Applied Intelligence
https://doi.org/10.1007/s10489-023-04482-y

DISCONA: distributed sample compression for nearest neighbor algorithm

Jedrzej Rybicki[1] · **Tatiana Frenklach**[2] · **Rami Puzis**[2]

Accepted: 19 January 2023

Abstract
Sample compression using ϵ-net effectively reduces the number of labeled instances required for accurate classification with nearest neighbor algorithms. However, one-shot construction of an ϵ-net can be extremely challenging in large-scale distributed data sets. We explore two approaches for distributed sample compression: one where local ϵ-net is constructed for each data partition and then merged during an aggregation phase, and one where a single backbone of an ϵ-net is constructed from one partition and aggregates target label distributions from other partitions. Both approaches are applied to the problem of malware detection in a complex, real-world data set of Android apps using the nearest neighbor algorithm. Examination of the compression rate, computational efficiency, and predictive power shows that a single backbone of an ϵ-net attains favorable performance while achieving a compression rate of 99%.

Keywords Distributed machine learning · Malware detection · Nearest neighbors · Sample compression · Big data

1 Introduction

This article discusses distributed sample compression for nearest neighbor algorithms from the perspective of the smartphone security domain. Smartphones have become an integral part of our everyday lives. With annual sales, estimated at 1.373 billion units in 2019 (according to [18]) and this figure expected to increase, they are set to become more widespread. One of the key factors behind this commercial success, is the possibility to extend and adjust their function according to personal requirements by installing various applications (apps). The extreme popularity of the handheld devices and mobile apps also means that they are trusted with more and more personal and sensitive data, ranging from browser history to health and physical activity records and banking information. This trend, in turn, underlines the need to improve the trustworthiness and security of the devices and, therefore, the user data.

There are currently over three million apps registered in the most popular marketplace, Google Play, as estimated by [3], providing the users with a myriad of additional functions. The apps are created by large developer studios, and recognized companies as well as anonymous individuals. The apps are published exclusively in a binary format. It is a huge effort to review their safety and trustworthiness.

In this paper, we present an AI-based method to support the classification of Android apps on a large scale. The content of the apps is analyzed and used to classify the app as malicious or benign. This technique is known as static malware analysis [4, 33]. Each app is disassembled, and a list of functions it uses is extracted. Formally, each app is represented as a data point in a metric space with distances defined according to the sets of their functions. Classifying Android apps based on their nearest labeled neighbor in this kind of representation has been proven to be efficient by [13].

✉ Jedrzej Rybicki
j.rybicki@fz-juelich.de

Tatiana Frenklach
tatianad@post.bgu.ac.il

Rami Puzis
puzis@bgu.ac.il

[1] Jülich Supercomputing Center, Forschungszentrum Jülich, Jülich, Germany

[2] Department of Software and Information Systems Engineering, Ben Gurion University of The Negev, Beer Sheva, Israel

Published online: 23 March 2023

In general, the nearest neighbor search (NNS) is a family of simple yet powerful techniques commonly used in machine learning. No abstract model is fitted to the training data, but each test sample is compared to the most similar training data points. The computational complexity of the NNS depends on the size of the training data set. Compressing the training set by creating an ϵ-net that retains only a small fraction of the original training samples has clear benefits [20]. Intuitively, it uses less space and shortens the search times, but can also reduce the classification accuracy. In our case, the problem at hand is too complex for conventional NNS (the data set is too large), meaning that a sample compression algorithm must be used. Furthermore, given the huge size of the data sets, such as the collections of Android apps, the sample compression process itself requires considerable resources that may not be available on a single machine.

Given the large set of apps and their functions, we aim to compress this set in a distributed manner to efficiently perform an NNS and classify apps as malicious or benign.Distribution is necessary for diving the workload but it also raises additional challenges.For instance, how can the outputs from distributed computations be merged to produce a uniform compressed data set? How does the distribution affect the trade-off between compression and accuracy? Should the compression parameters change in the case of distributed compression? etc.

In this paper, we discuss two approaches for distributed sample compression: the merge-based sample compression and the stream-based sample compression, and evaluate them on a large-scale, real-world data set of Android apps. The main contributions of this paper are as follows:

- We propose a novel distributed sample compression algorithm.
- Our results demonstrate the non-trivial parameterization of the ϵ-net for sample compression.
- We provide insights into the scalability of the proposed solution.
- We show that the compressed NNS achieves a favorable performance under the precision-recall curve of 0.9884 with a compression ratio of 0.9767.
- We attain these favorable results for a real-world problem, which was too complex for a conventional NNS solution.

The rest of the paper is structured as follows. Section 2 reviews the related work and provides a theoretical background to our algorithm. We then describe the details of the proposed solution in Section 3. We evaluate the presented solutions in Section 4 and conclude our paper with a summary and outlook in Section 5.

2 Related work

Before explaining our approach, we provide a brief review of relevant works in this field. The two relevant topics that constitute the basis for this work are approaches to malware detection and theoretical backgrounds to sample compression for the nearest neighbor search.

2.1 Malware detection

There is a substantial body of work on mobile app malware detection. Our approach belongs to the static analysis domain [4, 33] which analyzes the content of the app (and its metadata) rather than the runtime behaviour as in the case of dynamic methods [32, 40]. Both approaches have their strengths and weaknesses. With dynamic analysis, it is difficult to enforce all the possible execution paths of an app. This process can be made even more difficult if the attacker uses anti-tracking and anti-debugging techniques. Under certain circumstances, it therefore cannot be guaranteed that the app will not became malicious. Static analysis, on the other hand, can be made more difficult to conduct due to code obfuscation, i.e. deliberately making code more difficult to read and comprehend. This problem is to a certain extent, orthogonal to our work. There are a number of anti-obfuscation solutions; an extensive overview is prepared by Zang et al. [42].

Regardless of the way the information about an app was collected (dynamically or statically), it must be analyzed to classify the app as malicious or benign. A wide range of techniques and solutions were also employed here. A number of popular machine learning techniques were used: k-means by [39], vector-embedding and support vector machines by [4], and (deep) neural networks in [31]. For an extensive overview of the current state of the work in this field we refer the reader to a survey by Odusami et al. [28].

Our work is based on the foundations laid by [13], where among the others an overview of the effectiveness of the proposed methods in included. Those are summarized in Table 1. Neither one of these works considers distributed data processing. Moreover, distributed kNN is not used for static malware detection as is it would require searching through all the samples in the large malware databases each time a classification is made. Even if done in a distributed way this would be prohibitively expensive. Since we do not see such solutions in prior art our baseline is the non-distributed case. Our sample compression scheme on the other hand creates an implicit model (capturing domain knowledge) and speeds-up the subsequent classification. This is a crucial characteristic for the problem at hand.

Table 1 Summary of related work in static analysis [13]

Reference	App Representation	Data size	Algorithm	Accuracy	Time [s/app]
Chen et al. [8]	n-grams	2400	RF, SVM, KNN, NB	TPR=0.988; FPR=0.077	2.9
Taheri et al. [35]	API, permissions, intents	71103	FNN, ANN, WANN, KMNN	Accuracy from 0.9 to 0.99	NA
Frenklach et al. [13]	apps functions' definitions and executions, ASG, ASGnode2vec	10003	KNN, RF	AUC=0.987	>5E-2
Our method	app functions' definitions	10003 188452	KNN	AUC=0.988	>4E-3

2.2 kNN and compression for NNS

Nearest neighbor search, proposed originally by [15] remains a popular and powerful machine learning technique. Formally, given a set $S \in X$ of points in a metric space (X, d) with a distance function d and a query $q \in X$, the nearest neighbor search localizes the nearest point (in terms of d function) to q among the S. Translating this to our the setting of our study, for a given app we find an app closest to it in the unrestricted space of all apps based on the similarity in their function usage.

The two main challenges when employing kNN are the selection of value k and proper distance function, as highlighted by [41]. There are many different distance functions that can be used with kNN: Euclidean, Mahalanobis [38], Minkowski [22], Levenshtein [10], etc. There is no perfect distance function. The function is application-specific and should be able to detect the similarity between samples, allowing them to be compared and classified. In our work, we opted to use the Jaccard distance function based on the encouraging results from previous work on malware detection by [13]. The correct setting for k parameter is also specific to the application (data set) and interlinks with the selection of the distance function [41]. We refer the reader to the original work of Kontorovich et al. [20] for theoretical discussions of the value of the k parameter in ϵ-nets and selected $k = 1$ for our evaluations.

In most of the settings, the nearest neighbor search is a simple yet effective classification algorithm. In real-world situations (such as the one we are dealing with), however, it can suffer from a number of problems. It has high storage requirements (the training set needs to be stored); the efficiency of the classification declines with the increasing size of the data (i.e. more distance calculations are needed); and they have low noise tolerance (especially for the 1-NNS). As shown by [7], all these shortcomings can be addressed by data reduction techniques. The idea is to obtain a representative data set from the training data set that is smaller and can still be used to perform NNS with good accuracy. The accuracy on the compressed set is sometimes even higher, as it reduces the noise present in the full data set. The reduction techniques have different names: instance selection, prototype selection, data set condensations, and coresets. Regardless of the name, the goal is always to remove noisy and redundant data from the original data set before running the classification.

Coresets are probably the most general theoretical framework for sample size reduction [27]. "A coreset is a reduced data set which can be used as proxy for the full data set; the same algorithm can be run on the coreset as the full data set, and the result on the coreset approximates that on the full data set" [30]. Coresets are specific to the algorithm; there are solutions for the smallest enclosing ball, ϵ-kernel, quantiles, k-means, and k-median clustering (see [30] for an excellent overview). In addition, the coresets are built with many assumptions regarding the data set to derive fundamental guarantees on the upper bounds on the cardinally of the coreset. Our work, on the other hand, deals with a practical problem. It has to deal with noise and inconsistencies found in the data set. The application of coresets in a distributed setting also requires a merging algorithm that is specific to the algorithm used. A coreset for NNS proposed by [11] thus cannot be directly applied to our problem.

One of the first proposed data reduction techniques for the NNS is the Condensed Nearest Neighbor (CNN) Rule by [15]. In short, the algorithm takes an arbitrary starting point to initialize the condensed set. Remaining points from the training set are considered one at a time and if their nearest neighbor in the condensed set differs from their actual label, they are added to the condensed set. The algorithm has three main drawbacks: it is order-dependent, cannot handle inconsistent points (i.e. points with the same attributes but different labels), and has bad running times. The CNN rule therefore has to be extended and modified multiple times e.g. [2, 12, 23]. The fundamental idea behind these approaches is to split the CNN rule into two phases. In the first phase, instead of the random initialization of the condensed set, a representative of the training set is selected, which is then refined in the second phase of the algorithm. With a good initialization technique this ensures that the algorithm is order-independent. The condensation techniques use labels from the training set to improve the performance of the NNS algorithm.

In this study, we argue that real-world applications often produce inconsistent data sets, i.e. sets with points that have

the same coordinates but different labels. Our algorithm does not rely on the correct labels, or the consistency of the training set. We believe that if the problem at hand enables the use of labels, such a refinement could benefit from our work. An ϵ-net is a representation of the data set and could be refined in a similar way by classifying the remaining training points to boost its accuracy. MCNN was also used as a starting point for a parallel MCNN by [9]. The proposed algorithm works in a distributed setting but requires a lot of communication between cooperating nodes (probably MPI-based). Our algorithm, on the other hand, reduces the level of communication (as the partitions are analyzed independently), rendering the calculation more robust in a distributed setting.

2.3 ϵ-net-based compression for NNS

Papers by Kontorovich et al. [19, 20] form the theoretical basis of this work. They propose a novel approach for generating a subset for a nearest neighbor rule, i.e. sample compression that can still achieve good performance for predictions. For estimations of the prediction error at a given scale as well as the complexity of the set creation, we refer the reader to the original works. In our case, we focus more on the practical implications of such a compression, in particular the problem of distributing the compression process, which is not addressed in the aforementioned theoretical backgrounds.

In practice, real data sets may contain identical data points with different labels due to insufficient data or noise. This can be addressed by taking a majority vote among the k nearest neighbors, as suggested by [34]. We use a similar technique in the process of creating the compressed data set and for app classification. An alternative to majority vote are algorithms based on fuzzy class membership as reported in [6]. We believe that this technique could be integrated into our algorithm, although it is not beneficial for the crisp binary classification problem at hand.

Our compression uses point networks (also known as ϵ-net), as described by [21]. They also proposed to use the hierarchy of such networks to speed-up the search.Speeding up the search in this way was not our primary goal, but it could be an interesting avenue for future work.

One of our proposed algorithms (Stream DISCONA in Section 3.3.2) was inspired by on-line incremental learning. In short, the idea of incremental learning is to perform model creation when only parts of the data are available and to then update the models when new data arrive. Further information on such approaches is provided by [16] and [25].

Finally, it is worth mentioning the seminal work by Littlestone and Warmuth [24], who suggest that the compression process is reminiscent of learning. A compressed set of samples can thus be viewed as a degenerated model learnt from the distributed train data.

3 DISCONA algorithm

Our DIstributed Sample COmpression Algorithm (DISCONA) is based on the point networks. Such structures allow for a space-efficient representation of the metric spaces, while enabling a nearest neighbor search.

3.1 Point network creation

Let (X, d) be a metric space, in which X represents a set of points and d denotes a distance function. Let $K(y, \epsilon) = \{x \in X : d(y, x) \leq \epsilon\}$ denote a sphere of radius ϵ around y. Given ϵ, a point network of X is $Y \subset X$ that satisfies two conditions:

1. for every $x, y \in Y, d(x, y) \geq \epsilon$ and
2. $X \subseteq \cup_{y \in Y} K(y, \epsilon)$.

We define $d(p, Y) = \max_{y \in Y}\{d(p, y)\}$. Thus, for every node in a point network $y \in Y$ it is understood that $d(y, K(y, \epsilon)) \leq \epsilon$. Throughout this paper, we refer to a point network for a given fixed ϵ as an ϵ-net. A simple, brute-force algorithm for a point network creation is provided in [14].

The compression ratio of an ϵ-net depends on the value of the ϵ parameter. The higher the ϵ, the less points will be selected to constitute the network. For $\epsilon = 0$, on the other hand, only compression resulting from the removal of duplicates will be performed. Once an ϵ-net is created, the classification comes down to finding the nearest point to the given query q, but only out of the points in the ϵ-net ($Y \subset X$). q is then classified to the same class as its nearest neighbor in Y.

The metric space (X, d) comprises a set of points as well a distance function d. There are many possible distance functions to choose from. Based on the previous results in this area [13], we decided to use the Jaccard distance, which is based on the Jaccard coefficient. The coefficient measures the similarity between sets. For two sets, it is defined as the size of their intersection divided by the size of their union. The Jaccard distance is complementary to the coefficient:

$$d_j(A, B) = 1 - \frac{|A \cup B|}{|A \cap B|}. \quad (1)$$

In our case, the distance calculation between the apps is based on the set of functions they use. When pre-processing the data set, each app and each function becomes a unique (hash-based) identifier [see, 13]. If we take Example 1 of

two apps app_0 and app_1 which use functions as defined in the curly braces. The applications have five unique function usages two of which are common (101925, 178583). Thus, according to the previous definition, the Jaccard distance between these applications is 0.6.

Example 1 Jaccard distance calculation

$$app_0 = \{101925,\ 178583\}$$
$$app_1 = \{101925,\ 178583,\ 65996,\ 207973,\ 342566\}$$
$$d_j(app_0, app_1) = 1 - 2/5 = 1 - 0.4 = 0.6$$

An ϵ-net constitutes a compression scheme for a given set. Berend and Kontorovich [5] have showed that such a network, despite containing less information, can be used to correctly perform majority voting and the results will be consistent with the results of the majority of the nearest neighbors running on the full data set.

Instead, completely disregarding data points not in the ϵ-net, we use a slightly modified data structure. In the training (i.e. compression) phase, for each point of the ϵ-net (we refer to them as anchors) we store aggregated information about labeled data points in its vicinity, i.e. the distribution of app labels in $K(y, \epsilon)$ as shown in Algorithm 1. This aggregated information also helps to derive confidence bounds on the actual classification.

Require: ϵ
1: $Y \leftarrow \{$ random point from $X\}$
2: **for all** $p \in X$ **do**
3: $\quad y' = argmin_{y \in Y} d(p, y)$
4: $\quad$ **if** $d(p, y') \geq \epsilon$ **then**
5: $\quad\quad Y \leftarrow Y \cup \{p\}$
6: $\quad\quad y' \leftarrow p$
7: $\quad\quad C_{y',\mathcal{M}} \leftarrow 0$
8: $\quad\quad C_{y',\mathcal{B}} \leftarrow 0$
9: $\quad$ **end if**
10: $\quad$ inc $C_{y',l(p)}$
11: **end for**

Algorithm 1 Point network with label distributions.

Let malicious ($\mathcal{M}$) and benign ($\mathcal{B}$) denote the labels assigned to apps in X and $l : X \rightarrow \{\mathcal{M}, \mathcal{B}\}$ be the labeling function. Let $C_{p,\mathcal{M}}, C_{p,\mathcal{B}} \in \mathbb{N}$ denote the number of malicious and benign apps respectively in $K(p, \epsilon)$. Algorithm 1 maintains the incumbent ϵ-net Y and collects there every point that is farther than ϵ from existing anchors (see lines 3-5). Every point p in vicinity ($p \in K(y, \epsilon)$) of the nearest anchor $y \in Y$ is aggregated according to its label (see line 10). Note that every anchor is in vicinity of itself.

For the ϵ-net example in Example 2, the app with $id = 0$ is an anchor point and it aggregates information about 19 apps (including itself), which are all malicious. The app with $id = 4096$, on the other hand, aggregates information about 24 apps, 23 of which are benign.

Example 2 An aggregating ϵ-net

```
{
    0: [0, 19],
    16384: [595, 0],
    24576: [0, 116],
    4096: [23, 1]
}
```

3.2 Merging

The sheer size of the data we have to deal with, renders the calculation of ϵ-net s on a full set impractical. We, therefore look into a distributed solution. The set of apps is partitioned and the partitions are distributed among several compute nodes. All the information about an app (the functions used) is stored together on the same compute node. Each node derives an ϵ-net by processing the local data. The networks are subsequently merged together. Since the networks already compress the locally available data, their size should be much smaller than the size of the entire partition. Such an exchange is, therefore, feasible. In the following sections, we discuss different possibilities of merging results from partitions to ultimately form the solution.

3.2.1 Conservative merging

To form a single network from the ϵ-net s of the partitions, they need to be merged together. The following approach allows each anchor to aggregate only the data points that are closer than ϵ in the original data set. Given a set of ϵ-net s $Y_1, Y_2, \ldots$ create a network $\mathcal{Y} = \bigcup Y_i$. All the anchors from the input networks become anchors in the resulting network, retaining their label distributions. Only the label distributions of coinciding anchors are added together. The following example involves the merging of two networks, Y_1 and Y_2:

Example 3 Conservative merging

```
Y1 = {0: [0, 19], 4096: [23, 1], 16384: [595, 0]}
Y2 = {1: [1, 5], 4096: [12, 1], 5091: [150, 33], 9011:
    [411, 20]}
Y = {0: [0, 19], 1: [1, 5], 4096: [35, 2], 5091: [150,
    33], 9011: [411, 20], 16384: [595, 0]}
```

All unique anchors in each partition are transferred to the resulting network $\mathcal{Y}$ (including the label distribution). For

the common anchor (4096), the label distributions are added together. The resulting network $\mathcal{Y}$, however, may not satisfy the first condition in the definition of ϵ-net. It is likely to contain anchors that are closer to each other than the given ϵ. In practical terms, the resulting network is larger than it could have been, achieving lower compression.

3.2.2 Aggressive merging

Another possibility for merging two networks,is to build the ϵ-net hierarchically with $\bigcup Y_i$ as input data points. If two anchors from the input networks are closer than ϵ, then only one of them may be retained in the merged network $\mathcal{Y}$. As a result, all anchors in the merged network satisfy the condition $x, y \in \mathcal{Y}, d(x, y) \geq \epsilon$.

When an anchor $x \in Y_i$ is closer than ϵ to one or more anchors in $\mathcal{Y}$, its label distribution is aggregated by the closest one. This in another imperfect solution. Unlike the conservative merge, in this case the anchors may aggregate information from apps that were at a distance larger than ϵ in the original data set because $x \in K(y, \epsilon) \wedge y \in K(y', \epsilon) \nRightarrow x \notin K(y', \epsilon)$ Thus, we define the *effective radius* $\dot{\epsilon} \geq \epsilon$ of the merged network $\mathcal{Y}$ in such a way that each anchor $y \in \mathcal{Y}$ aggregates labels from data points in the original data set X at a distance of $\dot{\epsilon}$ or less.

3.3 Distributed network creation

Provided a mechanism to merge networks calculated at distributed computation nodes, we can create an ϵ-net for larger data sets. In this paper, we evaluated two algorithms of distributed network creation. We assume that each compute node has a partition of the original data set. The partitions are random, roughly equal in size, and do not overlap. In general, such an assumption can be achieved, for instance, by means of consistent hashing, attributing apps to particular compute nodes.

3.3.1 Merge-based DISCONA

In the merge-based approach, all nodes calculate ϵ-net in parallel for their partition of the data. The resulting networks are subsequently passed to a node responsible for merging. The process is schematically depicted in Fig. 1. Depending on the size of the resulting networks, the merging can either be performed in one shot, or as a sequence of smaller (e.g. pairwise) merges. In the second case, more nodes are responsible for merging, and thus the workload can be better distributed. However, more merges (of smaller networks) should be conducted so that the overall workload might even increase.

For this kind of network creation, aggressive merging is preferable. The anchors of the resulting networks are used to calculate a set of anchors for the merged network. Conservative merging would substantially increase the size of the resulting network, and its overhead depends on the number of partitions.

3.3.2 Stream-based DISCONA

In this case, one partition (i) is initially selected at random as an origin, and an ϵ-net is calculated for it. The idea here is to retain all anchors of the origin network ($\mathcal{Y} = Y_i$) and to only update their label distributions based on the ϵ-net s of other partitions. The origin network is then passed to other compute nodes. Each node j adjusts the label distributions of the anchors in origin network Y_i. Technically speaking, each node performs an NNS for the local data and origin anchors. Each local data point $x \in X_j$ is attributed to one anchor $y \in Y_i$ from the origin network. This attribution is used to create a local ϵ-net with the same anchors as Y_i but with label distributions specific to each partition. In the last step, the networks from all compute nodes are merged together in a conservative fashion. The process is shown in Fig. 2.

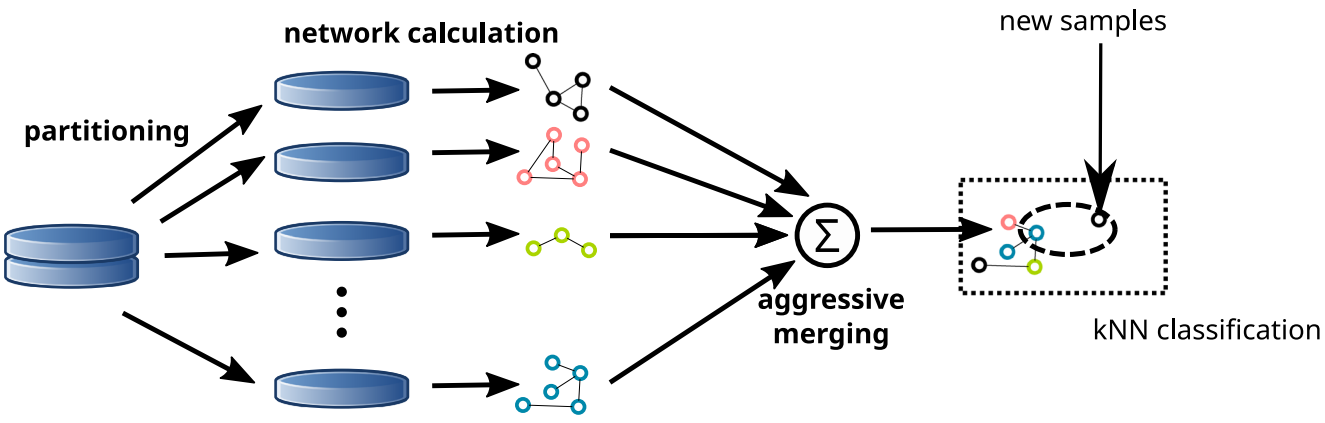

Fig. 1 Merge-based compression

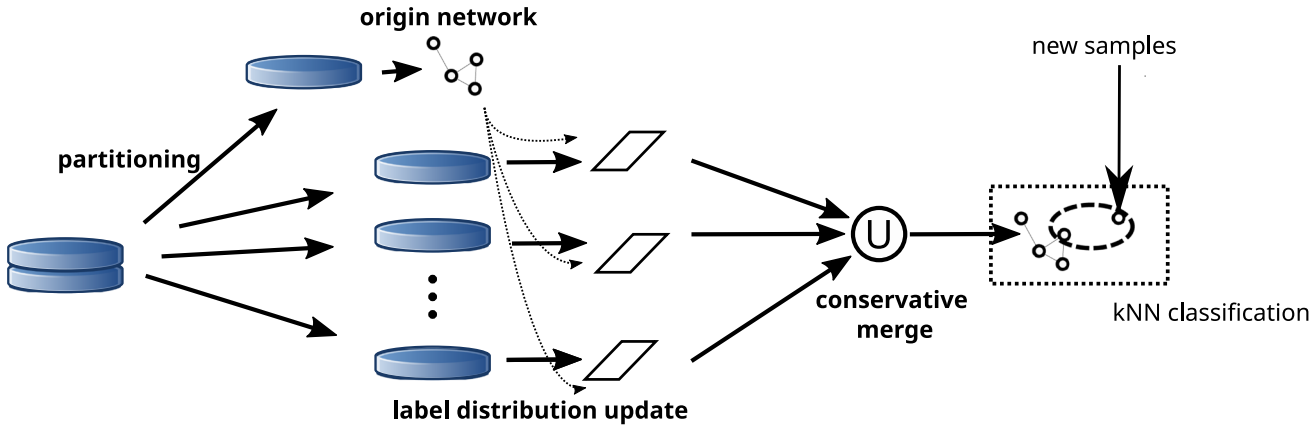

Fig. 2 Stream-based compression

The use of the aggressive merge algorithm is evidently a waste of resources in this case, as the anchors of the resulting networks stem from the origin network that was already a correct ϵ-net. Since the aggregation is based solely on the nearest neighbor calculation and only uses ϵ as an implicit, hidden parameter, the resulting network might have an effective $\dot{\epsilon} \geq \epsilon$.

4 Evaluation

Here we present the results of our study. We examine the compression achieved by our solution and compare it with the performance of the model working on the compressed data.

4.1 Setup and data sets

We conducted our experiments based on the data sets used in [13]. We refer the reader to the original work for details on how the data sets were created, and pre-processed. It is, however, important to mention one processing step: The data sets only include nonempty functions that are defined and/or used in more than 100 apps.

The code in this study was implemented in Python using popular libraries. In particular, we used pandas [26] and Scikit-learn [29] for data manipulation and model evaluation. The Matplotlib library [17] was used to create visualizations. The machine learning library Turi Create by [36] provided us with ways to efficiently calculate Jaccard similarity and perform NNS.

Throughout the evaluation, we use two data sets of different sizes. The smaller one enabled quick hypothesis testing, as some experiments are too expensive to be conducted on the full data set. In particular, the research question on effective ϵ requires a lot of pairwise distance computations and was, therefore, studied on the basis of the smaller subset. In addition, initial results on compression

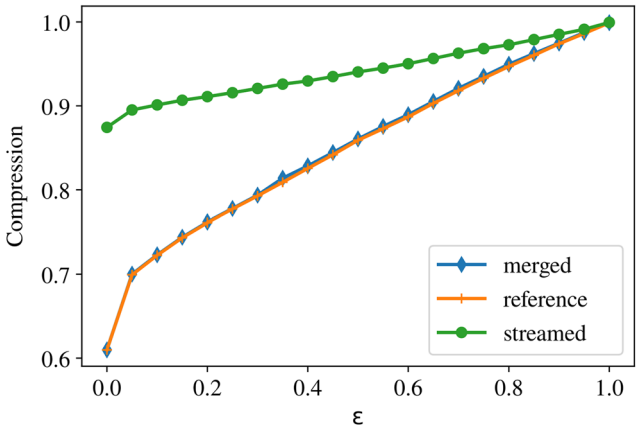

Fig. 3 Compression ratio for increasing radius

ratios, choosing ϵ, and the robustness of the streaming DISCONA were all obtained with the smaller set of 10003 apps: 4987 malicious apps obtained from [37], and 5016 benign apps collected using Androzoo API [1]. This data set (referred to as VTAz) was composed by Frenklach et al. [13] and is used here for performance comparison. The apps used about 700000 unique functions. The overall size of the data set was over 35000000 records. Prior to the experiments, we withheld a random test set of 1000 apps, which we later used to asses the quality of the predictions. The remaining data were divided into 4 distinct partitions.

After gaining the initial insights, the results from the VTAz data set were transferred to experiments with a large-scale Virus Total (VT) data set. It comprised of 95220 benign and 94241 malware apps obtained from [37]. An app is tagged as malware if it is detected as malicious by five or more VT anti viruses. The data set consisted of 188452 unique apps and 1052842 unique functions. The withheld test set comprised 1000 randomly selected apps and remained constant throughout the experiments.The remaining apps were dividend into 16 distinct, non-overlapping partitions.

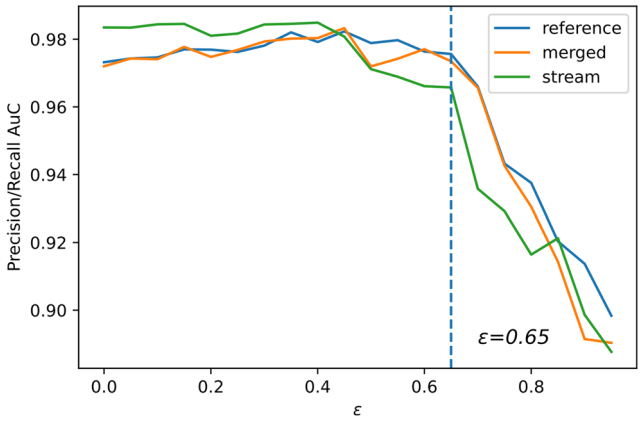

Fig. 4 Model performance at different network radius

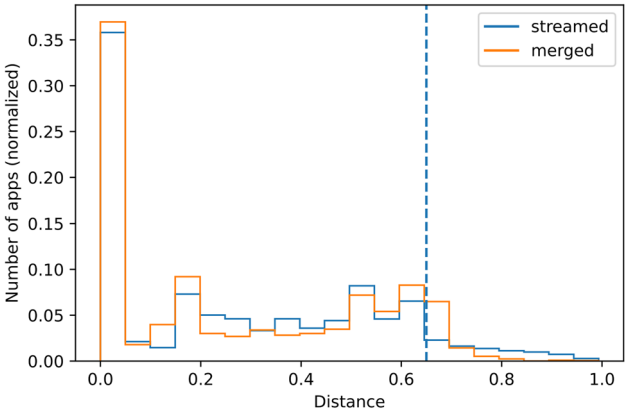

Fig. 7 Comparison of normalized distance distribution between streamed and merged network for $\epsilon = 0.65$

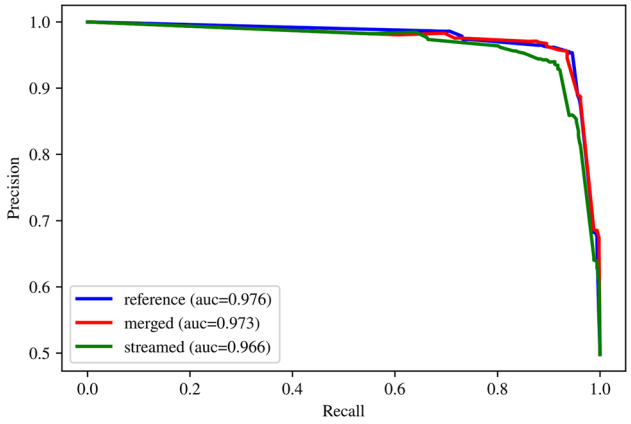

Fig. 5 Comparison of model performance for fixed $\epsilon = 0.65$

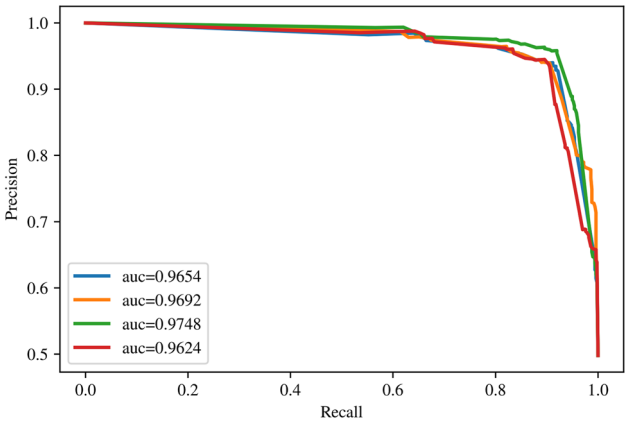

Fig. 8 Performance stability for the streamed network

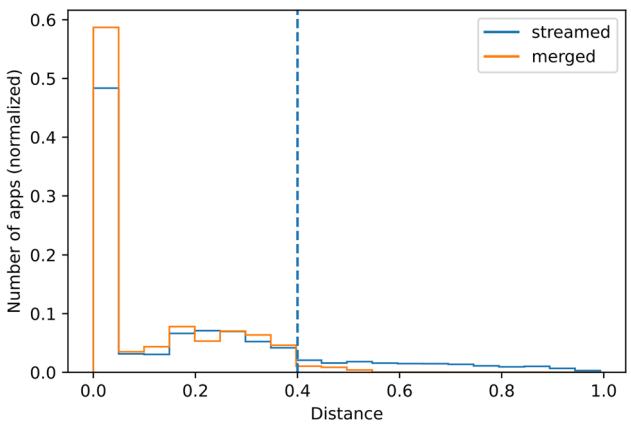

Fig. 6 Comparison of normalized distance distribution between streamed and merged network for $\epsilon = 0.4$

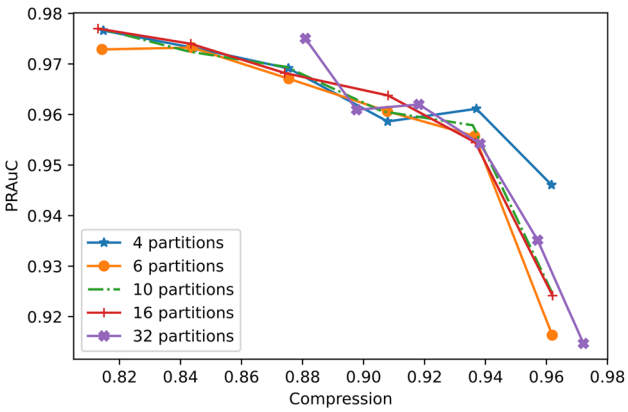

Fig. 9 The influence of number of partitions on the performance (merged network)

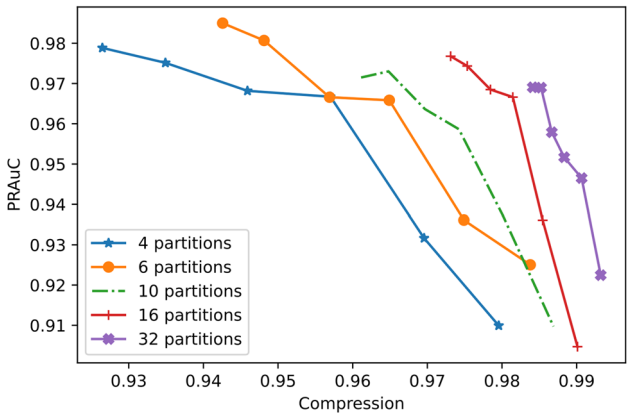

Fig. 10 The influence of number of partitions on the performance (streamed network)

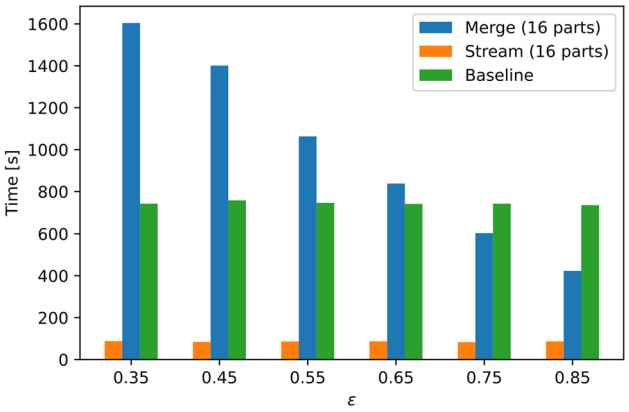

Fig. 13 Network creation times. Baseline adapted from [13]

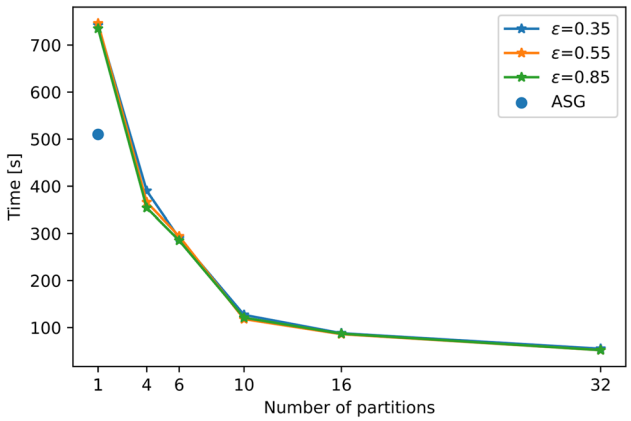

Fig. 11 Streamed network creation times

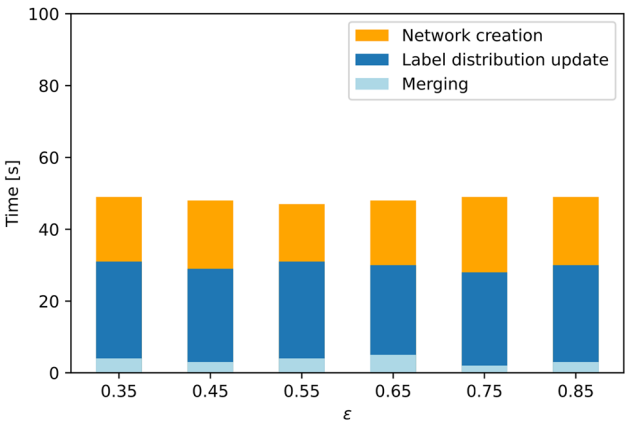

Fig. 14 Breakdown of streamed network creation phases (16 partitions)

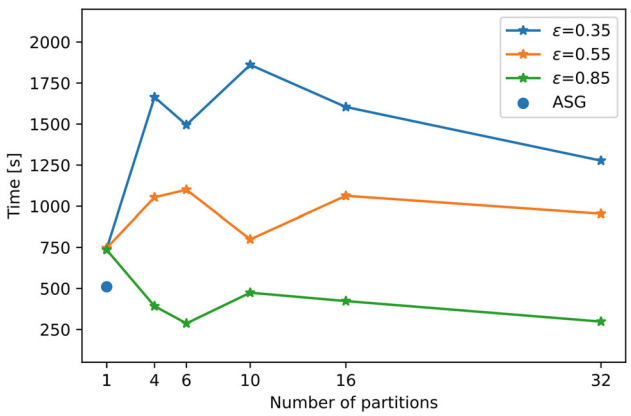

Fig. 12 Merged network creation times

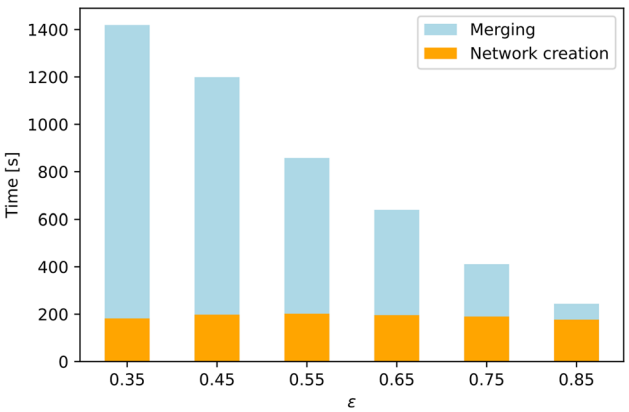

Fig. 15 Breakdown of merged network creation phases (16 partitions)

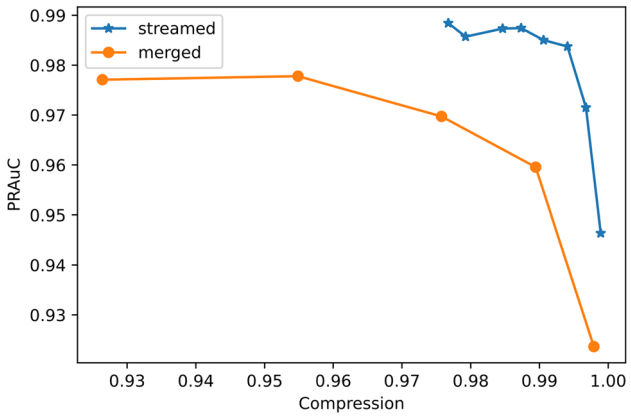

Fig. 16 Model performance as function of compression obtained on the full data set for different values of ϵ

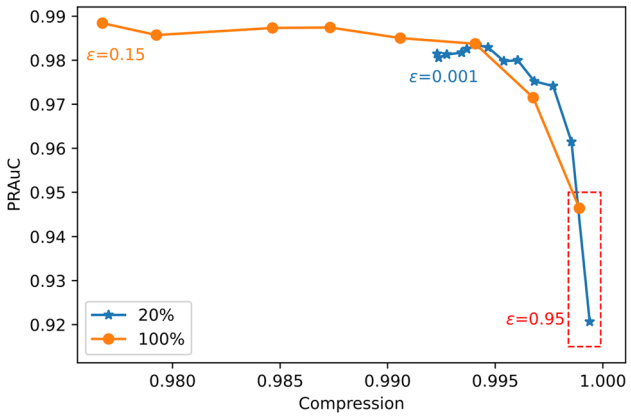

Fig. 17 Compression and performance of the streamed network for the origin created only from fraction (in percent) of initial partition

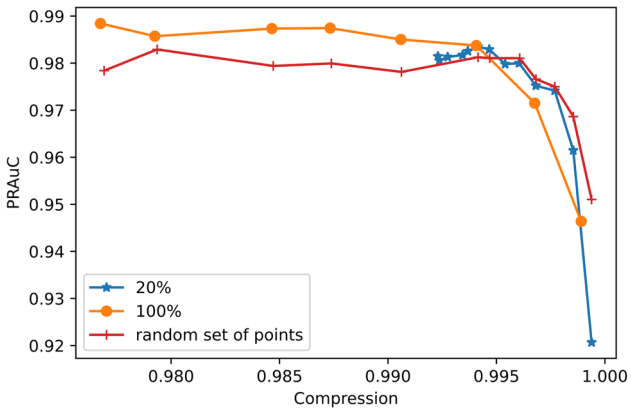

Fig. 18 Compression and performance of the streamed network for 100%, 20% origin, and random selection of points

4.2 Compression vs. predictive power

The goal of our work was to achieve the highest possible compression while preserving the high predictive power of the model using the compressed data. The compression ratio of the algorithm can be regulated using the radius parameter ϵ. The relation between compression and the radius is presented in Fig. 3 and was examined using VTAz. We define compression as the complement of the ratio between the size of the ϵ-net (number of anchors) divided by the size of the input data set (unique apps). The stream-based solutions result in stronger compression. Furthermore, it is worth noting that even for $\epsilon = 0$, we achieve substantial compression resulting from the removal of apps versions sharing the same frequent functions.

4.3 Phase transition of the ϵ hyper-parameter

We use precision and recall as the primary performance metrics in our evaluation. To calculate the performance metrics, we made predictions using resulting nets on the withheld set. To compare performance across different compression ratios using a single unparametrized measure, we used the area under the precision/recall curve (precision/recall AuC or PrAUC for short). We believe that malware classification is indeed best characterized by a trade-off between precision and recall. Precision shows how specific the malware detection is. For example, $1 - Precision$ quantifies the human effort involved in handling non-malware applications mistakenly classified as malware. Recall corresponds to the malware detection rate, quantifying the fraction of malware applications correctly classified as malware. Figure 4 shows the reduction in precision/recall AuC with increasing ϵ.

We compare three methods. The reference method calculated the ϵ-net for the entire data set without distribution. The merged and streamed networks are created in a distributed way. The performance appears to be stable (and high) for values up to $\epsilon_0 = 0.65$, as indicated by the dashed line on the plot. After this threshold, performance declines substantially for all compression schemes considered. In order to produce reference results, we used VTAz for this experiment.

For a better comparison of the networks, we present the full precision/recall plot for the threshold value of radius $\epsilon = \epsilon_0 = 0.65$ in Fig. 5. It is important to note that both distributed solutions achieve the performance of the reference network. The streamed network is only slightly worse than the merged one. Based on the results presented in Fig. 3, the compression for $\epsilon = \epsilon_0$ is 9.47×10^{-2} for the merged and 4.35×10^{-2} for the streamed network. Also, in terms of overall problem-specific model performance we are achieving pretty good results (compare Table 1).

4.4 The effective ϵ

As mentioned earlier in Section 3.2, the distributed construction of an ϵ-net may reduce its quality by aggregating apps in label distributions of incorrect anchors. This behavior manifests itself as an increase of the effective ϵ in the network, i.e. there will be pairs of app-anchors in the data set that are at a larger distance than the requested ϵ. To empirically assess the magnitude of this problem, we plot the normalized distance distributions for both merged and streamed networks in Figs. 6 and 7. The results were produced using the smaller data set VTAz. The dashed line represents the desired value of ϵ. The distance distribution tail to the right of the dashed line can be considered as an error. We can see that the distance distribution for the stream-based network is more skewed than for the merged one, and has a longer right tail. It means that the label distributions of anchors in the stream-based network are affected by apps that are significantly less similar to the anchor than in the case of the merge-based network. In particular, these apps include very unique functions, which do not have much in common with other apps in the data set. Such unique apps scattered across partitions, do not become anchors in the origin network, and thus are also not anchors in the output network $\mathcal{Y}$.Their exclusion from ϵ-net increases the compression ratio compared to the reference network and to the merge-based one in Fig. 3.

4.5 Robustness of the streamed network

The performance of the stream-based DISCONA depends on the selection of the initial anchors (origin network). To evaluate the sensitivity of the stream-based DISCONA to such a random selection, we conducted four additional experiments with the VTAz data set. With each run, a different partition was used to create the origin network (initial anchors selection). The results are depicted in Fig. 8. Here, we again fixed the network radius at the previously identified $\epsilon_0 = 0.65$ value. Although, there are some differences in performance, we believe that the overall stability of the streamed network performance is pretty high with regard to the selection of initial anchors. We also include the values of area under the curve, which show that the stream-based network is capable of producing results similar to the merge-based network in Fig. 5.

4.6 Performance as a function of the number of partitions

The number of partitions can influence the performance of distributed sample compression both in terms of the classification accuracy and the running time. A single large partition is the most accurate (at high ϵ) as it becomes the reference network (see Figs. 4 and 3 respectively). Here we investigate the influence of the number of partitions on the performance of sample compression. To this end, we divided the VTAz data set into 4,6,10,16, and 32 partitions.

As can be seen in Fig. 9, the merged network is insensitive to the number of partitions. We attribute the robustness of the merged network to the aggressive merging strategy used in the merge-based network construction. In contrast, in the streamed network creation, increasing the number of partitions increases the compression ratio (Fig. 10), on one hand. However, the higher compression ratio comes at the cost of classification accuracy. Updating the label distributions during conservative merging is unable to compensate for the loss of potential good anchors due to the decreasing size of the first partition.

4.7 Scalability evaluation

To evaluate the scalability of the proposed approach we measure the network creation times. Figure 11 depicts the network creation time as a function of the number of partitions for three selected values of ϵ on the VTAz dataset. We can see that regardless of the radius parameter the overall running times decline with increasing number of partitions.

We compare DISCONA to a single partition baseline adapted from [13]. The baseline finishes in 746 secs. which is 50% higher than they report for the same data set. We attribute this discrepancy to the difference in implementation and hardware. The streamed DISCONA is performing much better than the baseline across the range of ϵ values and with only few partitions. The network creation time drops down to 88 secs. and 55 secs. with 16 and 32 partitions respectively. This shows the benefits of the parallel label distribution update in the streamed DISCONA algorithm.

The situation is very different for the merged case, where the effect of the increase in the number of partitions is equivocal (see Fig. 12). Similar to the streamed case, the partitions are processed in parallel. However, the subsequent step of aggressive merging takes quadratic time $O\left(\left(\sum_i |Y_i|\right)^2\right)$ where Y_i is the network created for partition i. Thus, for higher values of ϵ, we observe improvements in the running time.

A comparison with a baseline model creation is depicted in Fig. 13. The number of partitions is fixed to 16. The streamed DISCONA shows favorable compute times. The merged DISCONA is faster than baseline only for higher values of ϵ.

To further corroborate the differences between the scalability of the streamed and merged cases we analyzed the breakdowns of the running time for a fixed number of 16 partitions. The time to create a streamed network (Fig. 14)

is dominated by the origin network creation (in orange) and label distribution update (in blue). For the merged case, however, as can be seen in Fig. 15), the dominating factor is the aggressive merge of networks Y_i (in light blue).

4.8 Performance as a function of compression

After performing the extended parameter study on the initial (smaller) VTAz data set, we applied the obtained knowledge to our full VT data set and used 16 partitions. In the stream-based DISCONA, the origin network was created from the first partition.

Firstly, we evaluated the achieved performance as a function of compression (see Fig. 16). Throughout this section, we use area under the precision/recall curve as a performance metric. In total, we created 8 streamed and 8 merged networks, with increasing ϵ values [0.15, 0.25, 0.45, 0.55, 0.65, 0.75, 0.85, 0.95]. Overall, the stream-based DISCONA exhibits a better trade-off between compression and PRAuC than the merge-based algorithm due to the exclusion of unique apps from the set of anchors.

The compression ratio of the streamed network has a lower bound corresponding to the size of the origin network. It can only use apps from the first partition in the origin network, even for very small or zero values of ϵ.

4.9 Sub-partitions for the streamed network

The compression of stream-based DISCONA can be influenced by the size of the origin network. We examined this in an experiment where the origin network was created from a subsample of the first partition from the VT data set. We generated an origin network for the 20%, 40%, 60%, and 80% of the apps from the first partition and then proceeded normally updating by the label distributions for each anchor according to the 16 VT partitions (including the remainder of the first partition). Thus, the networks comprise the same amount of information, although the number of anchors is (artificially) reduced.

The results of this experiment are presented in Fig. 17. For the purpose of clarity, only networks with origins created from 20% and 100% data points in the first partition (extreme case) are depicted. For each subpartition, we calculated networks with an increasing radius $\epsilon \in [0.001 : 0.95]$. The dashed rectangle, denotes the highest value of $\epsilon = 0.95$. We also noted the lowest values of ϵ for each network.

Sub-sampling the initial partition naturally allows for a further increase in the compression ratio. On the one hand, the performance of a network created from the full partition and $\epsilon = 0.85$ is inferior to the use of $\epsilon = 0.75$ and the 20% subsample, which results in a substantially smaller model. On the other hand, the best performance is achieved by ϵ-net s created from the full partition and low values of ϵ (left-hand side of the plot), as expected.

With the 20% partition, we almost reach the minimal compression ratio with $\epsilon = 0.001$. With these settings, 98% of unique apps in the sub-partition become anchors in the origin network similar to the phenomena described in Section 4.8.

4.10 Random networks

A substantial drop in performance for the high values of ϵ (dashed rectangle in Fig. 17) is caused by aggregating the vast majority of apps by the first (random) anchor. The remaining potential anchors aggregate a small number of apps, each diminishing the predictive power of the ϵ-net. These are clearly not the best settings for the proposed DISCONA algorithm, as it cannot show its full potential. In this regime, the proposed algorithms are likely to exhibit an inferior performance than a set of randomly selected points of the same size. To verify this hypothesis, we conducted one more experiment with the full VT data set, where instead of creating origin networks we selected a random set of points from the first partition.

The results of this experiment are presented in Fig. 18. A random selection of points indeed achieves a better performance than a point network with the highest ϵ values ($\epsilon > 0.994$). With decreasing ϵ and an increasing size of the random subset, however, the advantages of DISCONA can be clearly seen (left side of the plot). This is the regime where the algorithm can use its sophistication to select a good (rather than just small) set of representative malicious and benign apps. It should also be stressed that the random sample is used as an origin network and enriched by information from remaining partitions of our streamed DISCONA algorithm, which substantially increases its predictive power.

5 Conclusion

In this paper, we presented the first distributed sample compression for NNS based on ϵ-net. It is based on point-network generation and subsequent merges of the partition results. The algorithms were evaluated with a real-world data set to perform Android malware classification, and they solve the problem very well, achieving a performance of 0.9884 (measured as area under precision/recall curve) while maintaining a compression ratio of 0.9767. We extensively examined the significant trade-off between the compression and predictive power of the NNS, showing the best range of the ϵ parameter to work with the data set. We also demonstrated the scalability of the solution.

A future work, may examine alternative ways of building the ϵ-net. In particular, the aggressive merge phase can be accelerated by applying the distributed hierarchical merge approach. In addition, an application of the proposed solution beyond the malware classification, possibly requiring some domain-specific tweaks, should be pursued. A clear extension of the proposed algorithm is a multi-class classification case. This is theoretically possible, as shown by [19], but would require changes in the internal structures of the ϵ-net (see Example 2).

One of the proposed algorithms, stream-based DISCONA, allows for on-line learning which is relevant in the application area and could be further examined.

Acknowledgements The authors would like to thank the Helmholtz Information & Data Science Academy for enabling the cooperation. This research was also partially supported by the Israeli Council for Higher Education (CHE) via the Data Science Research Center, Ben-Gurion University of the Negev, Israel.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Ethics approval All authors contributed to the conception and design of the study. All authors read and approved the final manuscript.

Conflict of Interests The authors declare that they have no conflict of interest.

References

1. Allix K, Bissyandé TF, Klein J et al (2016) AndroZoo: collecting millions of Android apps for the research community. In: Proceedings of the 13th international conference on mining software repositories (MSR'16). ACM, New York, pp 468-471
2. Angiulli F (2005) Fast condensed nearest neighbor rule. In: Proceedings 22nd International Conference on Machine Learning (ICML'05). Association for Computing Machinery, New York, pp 25–32. https://doi.org/10.1145/1102351.1102355
3. AppBrain (2022) Android and Google Play statistics. https://www.appbrain.com/stats, last Accessed: 28 Apr 2022
4. Arp D, Spreitzenbarth M, Hübner M et al (2014) DREBIN: effective and explainable detection of android malware in your pocket. In: Symposium on network and distributed system security (NDSS). San Diego, Internet Society, pp 1–15. https://doi.org/10.14722/ndss.2014.23247
5. Berend D, Kontorovich A (2015) A finite sample analysis of the naive Bayes classifier. J Mach Learn Res 16(44):1519–1545
6. Bian Z, Vong CM, Wong PK et al (2022) Fuzzy KNN method with adaptive nearest neighbors. IEEE Trans Cybern 52(6):5380–5393. https://doi.org/10.1109/TCYB.2020.3031610
7. Cano JR, Aljohani NR, Abbasi RA et al (2017) Prototype selection to improve monotonic nearest neighbor. Eng Appl Artif Intell 60:128–135. https://doi.org/10.1016/j.engappai.2017.02.006
8. Chen T, Mao Q, Yang Y et al (2018) Tinydroid: a lightweight and efficient model for Android malware detection and classification. Mob Inf Syst 2018:9
9. Devi V, Meena L (2017) Parallel MCNN (pMCNN) with application to prototype selection on large and streaming data. Journal of Artificial Intelligence and Soft Computing Research 7:155–169. https://doi.org/10.1515/jaiscr-2017-0011
10. Dogan O, Oztaysi B (2019) Genders prediction from indoor customer paths by Levenshtein-based fuzzy kNN. Expert Syst Appl 136:42–49. https://doi.org/10.1016/j.eswa.2019.06.029
11. Flores-Velazco A, Mount DM (2020) Coresets for the nearest-neighbor rule. In: Proceedings of the 28th annual european symposium on algorithms (ESA 2020), Leibniz International Proceedings in Informatics (LIPIcs), vol 173. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp 47:1–47:19
12. Flores-Velazco A, Mount DM (2021) Boundary-sensitive approach for approximate nearest-neighbor classification. In: Proceedings of the 29th annual european symposium on algorithms (ESA 2021), pp 44:1–44:15. https://doi.org/10.4230/LIPIcs.ESA.2021.44
13. Frenklach T, Cohen D, Shabtai A et al (2021) Android malware detection via an app similarity graph. Computers & Security 109(1):1–32. https://doi.org/10.1016/j.cose.2021.102386
14. Gottlieb LA, Kontorovich A, Nisnevitch P (2016) Nearly optimal classification for semimetrics. In: Artificial Intelligence and Statistics, PMLR, pp 379–388
15. Hart PE (1968) The condensed nearest neighbor rule. IEEE Trans Inf Theory 14(5):515–516
16. Hoi SC, Sahoo D, Lu J et al (2021) Online learning: a comprehensive survey. Neurocomputing 459:249–289. https://doi.org/10.1016/j.neucom.2021.04.112
17. Hunter JD (2007) Matplotlib: a 2D graphics environment. Computing In Science & Engineering 9(3):90–95. https://doi.org/10.1109/MCSE.2007.55
18. IDC (2020) Smartphone market share. https://www.idc.com/promo/smartphone-market-share/os, last Accessed: 1 Oct 2021
19. Kontorovich A, Sabato S, Urner R (2017a) Active nearest-neighbor learning in metric spaces. J Mach Learn Res 18:1–38
20. Kontorovich A, Sabato S, Weiss R (2017b) Nearest-neighbor sample compression: efficiency, consistency, infinite dimensions. Adv Neural Inf Process Syst 30:1573–1583
21. Krauthgamer R, Lee JR (2004) Navigating Nets: simple algorithms for proximity search. In: Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04). Society for Industrial and Applied Mathematics, USA, pp 798–807
22. Kumbure MM, Luukka P (2022) A generalized fuzzy k-nearest neighbor regression model based on Minkowski distance. Granul Comput 7. https://doi.org/10.1007/s41066-021-00288
23. Liang T, Xu X, Xiao P (2017) A new image classification method based on modified condensed nearest neighbor and

convolutional neural networks. Pattern Recogn Lett 94:105–111. https://doi.org/10.1016/j.patrec.2017.05.019
24. Littlestone N, Warmuth M (1986) Relating data compression and learnability. Tech. rep., University of California Santa Cruz
25. Losing V, Hammer B, Wersing H (2018) Incremental on-line learning: a review and comparison of state of the art algorithms. Neurocomputing 275:1261–1274. https://doi.org/10.1016/j.neucom.2017.06.084
26. McKinney W (2010) Data structures for statistical computing in Python. In: Proceedings of the 9th Python in Science Conference (SciPy 2010), pp 56–61. https://doi.org/10.25080/Majora-92bf1922-00a
27. Munteanu A, Schwiegelshohn C (2017) Coresets-methods and history: a theoreticians design pattern for approximation and streaming algorithms. KI - Künstliche Intelligenz 32:37–53
28. Odusami M, Abayomi-Alli O, Misra S et al (2018) Android malware detection: a survey. In: Proceedings of the international conference on applied informatics (ICAI). Springer, Cham, pp 255–266. https://doi.org/10.1007/978-3-030-01535-0_19
29. Pedregosa F, Varoquaux G, Gramfort A et al (2011) Scikit-learn: machine learning in Python. J Mach Learn Res 12:2825–2830. ISSN 1533–7928
30. Phillips JM (2017) Coresets and sketches. In: Handbook of discrete and computational geometry. Chapman and Hall/CRC, pp 1269–1288
31. Qiu J, Zhang J, Luo W et al (2020) A survey of Android malware detection with deep neural models. ACM Comput Surv 53(6). https://doi.org/10.1145/3417978
32. Shankar VG, Somani G (2016) Anti-hijack: runtime detection of malware initiated hijacking in Android. In: Proceedings of the International Conference on Information Security & Privacy (ICISP2015), vol 78. Elsevier, Amsterdam, pp 587–594. https://doi.org/10.1016/j.procs.2016.02.105
33. Shatnawi AS, Yassen Q, Yateem A (2022) An android malware detection approach based on static feature analysis using machine learning algorithms. Procedia Computer Science 201:653–658. https://doi.org/10.1016/j.procs.2022.03.086
34. Stone CJ (1977) Consistent nonparametric regression. Ann Stat 5(4):595–620. https://doi.org/10.1214/aos/1176343886
35. Taheri R, Ghahramani M, Javidan R et al (2020) Similarity-based Android malware detection using hamming distance of static binary features. Futur Gener Comput Syst 105:230–247
36. Turi Developer Team (2022) Turi create. https://github.com/apple/turicreate, last Accessed: 28 Apr 2022
37. Virus Total (2020) VT Graph. https://www.virustotal.com/gui/graph-overview, last Accessed: 28 Apr 2022
38. Weinberger KQ, Saul LK (2009) Distance metric learning for large margin nearest neighbor classification. J Mach Learn Res 10:207–244
39. Wu DJ, Mao CH, Wei TE et al (2012) DroidMat: android malware detection through manifest and API calls tracing. In: Proceedings of the 7th asia joint conference on information security. IEEE Computer Society, Washington, pp 62–69. https://doi.org/10.1109/AsiaJCIS.2012.18
40. Yan LK, Yin H (2012) DroidScope: seamlessly reconstructing the OS and dalvik semantic views for dynamic Android malware analysis. In: Proceedings of the 21st USENIX Security Symposium. Bellevue, USENIX Association, pp 569–584
41. Zhang S, Li X, Zong M et al (2017) Learning k for KNN classification. ACM Trans Intell Syst Technol 8(3). https://doi.org/10.1145/2990508
42. Zhang X, Breitinger F, Luechinger E et al (2021) Android application forensics: a survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. Forensic Science International: Digital Investigation 39:301–285. https://doi.org/10.1016/j.fsidi.2021.301285

Dr. Jedrzej Rybicki is a post-doctoral researcher at Juelich Supercomputing Center. He received his PhD from Heinrich-Heine University of Duesseldorf. His main research interests include distributed systems, Big Data analysis tools, and Graph analysis.

Tatiana Frenklach is a Data Scientist at Illumex. She received the M.Sc. in Computer Science and B.Sc. with a specialization in Bioinformatics at Ben Gurion University. Her research focuses on malware detection on Android devices, Android static code analysis, graph theory, big data, machine learning, and deep learning. From 2015 she is a data scientist in Telekom Innovation Labs in Israel.

Dr. Rami Puzis is a senior lecturer in the Department of Software and Information Systems Engineering at Ben-Gurion University. He was a post-doctoral research associate in the Lab for Computational Cultural Dynamics, University of Maryland. His main research interests include network analysis with applications to security, social networks, computer communication, and simulations.