Applied Intelligence (2024) 54:12558–12575
https://doi.org/10.1007/s10489-024-05763-w

Improved KD-tree based imbalanced big data classification and oversampling for MapReduce platforms

William C. Sleeman IV[1] · Martha Roseberry[2] · Preetam Ghosh[2] · Alberto Cano[2] · Bartosz Krawczyk[3]

Accepted: 11 August 2024 / Published online: 18 September 2024

Abstract
In the era of big data, it is necessary to provide novel and efficient platforms for training machine learning models over large volumes of data. The MapReduce approach and its Apache Spark implementation are among the most popular methods that provide high-performance computing for classification algorithms. However, they require dedicated implementations that will take advantage of such architectures. Additionally, many real-world big data problems are plagued by class imbalance, posing challenges to the classifier training step. Existing solutions for alleviating skewed distributions do not work well in the MapReduce environment. In this paper, we propose a novel KD-tree based classifier, together with a variation of the SMOTE algorithm dedicated to the Spark platform. Our algorithms offer excellent predictive power and can work simultaneously with binary and multi-class imbalanced data. Exhaustive experiments conducted using the Amazon Web Service platform showcase the high efficiency and flexibility of our proposed algorithms.

Keywords Apache Spark · Amazon web services · Imbalanced data · k-dimensional trees · Machine learning · SMOTE

1 Introduction

Modern systems increasingly generate massive amounts of data, driving the desire and necessity to have algorithms that can learn from big data. The data mining community has developed, and continues to develop, many algorithms that are capable of learning from big data, but the ever increasing volume of data still presents challenges. Datasets may easily be larger than is possible to store on a single machine, or data may arrive continuously as an infinite stream, requiring rapid processing. For much big data, distributed and parallel processing is essential, which has led to an increase in distributed platforms.

The Message Passing Interface (MPI) is a well established framework for distributed computing which provides a communication protocol for writing parallel applications using distributed memory. While MPI gives a high level of control on how computational tasks are scheduled and data is transferred, it comes with a cost. The software developer is solely responsible for dispatching tasks, combining partial results, and managing slow or failed nodes from a potentially heterogeneous distributed environment. MapReduce platforms, such as Apache Hadoop and Spark [1], provide more generalized operations tailored for distributed computing. Unlike Hadoop, Spark uses in-memory computations to achieve higher performance. The Spark API facilitates the partitioning and communication required for distributed computing, along with being explicitly fault-tolerant.

One popular approach used to learn from these large datasets is the k-nearest neighbor (kNN) algorithm. This algorithm works by returning the k-nearest examples from the training dataset compared to an incoming query example. This simple algorithm can be used for both supervised and unsupervised problems but does not scale well computationally if the training dataset is large. To solve this problem,

✉ William C. Sleeman IV
fsleeman@gmail.com

Martha Roseberry
mroseberry@vcu.edu

Preetam Ghosh
pghosh@vcu.edu

Alberto Cano
acano@vcu.edu

Bartosz Krawczyk
bartosz.krawczyk@rit.edu

[1] Department of Radiation Oncology, Virginia Commonwealth University, Richmond, Virginia, USA

[2] Department of Computer Science, Virginia Commonwealth University, Richmond, Virginia, USA

[3] Center for Imaging Science, Rochester Institute of Technology, Rochester, New York, USA

approximate algorithms are often used to significantly speed up execution time, at a potential cost to accuracy.

Another issue common to big data problems is class imbalance. Many algorithms tend to favor the class with the most examples, also known as the majority class. This problem is compounded when the ratio of imbalance between classes is very large or there are many different classes present. One solution is to simply balance the dataset before preforming classification, often done by oversampling the smaller classes. The SMOTE algorithm [2] is a commonly used method which creates new artificial examples based on the existing ones to increase the size of the minority classes.

In this paper, we present a new approximate kNN algorithm based on the k-dimensional tree (KD-tree) data structure. This algorithm was implemented using the Scala programming language for the Apache Spark platform. To evaluate the strengths and weaknesses of this method, it is compared to an existing hybrid-spill tree implementation [3]. The relationship between tree leaf size and both running time and classifier accuracy is also investigated. Scalability of these two implementations are also evaluated on the Amazon Web Service (AWS) distributed computing environment. Additionally, to cope with skewed distributions of classes that are common in real-world big data, we present a novel algorithm to improve the SMOTE oversampling for both binary and multi-class imbalanced datasets using the KD-tree data structure.

This paper is organized as follows. Section 2 presents a background in k-nearest neighbor algorithms and the Apache Spark platform. Our proposed method and experimental setup is provided in Sections 3 and 4 with results presented in Section 5. Concluding remarks are given in Section 6.

2 Background

The k-nearest neighbors algorithm (kNN) is a lazy learner in which the predicted class for a query example is determined based on the classes of the k reference or training examples nearest to it. While quite simple, kNN can be highly accurate and is often used for classification problems. However, given a set of reference points R with n examples and a set of query points Q with m examples, kNN requires $\mathcal{O}(nm)$ time to compute the distances between all the query points and reference points. A common variation of kNN, the all-nearest neighbors problem, represents the case where R and Q are the same set, and in this case the complexity is $\mathcal{O}(n^2)$. For very large or high-dimensional datasets, this becomes infeasible. Different techniques have been developed to focus on addressing the curse of dimensionality [4, 5], classification [6, 7], using approximate nearest neighbors algorithms [8, 9], and developing parallel implementations of kNN [10, 11]. Some recent works [12, 13] have shown good results with the k-nearest neighbor based structural twin support vector machine (KNN-STSVM) [14] which pairs the traditional kNN and SVM algorithms.

2.1 Approximate *k*NN algorithms

2.1.1 Metric tree

A metric tree [15], shown in Fig. 1, is a binary tree that splits the node data, $N(v)$, based on a left and right pivot point, denoted as $v.lc$ and $v.rc$. The pivot points ideally have the maximum distance between any pair of examples in $N(v)$ but because this leads to $\mathcal{O}(n^2)$ time, the points are often chosen heuristically. A vector, μ, is drawn between the pivot points and the mid-point is used for splitting the data, where examples on each side are assigned to the corresponding child tree. At each node, the hypersphere radius is also calculated from the maximum distance between all points which can be used for backtracking. If a given query point or its nearest neighbor candidates falls within the radius around a pivot point, backtracking on the branch will be performed. Building the tree requires $\mathcal{O}(dn \log n)$ time, where d is the number of dimensions, but in practice will end early if the specified leaf size is greater than one. The total space required is $\mathcal{O}(n)$ and each query takes $\mathcal{O}(d \log n)$ time.

2.1.2 Spill tree

A spill-tree [16] is a variant of the metric-tree that instead of partitioning data at the middle point, uses a buffer of width τ to include additional examples from each side of the split as shown in Fig. 2. This can help to alleviate some classification errors when query points are close to the border. The value of τ is estimated using the approximate average distance from each example to their nearest neighbor [17]. Like the metric tree, the spill tree also has a build time of $\mathcal{O}(dn \log n)$ and a query time $\mathcal{O}(n)$. However, for any τ greater than zero, the additional examples included at each split will require more space than the basic metric tree. As the value of τ gets larger, the chances of finding the true nearest neighbors increase but so does the exclusion time and memory.

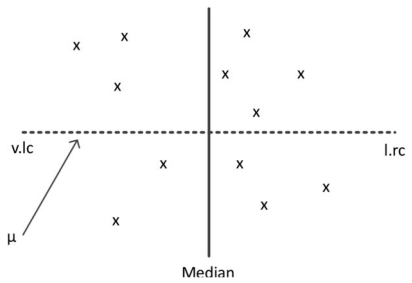


Fig. 1 Example of the partitioning of a metric tree

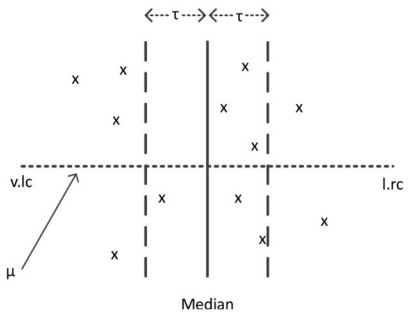


Fig. 2 Example of the partitioning of the spill tree. Each child tree will include examples in the opposing buffer region of width τ

2.1.3 KD-tree

As shown in Fig. 3, the KD-tree [18] is a binary tree that splits data at each node based on the median value of a feature, also called an axis. The KD-tree is often fast, as it does not need to perform $\ell 1$ or $\ell 2$ distance calculations on each examples at every node. The sorting portion of the KD-tree training is the limiting factor but is not dependent on the number of features, which is very beneficial for datasets with high dimensionality. The KD-tree takes $\mathcal{O}(n \log^2 n)$ time as the data at each split needs to be sorted, although it is not dependent on the number of dimensions. The total space required is $\mathcal{O}(n)$ and each query takes $\mathcal{O}(\log n)$ time.

2.1.4 Locality Sensitive Hashing (LSH)

Unlike the other described methods which partition data using a tree structure, LSH [19, 20] uses hashing which can work well with very high-dimensional data including images [21]. The idea is to use a hash function that assigns nearby examples to the same bucket. This approach can also generate multiple hash tables using different hashing functions and querying can build a set of close neighbors from each table. LSH has a training time of $\mathcal{O}(n \log n)$ and a query time of $\mathcal{O}(\log n)$. The hash tables require $\mathcal{O}(n)$ space for each table created.

2.2 Hybrid-spill tree implementation

In addition to these traditional data structures, Liu et al. [16, 22] introduced a improved data structure call a hybrid-spill tree which is a combination the traditional metric and spill-trees. Child trees are created using spill-trees unless one of the children is larger than ρ percent of the parent node's data. The default value of ρ is 0.7 and was unchanged in all experiments presented in this paper. If the ratio is larger than ρ, the spill-tree is removed and replaced with a metric tree. This helps to keep the overall tree balanced and ensures a $\mathcal{O}(\log n)$ depth. Branching of the hybrid-spill tree is halted when the number of examples before splitting is less than a user defined leaf size. Backtracking is performed on metric-tree nodes, but not on spill-tree nodes as the τ buffer is already including candidate examples that belong to an adjacent branch.

The combination of these two approaches was shown to have a good trade off between accuracy and runtime performance. The worse case runtime and space performance is the same as the metric tree and, assuming the tree is balanced, querying for the nearest neighbor is time $\mathcal{O}(m \log n)$.

To further aid the Spark based parallelism, the hybrid-spill tree itself is distributed. The training data is partitioned to create a shallow, top level metric-tree and for each leaf a hybrid-tree is created. When querying is performed, each example goes through the top metric-tree but without the backtracking. Instead, the top level search is done in parallel to avoid multiple iterations. If the query point is close to the boundary, both branches are traversed. This results in nearest neighbor candidates for each top tree traversed and the best overall k-nearest neighbors are returned. An existing implementation [3] of the hybrid-spill tree was used as a benchmark for the classification experiments in Section 4.

2.3 Apache spark

As the distance computation between any two points is entirely independent of any other distance computation, kNN can easily benefit from parallel and distributed implementations. Spark is a framework for parallel computing that was specifically designed for iterative computation, such as is common in machine learning and data mining algorithms. Because of this, Spark is an ideal platform for kNN [23–25].

Apache Spark is a popular implementation of the MapReduce architecture which provides a scalable platform for big data operations [26]. This model has three main phases: map, shuffle, and reduce. As shown in Fig. 4, data is first mapped to distributed computational nodes to provide parallelism. Intermediate results from the map step are shuffled, i.e. reordered or grouped, and then sent to the reduce phase where the final results are prepared. Unlike Apache Hadoop, another popular open source implementation on MapReduce, Spark stores intermediate results in main memory when possible and maximizes the available computational power by optimizing the DAG of dependent tasks.

Figure 5 shows how resources are managed and data is shared on a Spark cluster. On the Driver Node, the Spark Context object creates a Cluster Manager, such as YARN or Mesos, that requests resources from the Worker Nodes. From each Worker Node, Executors are assigned resources, which process computational tasks on one or more CPU threads. In addition to running as a truly distributed platform, Apache Spark can also utilize multi-threading on a single machine using the *standalone* mode which can be useful if the user has access to a large multi-core computer with sufficient memory.

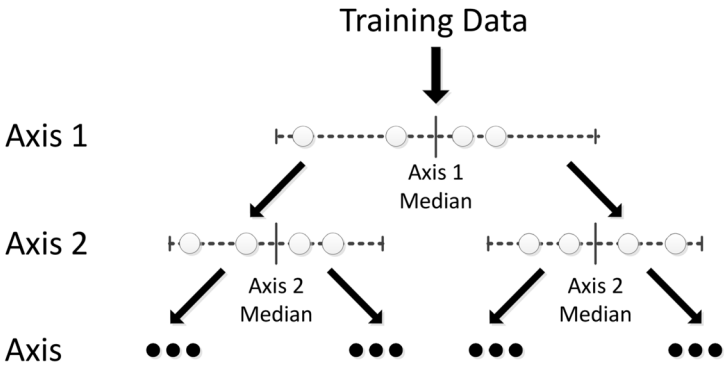


Fig. 3 KD-tree data structure. Data is partitioned at each branching based on the median value of the current axis

Apache Spark also supports the DataFrame data structure which provides popular SQL-style database optimization tools [27] and supports distributed MapReduce related operations like map, reduce, and filter. Code used for both KD and hybrid-spill trees [3] is based on DataFrames which will be abbreviated as *DF* in the following algorithms.

2.4 Imbalanced data

Learning from imbalanced data is one of the biggest challenges in modern machine learning [28], despite more than three decades of progress [29]. This is caused by a widespread appearance of skewed classes in various real-life domains. However, the difficulty lies not only in the uneven number of data examples among classes, but also in the example-level characteristics, such as class overlapping [30], borderline examples, small disjuncts, and label noise [29]. Most of works on imbalanced data focus on binary problems with well-defined minority and majority classes. Less attention has been paid to more complex cases of multi-class imbalanced data, where relationships among classes are much more difficult to analyze [31].

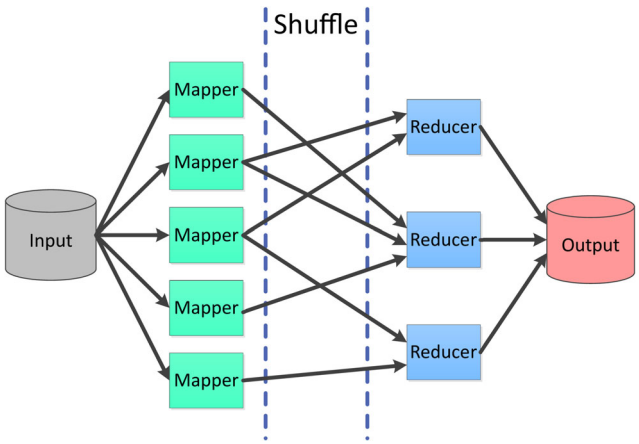


Fig. 4 The three main phases of the MapReduce model: map, shuffle, reduce

Imbalanced classification becomes even more challenging in the context of big data [29]. Here, we not only face vast volumes of data, but we also need to adapt existing algorithms to high-performance computing environments [32, 33], or propose novel solutions purely dedicated to large-scale datasets [34, 35].

Many standard classification algorithms fail when facing large-scale problems, even when implemented on dedicated architectures [36]. This can be explained by the fact that data partitioning among cluster nodes plays an important role in training local models and may actually corrupt the original class imbalance ratio, leading to over- or under-trained classifiers. Therefore, it is necessary to take this fact into account and embed node-based solutions into each base learner [37, 38]. Popular examples of successful usage of learning methods to cope with imbalanced big data include fuzzy rule-based [39] and associative classifiers [40], as well as Support Vector Machines [41].

Data preprocessing, most commonly in the form of sampling, is a highly popular branch of algorithms for alleviating

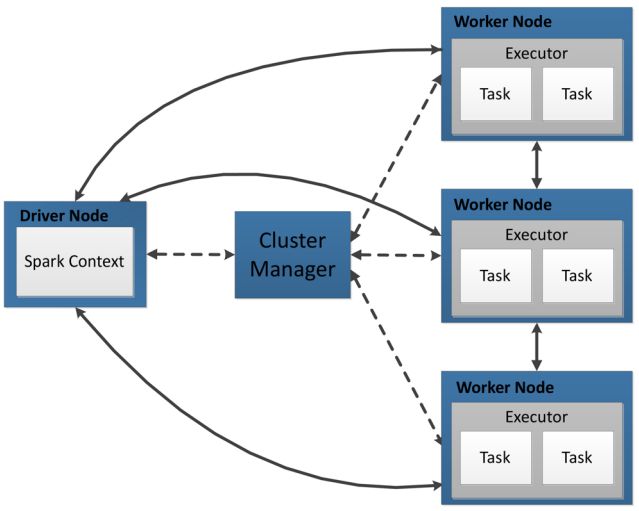


Fig. 5 The Spark cluster architecture for resource allocation and data transfer. Dashed lines show resource allocation and solid lines shows data transfer

class imbalance [42]. Despite their successes in small-scale datasets, their implementations on high-performance clusters is far from straightforward. Recent works on MapReduce-based oversampling [43] and undersampling [44], as well as their GPU-based counterparts [45] report various difficulties originating from local divisions of data examples. It was observed that despite their good performance on local nodes, they do not return a satisfactory global performance after the reduction phase. This shows that there is a need for novel sampling methods specifically dedicated to MapReduce environments. To the best of our knowledge, sampling for multi-class imbalanced problems in the big data context has not yet been discussed in the literature.

In the context of big data, imbalanced problems may be too difficult to be tackled by a single technique. A combination of algorithms, such as feature selection, data reduction, and sampling may be necessary to tackle such a task [46]. Alternatively, one may use ensembles trained at each node, combined with local class balancing, in order to improve the predictive power. Successful implementations include Random Forest [47] and ensembles of Neural Networks [48, 49].

3 Proposed algorithms

In this section, we present two different KD-tree based implementations for the Apache Spark framework: a classifier and a novel method for addressing class imbalance.

3.1 KD-tree classifier

While the KD-tree classifier has been implemented in several other languages and frameworks [50, 51], we are unaware of any publicly available implementations in Scala for Apache Spark. The popular *sklearn* Python package does include a KD-Tree implementation but as version 1.4.2 it only supports serial processing. Our KD-tree implementation is based on the existing Apache Spark hybrid-spill tree implementation [3] and since it fits the same machine learning pipelineing model the two methods are interchangeable. The top level metric-tree partitioning presented in the hybrd-spill tree was also used for the KD-tree to allow for a direct comparison.

The KD-tree classifier works as an approximate nearest neighbor algorithm by creating a KD-tree and performing DFS to find a leaf node. Examples present in a leaf node are tested to find the k-nearest neighbors. Like the spill-tree, a buffer region was used to help with the query examples near the median splits. A static buffer size of 25% of the node data size was chosen empirically, as it seemed to have a good balance between speed and performance while not introducing more complexity into this simple algorithm. Future work will be needed to determine if that value is universally appropriate or if there is another computationally inexpensive method for discovering an optimal value for a given dataset. Although training this KD-tree requires $\mathcal{O}(n \log^2 n)$ time compared to $\mathcal{O}(n \log n)$ for the hybrid-spill tree, its partitioning approach is independent to the number of dimensions. Like the hybrid-spill tree, querying for the nearest neighbor is also $\mathcal{O}(m \log n)$.

Algorithm 1 shows the pseudo code for the Apache Spark implementation for the KD-tree training phase. As with the hybrid-tree method, the query is performed by iterating through the trained tree. However, instead of a pivot point, the branch selection is based on the median value of the axis specified for that level in the tree. Once a leaf node is reached, all of the present examples are examined to find the k-nearest neighbors using the brute force approach; majority voting is performed to pick the predicted class. To improve the runtime of this algorithm, backtracking was not performed, but could be easily added if required.

Algorithm 1 Generate KD tree.

```
KD-TREE Data Structure:
  pivot: vector
  median: Double
  axis: Int
  radius: Double
  leftChild: Tree
  rightChild: Tree

procedure BUILD- KD- TREE(data: vector, leafSize: Int, axis: Int)
  if data.size == 0 then
    return Leaf(Empty)
  else if data.size <= leafSize then
    return Leaf(data)
  else
    sorted ← sortByAxis(data)
    medIdx ← sorted.size/2
    medExample ← sorted[medIdx]
    radius ← max(data.map(x → dist(x, data[medIdx]))
    left ← data[0, medIdx ∗ 1.25]
    right ← data[medIdx − medIdx ∗ 0.25, data.size]
    axis ← axis + 1
    return KD-TREE(medExample, medIdx, axis, radius,
      BUILD-KD-TREE(left, leafSize, axis),
      BUILD-KD-TREE(right, leafSize, axis))
  end if
end procedure
```

3.2 KD-tree based SMOTE

Challenges can arise when attempting to classify datasets that are class imbalanced. Classifiers tend to favor majority classes when presented with imbalanced data which can negatively impact performance, especially with the minority classes [29]. One solution is to balance the classes by oversampling the minority classes with methods such as random oversampling or SMOTE.

The SMOTE algorithm creates new examples for the minority classes to alleviate the class imbalance in the

dataset. For each class to be oversampled, five examples are randomly selected and the average of their feature values are used to create a new example. These synthetic examples are added to the dataset until the given minority class has the desired size. While these examples were not part of the original dataset, the idea is that they will occupy underrepresented regions of the true feature space which will then improve classifier accuracy. Algorithm 2 shows our Scala based Apache Spark implementation of SMOTE.

Algorithm 2 Generate SMOTE example DataFrame.

```
Ensure: Examples in DataFrame belong to the same class
  procedure SMOTE(DF: DataFrame, ClassLabel: Int)
    fvs = Array[5]
    for i = 1 to 5 do
      index ← Random.nextInt(DF.count)
      example ← DF[index]
      fvs[i] ← example.filter(featureVector)
    end for
    transposed ← fvs^T
    averages ← transposed.map(rowSum/5)
    smoteFeatureVector ← averages^T
    smoteExample ← {smoteFeatureVector, ClassLabel}
    return smoteExample
  end procedure
```

However, this approach may result in synthetic examples that do not well represent the true feature space for the given class. Figure 6 shows an example dataset with the classes represented by green triangles and blue circles. If the blue class needs to be oversampled with SMOTE, the traditional approach will sample from all possible examples in that class to create new synthetic examples. Shown in red crosses, we can see that the examples created by SMOTE often fall within the feature space of the wrong class and likely will negatively affect classification accuracy. This problem may be more pronounced if the level of class imbalance varies between clusters or if the feature space is not evenly represented.

Instead of having SMOTE sample from all possible class examples, we present a method that only uses examples that are close together. This can be achieved by clustering the training dataset using the leaves of a KD-tree. In our implementation, one KD-tree is created per class and each new SMOTE example is generated with five random examples from a single KD-tree leaf.

Algorithm 3's KD-CLUSTER-FIT method starts by filtering the dataset by class and calculates the size of the majority class. In our experiments, the minority classes were oversampled to match the size of the majority class. For each class, the CREATE-TREE method first determines which features to use for future tree splitting. Using these selected features, the BUILD-TREE method is started but returns an array of the resulting leaves instead of a complete KD-tree. Each of these leaves represent a cluster of approximate nearest neighbors.

Features that have less than three unique values will not be used. If a splitting feature contains only one unique value, there is no way to tell where to place the median splitting point and it would be unlikely that the partitioning would provide any information gain. While a median point could be easily found if two unique values exist for a splitting feature, the resulting branches could be highly imbalanced, which may negatively affect speed and accuracy.

After the features are selected, the specific feature to be used for each split can be adjusted. The SMOTE based class balancing did not work well in initial tests when features were used in the same order as presented in the training data. However, results improved when the feature with the smallest standard deviation at the current node was used. Future work will be required to determine what considerations are needed when addressing binary features and choosing the optimal feature at each split.

Algorithm 3 Fitting KD-tree model using SMOTE.

```
  procedure KD- CLUSTER- FIT(DF: DataFrame)
    labels ← DF.select("label").distinct()
    classDFs ← labels.map(x → DF.filter"label" == x))
    maxClassCount ← max(classDFs.map(x → x.size)))
    return labels.map(classDF → CREATE-TREE(classDF))
  end procedure

  procedure CREATE- TREE(DF: DataFrame, label: Int)
    for index from 0 until DF.numberOfFeatures do
      if DF[index].distinct() > 2 then
        axisToUse.append(index)
      end if
    end for
    return BUILD-TREE(DF, leafSize, axisToUse,
axisToUse[0])
  end procedure

  procedure BUILD- TREE(data: DataFrame, leafSize: Int, axis-
ToUse: Int, currentAxis: Int)
    if data.size == 0 then
      return EMPTY
    else if data.size <= leafSize then
      return data
    else
      nextAxis ← min(axisToUse.map(axis →
STD(data[axis])))
      sortedData ← data.sortedBy(nextAxis)
      leftData ← sortedData[0, sortedData.median]
      rightData ← sortedData[sortedData.median,
sortedData.size]
      leftResults ← BUILD-TREE(leftData,
       leafSize, axisToUse, nextAxis)
      rightResults ← BUILD-TREE(rightData,
       leafSize, axisToUse, nextAxis)
      return leftResult + rightResults
    end if
  end procedure
```

Once the training data has been clustered using the KD-tree leaves, oversampling can be performed with Algorithm 4. For each class, a corresponding leaf cluster is

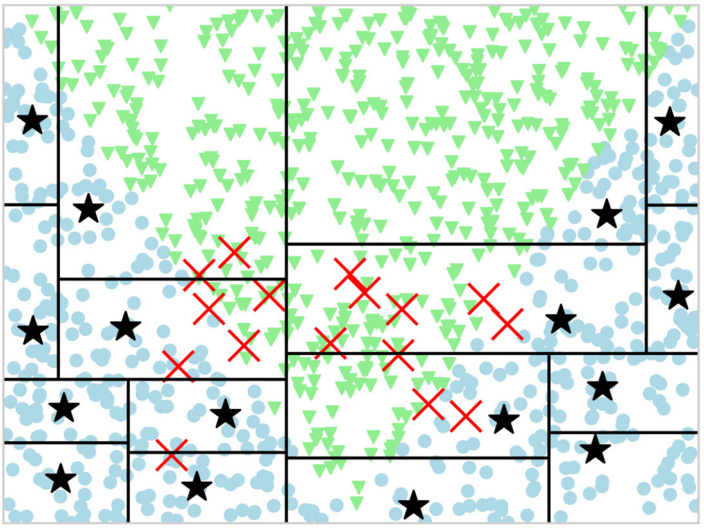


Fig. 6 Example dataset showing over sampling the blue circle class with SMOTE (red crosses) and with SMOTE based on KD-trees (black stars). The KD-tree partitioning of the blue circle class is overlaid with black lines

randomly chosen and five examples in that collection are used to generate a new SMOTE example. This process is continued until each class has as many examples as the majority class.

Algorithm 4 Oversampling with SMOTE.

```
procedure SMOTE- OVERSAMPLE(DF: DataFrame)
    labels ← DF.select("label").distinct()
    classDFs ← labels.map(x → DF.filter(label == x))
    maxClassCount ← max(classDFs.map(x → x.count))
    if usingStandardSMOTE then sampledClassDFs ←
classDFs.map(classDF →
        STANDARD-SMOTE-OVERSAMPLE
        (classDF, maxClassCount))
      balancedDF ← union(sampledClassDFs)
      return balancedDF
    else if usingKDTrees then
      classClusterArrays ← KD-CLUSTER-FIT(DF)
    end if
end procedure

procedure KD- CLUSTER- SMOTE- OVERSAMPLE(
    classDF: DataFrame, targetSampleCount: Int, trees: KDTree)
    samplesToAdd ← targetSampleCount - classDF.count()
sampledData ← trees.map(label → (0 to
samplesToAdd).map(x →
      SMOTE(x.randomCluster, label))
    return union(sampledData)
end procedure

procedure STANDARD- SMOTE- OVERSAMPLE(
    classDF: DataFrame, targetSampleCount: Int)
    if classDF.count() < targetSampleCount then
      samplesToAdd ← targetSampleCount −
classDF.count()
      newSamples ← (0 to samplesToAdd).map(x ←
        SMOTE(classDF, classDF.label)
      return union(classDF, newSamples)
    else
      return classDF
    end if
end procedure
```

3.3 Local versus distributed implementations

Most traditional machine learning and graph algorithms were designed for serial processing making their execution straightforward. Although adding multi-threading capabilities increases complexity, these algorithms often have sections that are trivial to parallelize, such as *for* loops that iterate over all training examples. As Algorithm 1 shows, most of the computational work is attributed to sorting which can easily benefit from parallelism. However, running these algorithms on a single computer will not be possible if the dataset becomes too large, therefore requiring a distributed solution.

While adding at least some parallelism with multi-threading may be trivial, upgrading serial algorithms to run efficiently on a distributed environment requires more planning from the start. Apache Spark gains parallelism by distributing DataFrames or their underlying resilient distributed dataset (RDD) data structure to computational nodes. This is done by performing MapReduce operations like map, filter and reduce on DataFrames and letting the Apache Spark framework to manage task scheduling in the background. Because most of the traditional algorithms do not follow these design patters, they usually have to be completely written from scratch or at least be based on sub-components that already have Apache Spark support. Cluster management and the extra network traffic does introduce some overhead, but it comes with an ease of use, fault tolerance through data replication and the ability to utilize a heterogeneous cluster environment.

3.4 Spark configurations

One of the main challenges with using Apache Spark is how to properly configure the cluster [52–54], something we also experienced while performing the experiments in

Section 4.3. There are currently over 100 parameters that can be adjusted, including the number of executors, amount of memory per executor, the shuffling behavior and when the base JVM garbage collector should run. Although many of these parameters can be safely kept at the default settings, those dealing with memory can be critical to efficiently or even successfully run jobs. For example, the cluster driver and executors have their own memory settings which can differ significantly, especially if they run on a heterogeneous cluster. Allocating too much memory can reduce the number of executors that can run on a node, while not allocating enough memory can cause out-of-memory errors. Choosing the optimal cluster parameters can be difficult and likely task specific, but finding the right combination can significantly improve runtime performance [55, 56].

4 Experimental study

This experimental study was designed to answer the following research questions (RQs):

- **RQ1:** Does the proposed KD-tree implementation outperform the existing state-of-the-art Hybrid-Spill Tree in both predictive power and computational efficiency?
- **RQ2:** Is the proposed SMOTE variant for the MapReduce platform capable of efficiently combating class imbalance, while avoiding the MapReduce pitfall of local data partitioning?
- **RQ3:** Is our method flexible enough to work with both binary and multi-class imbalanced problems, without a need for any changes in its structure?

4.1 Performance metrics

The average accuracy (AvAcc) and class balance accuracy (CBA) metrics were used for the model accuracy metrics because most of the presented datasets are multi-class and imbalanced. As shown in Formula (1), the basic accuracy only considers the global accuracy and may be biased towards to the majority classes. Both AvAcc and CBA take in account performance per class and introduce a higher penalty for predicting the minority class examples incorrectly.

$$\begin{aligned} accuracy &= \frac{tp + tn}{tp + tn + fp + fn} \\ AvAcc &= \sum_{i=1}^{C} \frac{tp_i + tn_i}{tp_i + tn_i + fp_i + fn_i} \\ CBA &= \sum_{i=1}^{C} \frac{mat_{i,i}}{\max(\sum_{j=1}^{C} mat_{i,j}, \sum_{j=1}^{C} mat_{j,i})} \end{aligned} \tag{1}$$

4.2 Datasets

To evaluate these algorithms, we have chosen seven datasets [57–62] with varying properties. As shown in Table 1, there is a wide variation in the number of classes and features for each dataset allowing us to see how the proposed methods behave when presented with these combinations. Also shown is the imbalance ratio of the largest class size over the smallest.

4.3 Experiments on AWS

To perform our experiments, we have chosen the Amazon Web Service (AWS) platform with Elastic MapReduce (EMR). The EMR platform allows for easy deployment of cluster based applications, such as Hadoop and Spark, in a distributed environment. All experiments were performed using 1, 2, 4, and 6 c5.2xlarge instances for the computational work with each c5.2xlarge virtual instance providing 8 vCores, or threads, and 16 GB of RAM. For both classification and KD-tree based SMOTE, we have allocated 10 GB of memory to each executor and used one executor per instance.

Table 1 Number of features, classes and maximum class balance ratio for each test dataset

Dataset	Instances	Features	Classes	Imb. Ratio
Cover type	581,012	54	7	104.4
Traffic violations	1,378,663	26	7	10.5
SEER	2,532,629	11	10	5.7
Intel sensors	2,219,803	5	58	31.2
IoT	3,000,000	115	11	4.9
SUSY	5,000,000	18	2	1.2
HIGGS	11,000,000	28	2	1.1

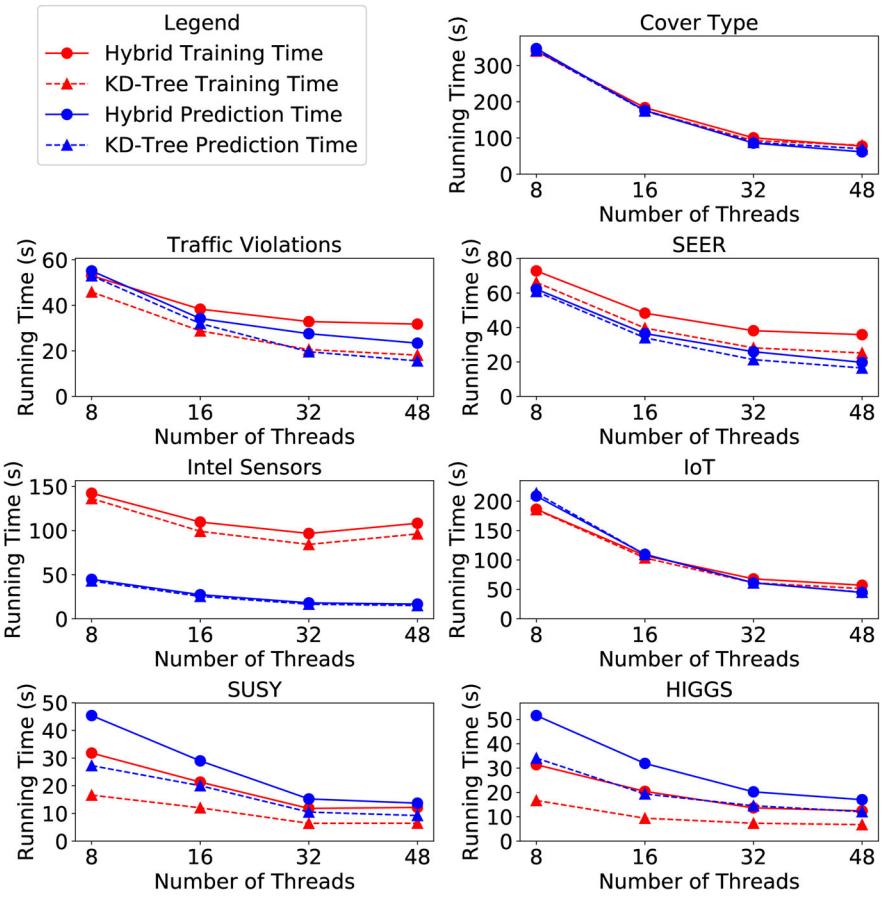


Fig. 7 Classification running times for leaf size 10

Our first experiments compare the runtime performance and classifier accuracy of the proposed KD-tree and the existing hybrid-spill tree classifiers. For each dataset, both algorithms are run with a combination of leaf sizes (10, 100, 500, 1000, 2500) and number of computational threads (8, 16, 32, 48). The second round of experiments compares the standard SMOTE algorithm against the proposed KD-tree based SMOTE modified algorithm. These experiments were run on all datasets with same number of threads as with the classifier experiments.

5 Results

This section presents the results of the KD-tree classifier and SMOTE experiments.

5.1 Classifiers

For the classifier experiments, we compared the running time and model accuracy between the existing Hybrid-Spill tree against our proposed KD-tree implementation. Two significant parameters for both algorithms are the leaf size of the tree and the k value for the number of examples to use for class prediction. Figures 7, 8 and 9 show running times for various leaf sizes against the number of threads used and the value k=5 was used for all experiments.

In almost all presented cases, the KD-tree was faster than the Hybrid-Spill tree for both training and prediction phases. Thread level scaling was also shown to be dependent on the leaf size for both algorithms. Figure 7 shows that adding more threads consistently improves running times but this effect is diminished as the leaf size increases. Using a leaf size of 500 still shows some scaling but the improvements do not continue beyond 16 or 32 threads and performance starts to degrade when the leaf size is 2500.

Increasing the leaf size improves training times for both algorithms, but the KD-tree can be almost two times faster than the Hybrid-Spill tree for small leaf sizes. However, this performance difference decreases as the leaf size increases and at leaf size 2500 the times are almost identical. The running time between the two methods is much more noticeable for making predictions. As the leaf size increases, the gap between the KD-tree and Hybrid-Spill tree performance

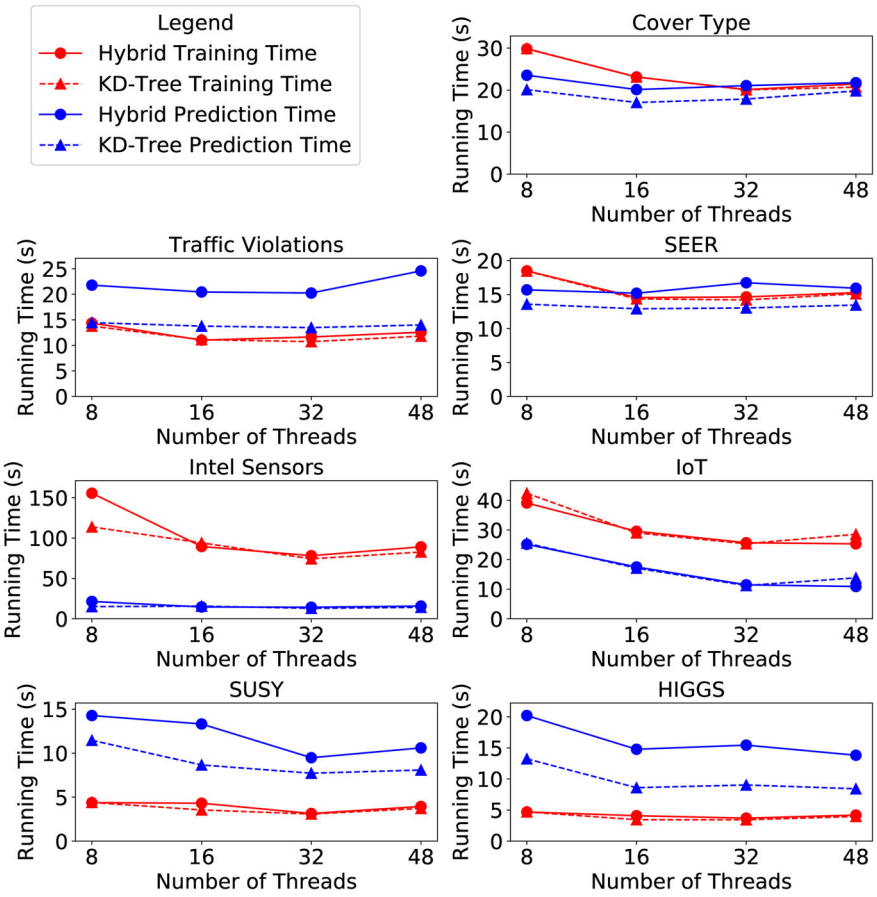


Fig. 8 Classification running times for leaf size 500

widens with many cases where the KD-tree is two to three times faster (**RQ1** answered). This is apparent in Fig. 10, where the KD-tree performance is mostly flat as leaf size increases but the Hybrid-Spill tree starts slowing down.

The Hybrid tree tends to be more accurate than the KD-tree for both metrics, but the AvAcc differences are minimal. However, the KD-tree performs best for two datasets including Covtype, where its CBA result is two times better than that with the Hybrid-Spill tree (**RQ1** answered) Fig. 11.

The Hybrid-Spill tree CBA results were not largely affected by increasing leaf sizes, but this generally had a negative impact on the KD-tree classifier, although this is opposite for the Covtype dataset. Additionally, the AvAcc and CBA metrics were not significantly influenced by the thread count for either the KD-tree or the Hybrid-Spill tree.

5.2 SMOTE

To test the impact of KD-trees on SMOTE, the Spark ML Random Forest classifier was used with 5-fold cross validation and the KD-tree leaf size was set to 64 as it seemed to work well across all datasets. We observed that the class prediction accuracy largely varies depending on the individual dataset. Figure 12 shows the accuracy ratios between standard SMOTE and SMOTE with KD-Tree example selection. Using standard SMOTE as a baseline, we can see that the tree method is best in five of the seven cases. The new method does worse on the SUSY dataset, but this only has a relative difference of 3%. The best result was with the Sensors dataset, where the KD-tree method had a 12% percent improvement on CBA accuracy. The average improvement across all seven datasets was 0.14% for AvAcc and 3.4% for CBA. While the KD-tree based method had little influence on AvAcc compared to standard SMOTE, it had a much larger effect on the CBA results. This may be because the more discerning KD-tree based oversampling method mostly benefits small and difficult to classify classes. Another observation is that the KD-tree clustering method shows the largest improvements on datasets with high class imbalance which may be the main strength of this approach (**RQ2** answered) (Fig. 13).

Tables 2 and 3 show the AvAcc and CBA values for the number of threads used. As shown in Table 2, there was very little difference for the AvAcc metric relative to the number of

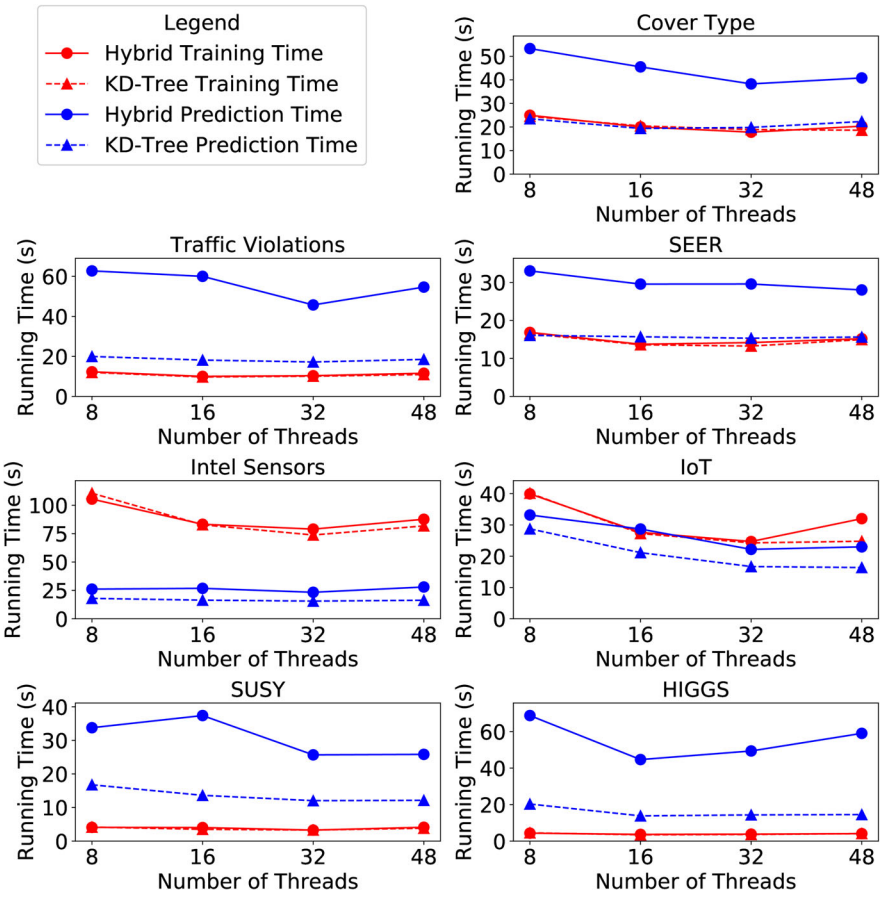


Fig. 9 Classification running times for leaf size 2500

threads used with all maximum and relative differences under 0.5%. However, there was a larger range of CBA values for all datasets with the largest relative difference being almost 8% for the Intel Sensors dataset. This dataset has a high number of classes, 58, which may be a contributing factor to this result. The changes in the minority class accuracy can have a significant effect on the CBA metric such that any perturbations in the class balancing may be more pronounced (**RQ3** answered).

5.3 Speed-up by total threads

In Table 4, we show a comparison of runtime speed-up between the two tree algorithms based on the number of threads used for both the training and predicting tasks. These results were from the experiments using a leaf size of 10 as it would be the most computational intensive and should better highlight the potential of runtime improvements. The speed-up rate shown was calculated by dividing the runtime of using eight threads by the time taken using the additional threads. Compared to using only eight, the increasing number of threads results in hardware ratios of 2, 4 and 6. This provides the theoretical runtime improvement for those three cluster configurations, although in practice results will be lower because of overhead and underutilized processors.

In these experiments, the KD-tree had a higher speed-up based on cluster size with four of the seven datasets. Both algorithms showed a higher rate of improvement with the classifier prediction task compared to the training. Although speed-ups were consistently demonstrated with additional hardware, the rate of improvement varied significantly between different datasets. The Cover Type dataset showed maximum gains of over $4x$, approaching the theoretical increase of $6x$, but best result for training with the Intel Sensor dataset was only $1.6x$. Although the speed-ups were often similar between these two algorithm, the KD-tree already had a lower running time as highlighted in Figs. 7-10.

5.4 Scale-up by tree depth

Similar to speed-up, we also assessed the performance increase based on scaling up the computational workload. Table 5 shows the runtime improvement ratio going from

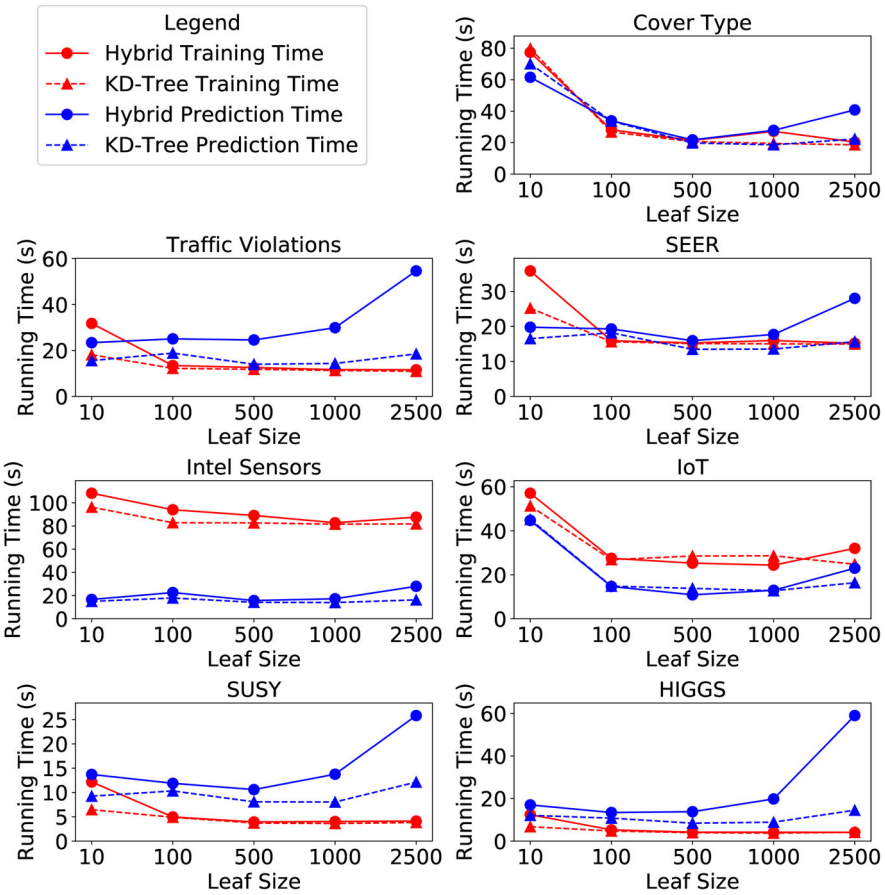


Fig. 10 Classification running times with 48 threads for each leaf size

eight to 48 threads with different leaf sizes. As the leaf size decreases, more child trees are needed making the overall tree deeper and requiring more computation. This additional work would likely result in better processor utilization and therefore potential runtime performance. These results show that cluster efficiency improves as the size of the task increases, with one exception of the Intel Sensors dataset.

6 Conclusions and future works

We have presented a new KD-tree classifier and a novel class balancing method, both implemented in Scala for the Apache Spark framework. Our comparisons between KD and hybrid-spill tree algorithms on the AWS distributed computing platform has given some insight on the strengths and weaknesses of these two methods. In this paper, we also demonstrated that the quality of oversampling with SMOTE can be improved using the leaves of trained KD-trees. These implementations are available on Github at https://github.com/fsleeman/spark-knn.

While the hybrid-spill classifier often provided the highest accuracy, it was slower than the KD-tree implementation. For two datasets, the KD-tree had the best classification accuracy showing that in some cases the KD-tree is both faster and more accurate. Our experiments showed that the leaf size played a significant role in running time of both methods, but was more significant for the hybrid-spill tree as the leaf sizes increased. Large leaves mean that the resulting tree will not be as tall, so more work will be required to perform nearest neighbor on the leaf examples. The hybrid-spill tree is at risk of getting penalized multiple times when backtracking on metric-trees, something that the KD-tree avoids. The number of threads did not have a significant effect on classification accuracy, but runtime performance increased until approximately 16 to 32 threads and this trend was exhibited for both algorithms.

For class balancing, our KD-tree clustering method showed classifier improvements for five out of seven datasets. The cluster based oversampling method showed the biggest difference when applied to the most class imbalanced datasets, highlighting a potential application for this method. Standard SMOTE oversampling ran faster except for the

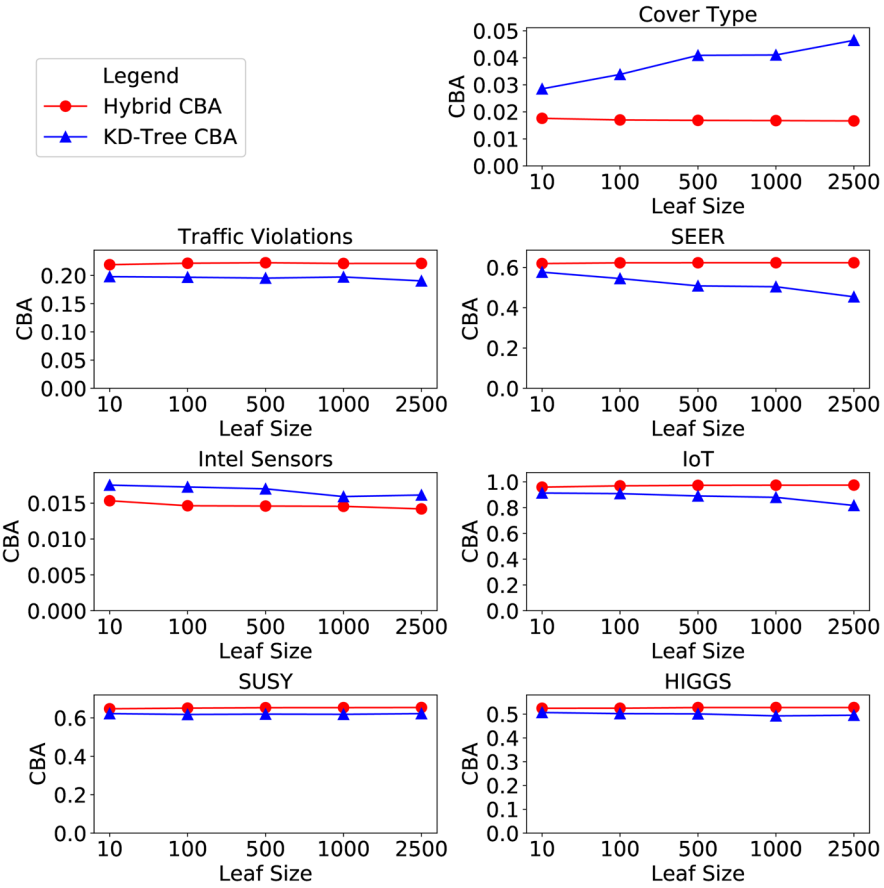


Fig. 11 CBA results for each leaf size using 48 threads

sensors dataset which has many classes, illustrating another potential use for the KD-tree clustering method. Like with the classification experiments, runtime performance increased until 16 to 32 threads. The datasets used in these experiments were large, but the size of the generated KD-trees were proportional to the number of examples in each class, significantly smaller than the original datasets.

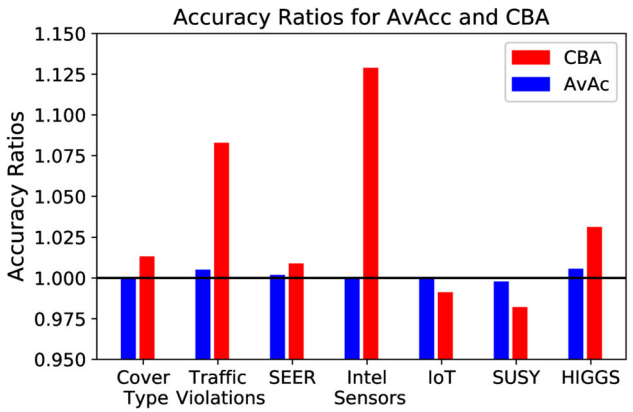


Fig. 12 Accuracy results between standard SMOTE and SMOTE with KD-trees

Apache Spark's primary data structure is the tabular DataFrame and the presented KD-tree algorithm should work on any data that can fit that format. One limitation of this and other traditional tree algorithms is that they were not designed to handle non-vector based data. These algorithms expect that each example will be a 1-dimensional vector of features rather than a multi-dimensional matrix or other more complicated structures. These tree algorithms can still use image data directly if flattened, but this approach will likely reduce predictive performance as some spacial information will be lost. Future work should include the design and implementation of tree algorithms that inherently support spatial data formats and their corresponding distance metrics.

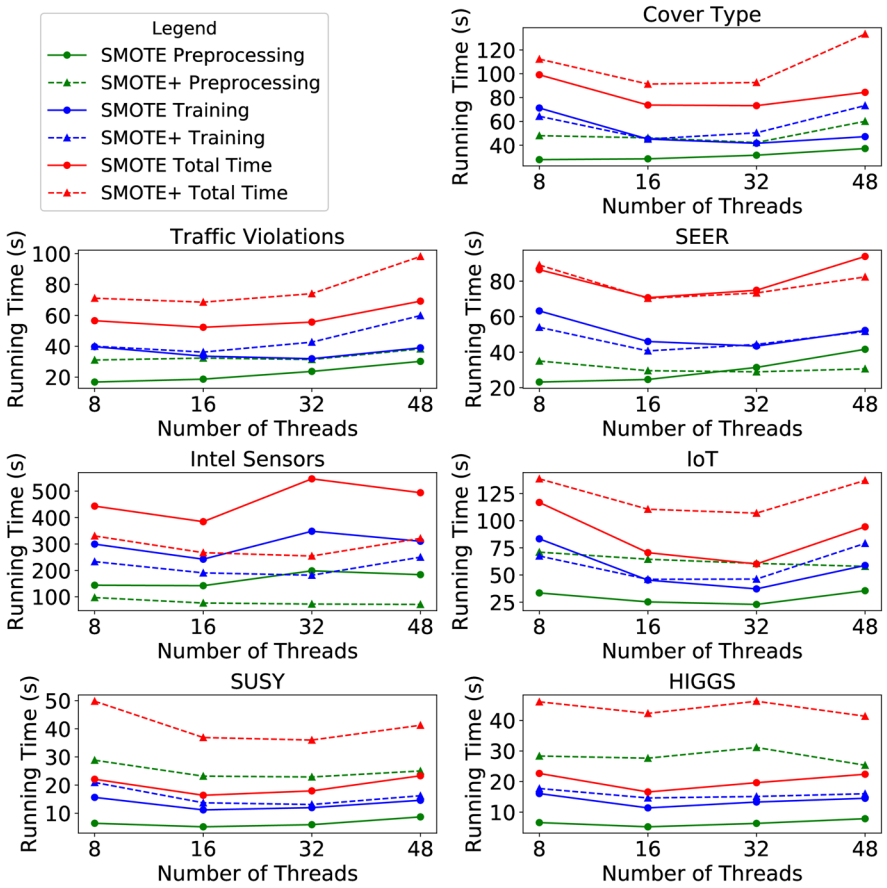


Fig. 13 Running times for SMOTE and SMOTE with KD-trees

Table 2 AvAcc accuracy ratio between SMOTE and SMOTE with KD-trees

Dataset	Total threads				Max relative difference (%)	Max absolute difference
	8	16	32	48		
Cover type	64.35	64.36	64.34	64.36	0.03	0.02
Traffic violations	81.08	81.20	81.08	81.16	0.15	0.12
SEER	93.32	93.37	93.33	93.38	0.06	0.06
Intel sensors	94.90	94.91	94.90	94.90	0.01	0.01
IoT	99.61	99.54	99.52	99.39	0.22	0.22
SUSY	77.60	77.56	77.66	77.59	0.13	0.10
HIGGS	66.87	67.16	67.00	67.16	0.43	0.29

Table 3 CBA accuracy ratio between SMOTE and SMOTE with KD-trees

Dataset	Total threads				Max relative difference (%)	Max absolute difference
	8	16	32	48		
Cover type	4.67	4.69	4.78	4.61	3.69	0.17
Traffic violations	21.72	22.13	21.65	21.64	2.26	0.49
SEER	44.80	45.20	45.06	44.77	0.96	0.43
Intel sensors	0.94	0.92	0.88	0.95	7.95	0.07
IoT	96.61	96.25	95.99	95.10	1.59	1.51
SUSY	73.46	73.48	73.48	73.38	0.14	0.10
HIGGS	64.44	65.28	64.99	65.85	2.19	1.41

Table 4 How many times faster the tree algorithms run using additional threads

Dataset	Algorithm	Training total threads			Predicting total threads		
		16 (2x)	32 (4x)	48 (6x)	16 (2x)	32 (4x)	48 (6x)
Cover type	Hybrid	1.85	3.38	**4.39**	**1.97**	**4.04**	**5.63**
	KD-tree	**1.93**	**3.67**	4.26	1.96	3.89	4.88
Traffic violations	Hybrid	1.38	1.61	1.66	1.61	2.00	2.35
	KD-tree	**1.59**	**2.22**	**2.52**	**1.65**	**2.69**	**3.39**
SEER	Hybrid	1.50	1.90	2.03	1.70	2.39	3.15
	KD-tree	**1.66**	**2.34**	**2.62**	**1.78**	**2.85**	**3.69**
Intel sensors	Hybrid	1.29	1.47	1.31	1.63	2.47	2.68
	KD-tree	**1.37**	**1.61**	**1.41**	**1.69**	**2.59**	**2.85**
IoT	Hybrid	1.73	2.74	3.26	1.90	3.41	4.66
	KD-tree	**1.79**	**3.03**	**3.61**	**1.95**	**3.49**	**4.72**
SUSY	Hybrid	**1.49**	**2.69**	**2.61**	**1.56**	**2.97**	**3.30**
	KD-tree	1.38	2.58	2.57	1.36	2.60	2.95
HIGGS	Hybrid	1.53	**2.29**	**2.52**	1.61	**2.54**	**3.03**
	KD-tree	**1.77**	2.27	2.46	**1.76**	2.34	2.82

All values are relative to using eight threads
The bold emphasis was added to highlight the result from the best performing algorithm (KD-tree or Hybrid) for each dataset/leaf count combindation

Table 5 Runtime improvements by going from 8 to 48 threads based on an increasing workload, represented by the decreasing leaf count

Dataset		Leaf Count				
		2500	1000	500	100	10
Cover type	Hybrid	1.22	0.98	1.38	1.83	**4.39**
	KD-tree	**1.31**	**1.41**	**1.44**	**1.87**	4.26
Traffic violations	Hybrid	1.06	1.10	1.14	1.16	1.66
	KD-tree	**1.09**	**1.11**	**1.16**	**1.27**	**2.52**
SEER	Hybrid	**1.11**	1.08	1.20	**1.35**	2.03
	KD-tree	1.10	**1.15**	**1.22**	1.33	**2.62**
Intel sensors	Hybrid	1.20	**1.61**	**1.74**	1.15	1.31
	KD-tree	**1.35**	1.34	1.37	**1.34**	**1.41**
IoT	Hybrid	1.24	**1.60**	**1.54**	1.73	3.26
	KD-tree	**1.61**	1.37	1.48	**2.01**	**3.61**
SUSY	Hybrid	0.99	1.07	1.11	**1.19**	**2.61**
	KD-tree	**1.08**	**1.15**	**1.18**	1.16	2.57
HIGGS	Hybrid	1.06	1.03	1.11	1.18	**2.52**
	KD-tree	**1.09**	**1.12**	**1.18**	**1.25**	2.46

The bold emphasis was added to highlight the result from the best performing algorithm (KD-tree or Hybrid) for each dataset/leaf count combindation

Author Contributions William C. Sleeman IV: Conceptualization, Software, Writing - original draft, Writing - review & editing; Martha Roseberry: Conceptualization, Writing - original draft, Writing - review & editing; Preetam Ghosh: Conceptualization, Supervision. Writing - review & editing; Alberto Cano: Funding acquisition, Supervision, Writing - review & editing; Bartosz Krawczyk: Supervision, Writing - review & editing

Funding This research was partially supported by the Amazon AWS Machine Learning Research award.

Data Availability Statement The datasets used in this work are publicly available and referenced in the bibliography section.

Code Availability Statement https://github.com/fsleeman/spark-knn

References

1. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ (2016) Apache Spark: a unified engine for big data processing. Commun ACM 59(11):56–65. https://doi.org/10.1145/2934664
2. Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) SMOTE: synthetic minority over-sampling technique. J Artif Intell Res 16:321–357. https://doi.org/10.1613/jair.953
3. Fang F (2018) spark-knn. https://github.com/saurfang/spark-knn. Accessed: 12-14-2018
4. Su Z, Hu Q, Denoeux T (2020) A distributed rough evidential k-nn classifier: integrating feature reduction and classification. IEEE Trans Fuzzy Syst 29(8):2322–2335. https://doi.org/10.1109/TFUZZ.2020.2998502
5. Sun L, Zhang J, Ding W, Xu J (2022) Feature reduction for imbalanced data classification using similarity-based feature clustering with adaptive weighted k-nearest neighbors. Inf Sci 593:591–613. https://doi.org/10.1016/j.ins.2022.02.004
6. Taunk K, De S, Verma S, Swetapadma A (2019) A brief review of nearest neighbor algorithm for learning and classification. In: 2019 International Conference on Intelligent Computing and Control Systems (ICCS), pp 1255–1260. https://doi.org/10.1109/ICCS45141.2019.9065747. IEEE
7. Cunningham P, Delany SJ (2021) k-nearest neighbour classifiers-a tutorial. ACM Comput Surv (CSUR) 54(6):1–25. https://doi.org/10.1145/3459665
8. Jo J, Seo J, Fekete J (2018) Panene: a progressive algorithm for indexing and querying approximate k-nearest neighbors. IEEE IEEE Trans Vis Comput Graph 26(2):1347–1360. https://doi.org/10.1109/TVCG.2018.2869149
9. Li W, Zhang Y, Sun Y, Wang W, Li M, Zhang W, Lin X (2019) Approximate nearest neighbor search on high dimensional data–experiments, analyses, and improvement. IEEE Trans Knowl Data Eng 32(8):1475–1488. https://doi.org/10.1109/TKDE.2019.2909204
10. Gowanlock M (2021) Hybrid knn-join: parallel nearest neighbor searches exploiting cpu and gpu architectural features. J Parallel Distrib Comput 149:119–137. https://doi.org/10.1016/j.jpdc.2020.11.004
11. Skryjomski P, Krawczyk B, Cano A (2019) Speeding up k-nearest neighbors classifier for large-scale multi-label learning on GPUs. Neurocomputing 354:10–19. https://doi.org/10.1016/j.neucom.2018.06.095
12. Mir A, Nasiri JA (2018) Knn-based least squares twin support vector machine for pattern classification. Appl Intell 48(12):4551–4564. https://doi.org/10.1007/s10489-018-1225-z
13. Xie F, Xu Y (2019) An efficient regularized k-nearest neighbor structural twin support vector machine. Appl Intell 49:4258–4275. https://doi.org/10.1007/s10489-019-01505-5
14. Pan X, Luo Y, Xu Y (2015) K-nearest neighbor based structural twin support vector machine. Knowl-Based Syst 88:34–44. https://doi.org/10.1016/j.knosys.2015.08.009
15. Uhlmann JK (1991) Satisfying general proximity/similarity queries with metric trees. Inf Process Lett 40(4):175–179. https://doi.org/10.1016/0020-0190(91)90074-R
16. Liu T, Moore A, Yang K, Gray A (2004) An investigation of practical approximate nearest neighbor algorithms. Adv Neural Inf Process Syst 17. https://doi.org/10.5555/2976040.2976144
17. Clarkson KL (2006) Nearest-neighbor searching and metric space dimensions. Nearest-neighbor methods for learning and vision: theory and practice 15–59
18. Bentley JL (1975) Multidimensional binary search trees used for associative searching. Commun ACM 18(9):509–517. https://doi.org/10.1145/361002.361007
19. Gionis A, Indyk P, Motwani R (1999) Similarity search in high dimensions via hashing. In: Proc 25th VLDB Conf, pp 518–529. https://doi.org/10.5555/645925.671516
20. Kanj S, Brüls T, Gazut S (2018) Shared nearest neighbor clustering in a locality sensitive hashing framework. J Comput Biol 25(2):236–250. https://doi.org/10.1089/cmb.2017.0113
21. Ren X, Zheng X, Cui L, Wang G, Zhou H (2022) Asymmetric similarity-preserving discrete hashing for image retrieval. Appl Intell 1–18. https://doi.org/10.1007/s10489-022-04167-y
22. Liu T, Rosenberg C, Rowley HA (2007) Clustering billions of images with large scale nearest neighbor search. In: IEEE Workshop on Applications of Computer Vision (WACV'07), pp 28–28. https://doi.org/10.1109/WACV.2007.18
23. Maillo J, Ramírez S, Triguero I, Herrera F (2017) kNN-IS: an iterative Spark-based design of the k-nearest neighbors classifier for big data. Knowl-Based Syst 117:3–15. https://doi.org/10.1016/j.knosys.2016.06.012
24. Ramírez-Gallego S, Krawczyk B, García S, Woźniak M, Benítez JM, Herrera F (2017) Nearest neighbor classification for high-speed big data streams using Spark. IEEE Transactions on Systems, Man, and Cybernetics: Syst 47(10):2727–2739. https://doi.org/10.1109/TSMC.2017.2700889
25. Gonzalez-Lopez J, Ventura S, Cano A (2018) Distributed nearest neighbor classification for large-scale multi-label data on Spark. Futur Gener Comput Syst 87:66–82. https://doi.org/10.1016/j.future.2018.04.094
26. Ramírez-Gallego S, Fernández A, García S, Chen M, Herrera F (2018) Big data: tutorial and guidelines on information and process fusion for analytics algorithms with mapreduce. Inf Fusion 42:51–61
27. Villarroya S, Baumann P (2022) A survey on machine learning in array databases. Appl Intell 1–24. https://doi.org/10.1007/s10489-022-03979-2
28. Krawczyk B (2016) Learning from imbalanced data: open challenges and future directions. Progress in Artificial Intelligence 5(4):221–232. https://doi.org/10.1007/s13748-016-0094-0
29. Fernández A, García S, Galar M, Prati RC, Krawczyk B, Herrera F, Fernández A, García S, Galar M, Prati RC et al (2018) Foundations on imbalanced classification. Learning from Imbalanced Data Sets 19–46. https://doi.org/10.1007/978-3-319-98074-4_2
30. Sáez JA, Galar M, Krawczyk B (2019) Addressing the overlapping data problem in classification using the one-vs-one decomposition strategy. IEEE Access 7:83396–83411. https://doi.org/10.1109/ACCESS.2019.2925300

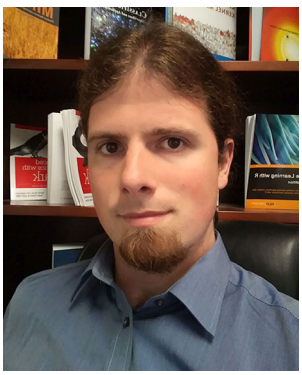
31. Lango M, Stefanowski J (2022) What makes multi-class imbalanced problems difficult? an experimental study. Expert Syst Appl 199:116962. https://doi.org/10.1016/j.eswa.2022.116962
32. Cano A (2018) A survey on graphic processing unit computing for large-scale data mining. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery **8**(1). https://doi.org/10.1002/widm.1232
33. Cano A, Krawczyk B (2019) Evolving rule-based classifiers with genetic programming on GPUs for drifting data streams. Patt Recog 87:248–268. https://doi.org/10.1016/j.patcog.2018.10.024
34. Hasanin T, Khoshgoftaar TM, Leevy JL, Bauder RA (2019) Severely imbalanced big data challenges: investigating data sampling approaches. J Big Data 6(1):1–25. https://doi.org/10.1186/s40537-019-0274-4
35. Sleeman WC IV, Krawczyk B (2021) Multi-class imbalanced big data classification on spark. Knowl-Based Syst 212:106598. https://doi.org/10.1016/j.knosys.2020.106598
36. Hasanin T, Khoshgoftaar TM, Leevy JL, Seliya N (2019) Examining characteristics of predictive models with imbalanced big data. J Big Data 6:69. https://doi.org/10.1186/s40537-019-0231-2
37. Abdel-Hamid NB, El-Ghamrawy SM, El-Desouky AI, Arafat H (2018) A dynamic Spark-based classification framework for imbalanced big data. J Grid Comput 16(4):607–626. https://doi.org/10.1007/s10723-018-9465-z
38. Hassib EM, El-Desouky AI, El-Kenawy EM, El-Ghamrawy SM (2019) An imbalanced big data mining framework for improving optimization algorithms performance. IEEE Access 7:170774–170795. https://doi.org/10.1109/ACCESS.2019.2955983
39. Fernández A, Almansa E, Herrera F (2017) Chi-Spark-RS: an Spark-built evolutionary fuzzy rule selection algorithm in imbalanced classification for big data problems. In: IEEE Int Conf Fuzzy Syst, pp 1–6. https://doi.org/10.1109/FUZZ-IEEE.2017.8015520
40. Almasi M, Abadeh MS (2018) A new MapReduce associative classifier based on a new storage format for large-scale imbalanced data. Clust Comput 21(4):1821–1847. https://doi.org/10.1007/s10586-018-2812-9
41. Fernández A, del Río S, Chawla NV, Herrera F (2017) An insight into imbalanced big data classification: outcomes and challenges. Complex Intell Syst 3:105–120. https://doi.org/10.1007/s40747-017-0037-9
42. Chen H, Shen Y (2017) Reducing imbalance ratio in MapReduce. In: IEEE International Symposium on Cloud and Service Computing, pp 279–282. https://doi.org/10.1109/SC2.2017.54
43. Basgall MJ, Hasperué W, Naiouf MR, Fernández A, Herrera F (2019) An analysis of local and global solutions to address big data imbalanced classification: a case study with SMOTE preprocessing. In: Conference on cloud computing and big data, pp 75–85. https://doi.org/10.1007/978-3-030-27713-0_7
44. Triguero I, Galar M, Bustince H, Herrera F (2017) A first attempt on global evolutionary undersampling for imbalanced big data. In: IEEE Congress on Evolutionary Computation (CEC) pp 2054–2061.https://doi.org/10.1109/CEC.2017.7969553
45. Gutiérrez PD, Lastra M, Benítez JM, Herrera F (2017) SMOTE-GPU: big data preprocessing on commodity hardware for imbalanced classification. Prog Artif Intell 6(4):347–354. https://doi.org/10.1007/s13748-017-0128-2
46. Triguero I, del Río S, López V, Bacardit J, Benítez JM, Herrera F (2015) ROSEFW-RF: the winner algorithm for the ECBDL'14 big data competition: an extremely imbalanced big data bioinformatics problem. Knowl-Based Syst 87:69–79. https://doi.org/10.1016/j.knosys.2015.05.027
47. del Río S, López V, Benítez JM, Herrera F (2014) On the use of MapReduce for imbalanced big data using random forest. Inf Sci 285:112–137. https://doi.org/10.1016/j.ins.2014.03.043
48. Zhai J, Zhang S, Wang C (2017) The classification of imbalanced large data sets based on MapReduce and ensemble of ELM classifiers. Int J Mach Learn Cybern 8(3):1009–1017. https://doi.org/10.1007/s13042-015-0478-7
49. Zhai J, Zhang S, Zhang M, Liu X (2018) Fuzzy integral-based ELM ensemble for imbalanced big data classification. Soft Comput 22(11):3519–3531. https://doi.org/10.1007/s00500-018-3085-1
50. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: machine learning in Python. J Mach Learn Res 12:2825–2830. https://doi.org/10.5555/1953048.2078195
51. Wehr D, Radkowski R (2018) Parallel kd-tree construction on the gpu with an adaptive split and sort strategy. Int J Parallel Program 46:1139–1156. https://doi.org/10.1007/s10766-018-0571-0
52. Ahmed N, Barczak AL, Rashid MA, Susnjak T (2021) An enhanced parallelisation model for performance prediction of apache spark on a multinode hadoop cluster. Big Data and Cognitive Computing 5(4):65
53. Aziz K, Zaidouni D, Bellafkih M (2019) Leveraging resource management for efficient performance of apache spark. J Big Data 6(1):78
54. Chicco D, Ferraro Petrillo U, Cattaneo G (2023) Ten quick tips for bioinformatics analyses using an apache spark distributed computing environment. PLoS Comput Biol 19(7):1011272
55. Minukhin S, Brynza N, Sitnikov D (2020) Analyzing performance of apache spark mllib with multinode clusters on azure hdinsight: spark-perf case study. In: International scientific conference "intellectual systems of decision making and problem of computational intelligence", pp 114–134. Springer
56. Singh T, Gupta S, Satakshi Kumar M (2023) Adaptive load balancing in cluster computing environment. J Supercomput 79(17):20179–20207
57. (1998) Remote Sensing and GIS Program, Colorado State University: Covertype data set. Retrieved from: https://archive.ics.uci.edu/ml/datasets/Covertype
58. (2018) Montgomery County of Maryland: Traffic violations. Retrieved from: https://catalog.data.gov/dataset/traffic-violations-56dda (2018)
59. (2018) Surveillance, Epidemiology, and End Results (SEER) Program (www.seer.cancer.gov) Research Data (1975-2016), National Cancer Institute, DCCPS, Surveillance Research Program, released April 2019, based on the November 2018 submission
60. (2004) Intel Berkeley Research Lab: Intel lab data. Retrieved from: http://db.csail.mit.edu/labdata/labdata.html (2004)
61. Meidan Y, Bohadana M, Mathov Y, Mirsky Y, Shabtai A, Breitenbacher D, Elovici Y (2018) N-baiot—network-based detection of iot botnet attacks using deep autoencoders. IEEE Pervasive Comput 17(3):12–22. https://doi.org/10.1109/MPRV.2018.03367731
62. Baldi P, Sadowski P, Whiteson D (2014) Searching for exotic particles in high-energy physics with deep learning. Nat Commun 5:4308. https://doi.org/10.1038/ncomms5308

William C. Sleeman IV is an Instructor in the Department of Radiation Oncology at Virginia Commonwealth University, Richmond VA. He received Computer Engineering BS and MS degrees from the University of Virginia in 2004 and Virginia Commonwealth University in 2007, respectively, and a PhD in Computer Science from Virginia Commonwealth University in 2021. His research interests include machine learning, imbalanced data, high performance computing and health informatics.

Martha Roseberry is an Applications Analyst at Virginia Commonwealth University, Richmond VA, USA. She obtained her Ph.D. in Computer Science from Virginia Commonwealth University, her M.L.I.S from Kent State University and her B.A. in Physics from The College of Wooster. Her research interests include machine learning, multi-label learning, and data streaming mining.

Preetam Ghosh is a Professor in the Department of Computer Science at Virginia Commonwealth University. He obtained his MS and PhD degrees in Computer Science Engineering from UT-Arlington and BS in Computer Science from Jadavpur University, Kolkata India. His research interests include algorithms, stochastic modeling, discrete event simulation and data analytics in biological systems and mobile computing related issues in pervasive grids that has resulted in several federally funded research projects from NSF, NIH, DoD and USVHA.

Alberto Cano is an Associate Professor with the Department of Computer Science, Virginia Commonwealth University, USA, where he heads the High-Performance Data Mining Lab. He obtained his BSc degrees in Computer Engineering and in Computer Science from the University of Cordoba, Spain, in 2008 and 2010, respectively, and his MSc and PhD degrees in Intelligent Systems and Computer Science from the University of Granada, Spain, in 2011 and 2014, respectively. His research is focused on machine learning, data mining, data streams, general-purpose computing on graphics processing units, Apache Spark, and evolutionary computation. He has published over 58 articles in high-impact factor journals, 58 contributions to international conferences, two book chapters, and one book in the areas of machine learning, data mining, and parallel, distributed, and GPU computing. Dr. Cano is Area Editor of the journal Information Fusion. He is also the Faculty Director of the VCU High Performance Research Computing core facility.



Bartosz Krawczyk is an Assistant Professor in the Chester F. Carlson Center for Imaging Science at Rochester Institute of Technology, where he heads Machine Learning and Computer Vision (MLVision) Lab. He received the M.Sc. and Ph.D. degrees from the Wroclaw University of Science and Technology, Wroclaw, Poland, in 2012 and 2015, respectively.

Dr. Krawczyk has authored more than 60 journal articles and more than 100 contributions to conferences. He has coauthored the book Learning from Imbalanced Datasets (Springer, 2018). He is a Program Committee member for high-ranked conferences, such as KDD (Senior PC member), AAAI, IJCAI, ECML-PKDD, IEEE BigData, and IJCNN. He was a recipient of prestigious awards for his scientific achievements such as the IEEE Richard Merwin Scholarship, the IEEE Outstanding Leadership Award, and the Amazon Machine Learning Award, among others. He served as a Guest Editor for four journal special issues and as the Chair for 20 special session and workshops. He is the member of the editorial board for Applied Soft Computing (Elsevier).