[4] M. Blachnik, *Metody bazujące na prototypach w zastosowaniu do eksploracji danych*. Silesian Technical University, 2019.

[5] M. Kordos, “Optimization of evolutionary instance selection,” in *Artificial Intelligence and Soft Computing*, ser. Lecture Notes in Artificial Intelligence. Springer-Verlag, 2017, vol. 10245, pp. 359–369.

[6] M. Blachnik and W. Duch, “LVQ algorithm with instance weighting for generation of prototype-based rules,” *Neural Networks*, vol. 24, no. 8, pp. 824–830, 2011.

[7] R. M. Cameron-Jones, “Instance selection by encoding length heuristic with random mutation hill climbing,” in *Proceedings of the Eighth Australian Joint Conference on Artificial Intelligence*, Australia, 1995, pp. 99–106.

[8] D. S. Broomhead and D. Lowe, “Multivariable functional interpolation and adaptive networks,” *Complex Systems*, vol. 2, no. 3, pp. 321–355, 1988.

[9] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, “Extreme learning machine: a new learning scheme of feedforward neural networks,” in *International Joint Conference on Neural Networks*. Budapest, Hungary: IEEE Press, 2004, pp. 985–990.

[10] ——, “Extreme learning machine: theory and applications,” *Neurocomputing*, vol. 70, no. 1–3, pp. 489–501, 2006.

[11] N. Jankowski, “Comparison of prototype selection algorithms used in construction of neural networks learned by SVD,” *International Journal of Applied Mathematics and Computer Science*, vol. 28, no. 4, pp. 719–733, 2018. [Online]. Available: https://www.amcs.uz.zgora.pl/?action=paper&paper=1464

[12] D. R. Wilson and T. R. Martinez, “Reduction techniques for instance-based learning algorithms,” *Machine Learning*, vol. 38, no. 3, pp. 257–286, 2000.

[13] Álvar Arnaiz-González, J.-F. Díez-Pastor, J. J. Rodríguez, and C. García-Osorio, “Instance selection of linear complexity for big data,” *Knowledge-Based Systems*, vol. 107, pp. 83–95, 2016.

[14] J. A. Olvera-López, J. A. Carrasco-Ochoa, and J. F. Martínez-Trinidad, “A new fast prototype selection method based on clustering,” *Pattern Analysis and Applications*, vol. 13, no. 2, pp. 131–141, 2009.

[15] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis, *R-Trees: Theory and Applications*. Springer, 2006.

[16] P. Yianilos, “Data structures and algorithms for nearest neighbor search in general metric spaces,” in *In Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 1993, pp. 311–321.

[17] R. Brown, “Building a balanced k-d tree in $O(kn \log n)$ time,” *Journal of Computer Graphics Techniques*, vol. 4, no. 1, pp. 50–68, 2015.

[18] M. Bawa, T. Condie, and P. Ganesan, “LSH forest: self-tuning indexes for similarity search,” in *Proceedings of the 14th international conference on World Wide Web*, Chiba, Japan, 2005, pp. 651–660.

[19] S. Har-Peled, P. Indyk, and R. Motwani, “Approximate nearest neighbor: Towards removing the curse of dimensionality,” *Theory of computing*, vol. 8, pp. 321–350, 2012.

[20] N. Jankowski and M. Orliński, “Fast algorithm for prototypes selection—trust-margin prototypes,” in *Artificial Intelligence and Soft Computing*, ser. Lecture Notes in Computer Science, L. Rutkowski, R. Scherer, M. Korytkowski, W. Pedrycz, R. Tadeusiewicz, and J. Zurada, Eds., vol. 11508. Springer, 2019, pp. 583–594.

[21] C. J. Merz and P. M. Murphy, “UCI repository of machine learning databases,” 1998, http://www.ics.uci.edu/∼mlearn/MLRepository.html.

[22] G. Loosli, S. Canu, and L. Bottou, “Training invariant support vector machines using selective sampling,” in *Large Scale Kernel Machines*, L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, Eds. Cambridge, MA.: MIT Press, 2007, pp. 301–320.