BOHR International Journal of Engineering
2022, Vol. 1, No. 1, pp. 42–51
DOI: 10.54646/bije.2022.09
www.bohrpub.com

ORIGINAL ARTICLE

Optimizing random forests: spark implementations of random genetic forests

Sikha Bagui* **and Timothy Bennett**

Department of Computer Science, University of West Florida, Pensacola, FL, United States

***Correspondence:**
Sikha Bagui,
bagui@uwf.edu

Received: 23 September 2022; **Accepted:** 03 October 2022; **Published:** 15 October 2022

The Random Forest (RF) algorithm, originally proposed by Breiman et al. (1), is a widely used machine learning algorithm that gains its merit from its fast learning speed as well as high classification accuracy. However, despite its widespread use, the different mechanisms at work in Breiman's RF are not yet fully understood, and there is still on-going research on several aspects of optimizing the RF algorithm, especially in the big data environment. To optimize the RF algorithm, this work builds new ensembles that optimize the random portions of the RF algorithm using genetic algorithms, yielding Random Genetic Forests (RGF), Negatively Correlated RGF (NC-RGF), and Preemptive RGF (PFS-RGF). These ensembles are compared with Breiman's classic RF algorithm in Hadoop's big data framework using Spark on a large, high-dimensional network intrusion dataset, UNSW-NB15.

Keywords: random forest, genetic algorithm, random genetic forest, bagging, big data, Hadoop, spark, machine learning, network intrusion data

Introduction

The Random Forest (RF) algorithm, originally proposed by Breiman et al. (1) and Breiman (2), is a widely used machine learning (ML) algorithm that gains its merit from its fast learning speed as well as high classification accuracy. Since RF works well with high-dimensional data (3), this robust ensemble ML algorithm is widely used with medical data (4), gene expression data (5), image processing, pattern recognition, document retrieval (6), and many other fields that inherently have high-dimensional data. More recently, the RF algorithm has also been widely used with the classification of network intrusion data (7, 8).

Random Forest utilizes a large number of decision trees to classify data and is less susceptible to overfitting. It works well with noisy data (9). However, despite its widespread use, the different mechanisms at work in Breiman's classic RF are not yet fully understood, and there is still on-going research on several aspects of optimizing the RF algorithm.

Azar et al. (4) successfully used genetic algorithms (GA) with other methods as a preprocessing step in feature selection. The optimally selected features were then used in the RF algorithm (4). Other researchers have attempted to find the best default parameters to better tune the RF algorithm (10). According to Breiman (2), the accuracy of RF depends on the strength and correlation of classifiers and the feature vectors.

This paper looks at controlling the random portions of the RF algorithm using GAs. Because GAs are frequently used in optimization schemes, the goal of this work is to fine-tune the GA so that both RF performance and accuracy are improved.

Contrasting with previous preprocessing solutions, controlling the random portions of the RF algorithm would require a new hybrid algorithm; hence, this paper proposes new ensembles that combine the optimization of GA with RF, yielding Random Genetic Forests (RGF), Negatively Correlated RGF (NC-RGF), and Preemptive RGF (PFE-RGF).

The concept behind RGF was to utilize the ability of GAs to converge on optimal configurations while still retaining some randomness. According to Breiman (2), RF handles noisy data by taking advantage of randomness with the law of large numbers. Thus, the GA implementation would need

to accommodate the desirable trait of randomness while converging on more optimal configurations.

While the classification performance of an algorithm is important, scalability is also just as important, especially in today's big data environment. Hence, in this work, in addition to creating the new hybrid algorithms or ensembles, the algorithms or ensembles have been implemented in Spark on Hadoop's Map/Reduce framework.

Hadoop (11) is a generic distributed framework that provides two features most important for big data solutions: (i) a distributed file system, Hadoop's Distributed File System (HDFS), for efficiently storing large data sets and (ii) an efficient distributed and parallel execution framework called MapReduce to process data stored in HDFS. Data stored in HDFS is divided into blocks, and each block is stored on a different cluster node, allowing for parallel processing on the nodes. Spark, which sits on top of Hadoop's distributed framework, takes advantage of Hadoop's parallel distributed framework.

The hybrid algorithms or ensembles were tested using the UNSW-NB15 (12) dataset, a modern high-dimensional network intrusion detection dataset with 2.5 million rows, with the intention of building efficient modern intrusion detection systems.

To summarize, the novelty of this work lies in building new algorithms or ensembles that optimize the random portions of the RF algorithm using GAs, yielding, RGF, NC-RGF, and PFS-RGF. These algorithms, or ensembles, are compared with the classic RF algorithm using Spark in Hadoop's big data framework using a large, high-dimensional network intrusion dataset.

The rest of the paper is organized as follows: the section that follows immediately presents the base algorithms used in the Big Data Framework, which is followed by the section that covers the related works. Thereafter, we present the section that introduces the new ensembles created, which is followed by the section that presents the fitness functions and feature metrics used. Thereafter, we present the section that details the experimentation, which is followed by the section that presents the results and conclusions. We conclude the paper with the section that presents the future work.

Background: base algorithms used and the big data framework

Random forest

The RF algorithm is an ensemble of decision tree classification or regression algorithms (9). The origin of the ensemble came from Breiman (2). The decision tree algorithm alone is prone to inaccurate predictions due to the combination of noisy data and its implementation (9).

To remedy this, Breiman (2) designed the RF ensemble, which takes advantage of concepts from statistics. Instead of relying on one decision tree, the ensemble generates multiple diverse trees using bagging (short for bootstrapping aggregation). Bagging is a form of random sampling in which finite sets of samples are selected randomly and replaced. In addition to bagging, further randomness is injected into the ensemble with random feature selection.

During decision tree construction, a random set of features is selected for splitting. From this set, the best attribute is selected for splitting using CART methodologies (2). The combination of bagging, random feature selection, and the generation of enough decision trees gives random forests the edge in accuracy.

The theory behind RF works due to the law of large numbers. The law of large numbers states: For any consistent random variable, as the number of samples grows, the average of the provided samples converges up to a limit. This limit is closer to the true mean of the random variable (13).

Thus, as the number of trees grows, the average generalization error rate converges upon its true value. Breiman (2) provides a proof for this theorem in his original paper.

The classic RF algorithm uses bagging and random feature selection as its forms of randomization. Each tree selects random features (or attributes) from the data set when generating each node. The number of features selected is usually a parameter set in the procedure, say F. A feature is selected at each node, and the samples are split.

After the ensemble is trained, the random forest can now be queried to classify samples. A vote amongst all generated trees is made to determine the classification of a particular example. The classification with the largest plurality is the resulting classification (2). The pseudocode of a classic RF algorithm is presented in **Figure 1**.

While Breiman (2) used bagging with random feature selection, this is not a requirement for the algorithm. Any type of randomization can be used to create a random forest. However, Breiman (2) provided insight on how to measure the performance of randomization on any dataset.

In order for the algorithm to provide optimum accuracy, the randomization used must do two things: increase classifier strength and decrease correlation between classifiers. Thus, datasets are important too. Different datasets will have different values of natural strength and correlation. Weaker datasets will tend to have higher generalized error rates as well.

As stated earlier, the accuracy of a random forest algorithm is based on the choice of randomness and the dataset. Breiman (2) ran many tests for multiple datasets. The tests were run against not only variations of random forest but also other algorithms.

Breiman (2) particularly focused on comparisons against Adaboost (14). While Adaboost performed well in terms of accuracy for consistent data, RF performed better than Adaboost for datasets that contained noise.

```
1 function RandomForest (training set S, feature vector F)
2       treeList[N] ← empty set
3       for i to N do
4               B ← A bootstrap sample from S
5               tree ← RandomizedDecisionTree(B, F)
6               treeList.add(tree)
7       end for
8       return treeList
9 end function
10 function RandomizedDecisionTree(sample population B, feature vector F)
11      tree ← empty set
12      If (B has same classification) return classification
13      Else if (B or F is empty) return plurality
14      Fk ← randomly select K features from F
15      A ← select best attribute using CART
16      Fa ← F - A
17      Split based on attribute A
18      for each value in A
19              subtree ← RandomizedDecisionTree(Ev,Fa)
20              tree.add(subtree)
21      return tree
20 end function
```

FIGURE 1 | Pseudocode for the classic RF algorithm (2).

Genetic algorithms

Genetic algorithms were introduced in 1973 by Holland (15). Holland (15) described genetic algorithms as adaptive solutions to optimization problems. The original journal article depicted the algorithm as a series of trials with genetic plans. Each trial would have a chance of increasing the performance of some data structure or program.

Along with the theory behind genetic algorithms, Holland (15) give theoretical lower and upper bounds on the potential optimization achievable. Thus, after so many trials, the offspring no longer produces better results, regardless of whether a global optimum is reached.

While many variants of the GA exist, most versions can be broken down into an algorithm analogous to natural selection or evolution. The basic algorithm has four phases: initialize the population, selection, crossover, and mutation.

```
1) Randomly initialize populations p
2) Determine fitness of population
3) Until convergence repeat:
    a) Select parents from population
    b) Crossover and generate new population
    c) Perform mutation on new population
    d) Calculate fitness for new population
```

FIGURE 2 | Pseudocode for the genetic algorithm.

Russell and Norvig (16) provide an excellent, simplified explanation of genetic algorithms using states of a search space for intelligent agents (16).

An initial population of encoded "genes" is generated either randomly, naturally, or by some other means. The encoding should represent a data structure or program of interest. Encodings are usually long bit strings that appear analogous to DNA.

Once the initial population is created, a subset of encodings will need to be selected. The selection process applies a fitness function to each encoding. The fittest encodings are selected for reproduction. Encodings are paired together randomly, and crossover commences.

The crossover phase mixes randomly selected "genes," or strings of characters, to create an offspring with genes from each parent. Once an offspring has been created, mutation has a chance of occurring. Mutation is a random event in which the offspring's encoding is modified in some way.

After the mutation phase, all newly created offspring encodings are added to the population to be used in the next iteration (16). **Figure 2** presents the pseudocode for the Genetic Algorithm.

The earlier discussed phases can be combined to form the basic genetic algorithm. GeeksforGeeks et al. (17) provide a relatively simple algorithm using these phases. One important step not yet mentioned is program termination. There are a couple of ways to terminate a genetic algorithm.

In **Figure 2**, the algorithm appears to not terminate until a state of convergence has been reached. Because there is an

```
function Genetic-Algorithm (population, Fitness-FN) returns an individual
inputs: population, a set of individuals
        Fitness-FN, a function that measures the fitness of an individual
repeat
        new_population <- empty set
        for i = 1 to Size(population) do
                x <- Random-Selection(population, Fitness-FN)
                y <- Random-Selection(population, Fitness-FN)
                child <- Reproduce(x,y)
                if(small random probability)then child <- Mutate(child)
                add child to new population
        population <- new_population
until some individual is fit enough, or enough time has elapsed
return the best individual in population, according to Fitness-FN
```

FIGURE 3 | Genetic algorithm with threshold.

upper bound on the achievable optimum, the fitness scores of the population will eventually converge on that limit. Once convergence has occurred, no new offspring will be better than the current best encoding (17).

While this is a desirable state, the genetic algorithm can be subjected to local maxima or minima in some cases. Alternatively, a threshold can be used. The algorithm from Russell and Norvig (16) uses such termination, as shown in **Figure 3**. Instead of waiting for convergence, a defined threshold is used to halt execution.

This allows the algorithm to select a good enough candidate instead of the best possible candidate. However, caution must be used with this method. Not all arrangements may meet the threshold. Thus, other safeguards should be in place to handle these cases. Russell and Norvig (16) use a timeout to halt execution if the threshold is never met. **Figure 3** presents the pseudocode of a GA with a threshold.

Big data

For this work, Spark was used in Hadoop's big data framework. The implementations utilized the Map/Reduce framework, which is the foundation of Hadoop/Spark. Spark's map/reduce takes advantage of the parallelization that the framework offers. In order for any ML algorithm to perform robustly, the algorithm must have the ability to scale.

Spark offers such scalability, and RF is conducive to parallelization. In this design, the RGF implementation maps its bootstrap samples, which partition one or more samples to nodes. Each node then generates one or more decision trees, which are then collected (i.e., reduced) by the Spark driver as a collection of decision trees.

The driver then maintains the collection of decision trees in the appropriate class as the actual forest. The RGF algorithm then takes further advantage of parallelization during validation. During the testing phase, RGF maps a test sample to Spark nodes. Each node then queries its collection of decision trees with each sample and reduces them to a collection of votes for a particular class.

The class votes are tallied, and the class with the most votes is the decided class for the sample. This parallelization enables the possibility of scaling for large datasets. Any system can continue to add nodes as needed to process more trees in parallel during both training, testing, and production queries; however, this is not without cost.

While more nodes tend to add a performance benefit, more nodes also mean more network overhead and communication. In addition, Spark occasionally requires data shuffles between nodes, which is costly in both network bandwidth and computation (18). Thus, there is a limit for the potential number of beneficial nodes, such that beyond that limit, additional nodes will degrade performance due to overhead.

Related works

Researchers continue to search for new versions of RF to increase accuracy. Duroux and Scornet (3) found other versions of RFs that yielded better or equivalent accuracy compared to the original algorithms. These algorithms include median and quantile random forests.

Median random forests are a variation of Breiman's original RF. The original RF uses bootstrap aggregation (bagging) with random feature selection (2). At each node during tree creation, the best feature is selected from a random set of features from the training set. Feature selection for splitting is typically based on metrics that attempt to minimize the correlation of classifiers while increasing strength.

Median forests are similar to the original random forests with the exception of random feature selection and bagging. Instead of using a traditional bootstrap sample, median forests were used. Additionally, at each node split, one random feature is selected without replacement, and the empirical median of the random feature is used to split the data (3).

Some researchers have also utilized GAs to optimize the RF-ML ensemble. Azar et al. (4) used GAs as a preprocessing optimization for feature selection. The optimization would find the best set of features to input into the RF algorithm. This optimization was used in tandem with other preprocessing methods.

Other researchers used more complex methods with GAs. Elyan and Gaber (19) used GAs with class decomposition to enhance the RF algorithm (20). Their research mostly focused on medical diagnosis datasets; however, some data from other domains was also included to prove generality (20).

From this research, the algorithm Random Forest Genetic Algorithms (RFGA) was created, which combines these methods. RFGA was compared against other competitive

Algorithm 3 Bagging-RGF

```
1 function Bagging-RGF(training set S , feature vector F)
2       treeList[N] ← empty set
3       bootstrapSamplePool[N] ← GeneticAlgorithm(initPopulation,fitnessFunction)
4       for i to N do
5               B ← bootstrapSamplePool[i]
6               tree ← RandomizedDecisionTree(B, F)
7               treeList.add(tree)
8       end for
9       return treeList
10 end function
11 function RandomizedDecisionTree(sample population B, feature vector F)
12      tree ← empty set
13      If (B has same classification) return classification
14      Else if (B or F is empty) return plurality
15      F_k ← randomly select K features from F
16      A ← select best attribute using CART
17      F_a ← F - A
18      Split based on attribute A
19      for each value in A
10              subtree ← RandomizedDecisionTree(E_v,F_a)
21              tree.add(subtree)
22      return tree
23 end function
```

FIGURE 4 | Pseudocode for bagging-random genetic forest.

algorithms such as the original random forest algorithm and AdaBoost. The RFGA generally showed favorable results.

Elyan and Gaber (19)'s motivation to use class decomposition with GAs to optimize RF is derived from two points: first, class decomposition can increase classification accuracy, and second, GAs can optimize the newly discovered subclasses and RF parameters (20). In this context, class decomposition refers to the use of clustering analysis and other techniques to decompose datasets and discover new subclasses.

Cluster analysis is the process of partitioning datasets into groups or clusters in which each observation in a cluster is similar in some way (9). For RFGA, the *k*-means clustering algorithm was used to discover new clusters (20). The justification for using class decomposition by Elyan and Gaber (19) is based on evidence of increasing use of clustering in medical datasets and the random forest algorithm.

Ensembles generated

Two ensembles were created using the RGF: the bagging-RGF and the pre-emptive RGF.

Bagging-RGF

The first ensemble attempts to control the randomness of the bootstrap samples. For each tree t in the set of generated trees T, a random bootstrap sample Bt is selected from a training set D with replacement (2). Instead of randomly selecting each bootstrap sample, a variation of the genetic algorithm is used to generate samples.

Let each sample s, in D, be uniquely indexed from 0 to $|D|$. We can encode a bit string E such that each bit indicates whether a sample is selected or not. We will then need to specify a fitness function f, such that minimizes the out-of-bag error used to validate random forests. Some challenges with this approach will need to be addressed.

The genetic algorithm converges upon either a local or global minima/maxima (15). This convergence may positively or negatively affect the accuracy of the algorithm. Additionally, finding a good fitness function may be difficult. One potential measurement is to ensure fewer duplicate samples are included.

The replacement aspect of the bootstrap sample is to allow for some duplication (2). Since bagging is completely random, there is a chance some forests may contain too many duplicate samples. Therefore, a good potential fitness

Algorithm 4 PFS-RGF

```
function PFS-RGF(training set S , feature vector F)
    treeList[N] ← empty set
    featureSetPool[N] ← GeneticAlgorithm(initPopulation, fitnessFunction)
    for i to N do
        B ← a random bootstrap sample
        tree ← RandomizedDecisionTree(B, featureSetPool[i])
        treeList.add(tree)
    end for
    return treeList
end function
function RandomizedDecisionTree(sample population B, feature vector F)
    tree ← empty set
    If (B has same classification) return classification
    Else if (B or F is empty) return plurality
    F_k ← randomly select K features from F
    A ← select best attribute using CART
    F_a ← F - A
    Split based on attribute A
    for each value in A
        subtree ← RandomizedDecisionTree(E_v,F_a)
        tree.add(subtree)
    return tree
end function
```

FIGURE 5 | Pseudocode for pre-emptive-RGF.

function is to measure the amount of duplication across the samples and use a threshold to reduce any duplication beyond the limit.

With the right configuration, this may produce consistent results and reduce out-of-bag errors caused by duplication. Another option or variant is to generate a pool of bootstrap samples and then randomly select from it with replacement, allowing for duplication. We will refer to this algorithm as bagging-RGF. Bagging-RGF is presented in **Figure 4**.

Pre-emptive feature selection-RGF

The second ensemble uses GAs to generate random trees in a RF. Let N be the number of trees generated in a RF algorithm. Any particular tree t can consist of f number of features in a feature vector F. Therefore, a tree t can be encoded as a bit string, where 1 or 0 indicates whether a feature is in the tree or not.

Allow each bit position to represent a feature, with each feature having a unique index. We can then use a GA to create a pool of feature sets for each tree. Each tree's construction will only use a selected feature subset. For this particular ensemble, we can use the CART calculations/methodology from Breiman (2) to evaluate a particular configuration.

This may reduce the number of trees required to get a decently accurate RF. Similar challenges will also be experienced with this solution. Convergence and local optima may negatively affect results. This algorithm will be referred to as "pre-emptive feature selection RGF" or "PFS-RGF."

Each proposed algorithm focuses on enhancing a particular aspect of the RF algorithm. These aspects include out-of-bag error, the number of random trees, and tree depth. While these are important, the overall goal is to retain or improve the accuracy of RF. Therefore, each algorithm will need to be tested for accuracy.

Another common problem with GAs is that they require an initial population in order to produce child configurations. Each algorithm will have to create a random set of configurations for the initial population. The cardinality of the initial population can be configured as a parameter of the algorithm, and experiments will need to be conducted to tune it to an optimal value.

Additionally, performance enhancements will be required. Instead of selecting the fittest individual, a variant of the GA could be used to create a genetic pool of encodings. We can define a minimum fitness level required to only take the fittest subset of the children.

Fitness functions and feature metrics

Since the encodings used in the GA will represent features, a fitness function will need to measure the quality of each feature. Several metrics and methods exist to score attributes. Correlation analysis is a typical measurement of quality in features (9). The correlation of two variables shows how they are related and can be determined through multiple methods. The most common are the chi-squared test and the correlation coefficient (9).

The chi-squared test is used on nominal data using frequency tables. The correlation coefficient can be calculated using the covariance of two variables divided by the product of their standard deviations (13). The correlation between all features can be calculated in a correlation matrix.

While correlation measurements are valuable, correlation does not guarantee better performance. Two variables may appear to be correlated but may not be correlated at all in reality. Thus, different metrics will be required to fully measure a feature. Principal component analysis (PCA) may be of use to measure the importance of features. PCA is a form of dimensionality reduction used to select key features or components of a dataset.

A weighted summation of correlation and inclusion of features in PCA can be used to calculate a score. Since we need to know the correlation between every feature, an average correlation score can be used. Allow F_i and F_k to be arbitrary features in a feature vector F of cardinality n, with i, $k \leq n$ and $i \neq k$. Let the average correlation score for a feature F_i be:

$$C_i = \frac{\sum_{k=0}^{n} CORR(F_i,\ F_k)}{n}\ s.t.\ k \neq i$$

where the function CORR is an arbitrary measurement of correlation. The feature's average correlation score can be used in an equation to calculate a weighted score with other measurements.

For example, if PCA is performed on a dataset, a subset of features would be merged and selected. If a feature is selected, a weight can be added, increasing the chances that the feature will be included in a genetic coding population. The full fitness function may appear as follows:

$$f(i) = w_1 C_i + w_2 p_i + g(i)\ s.t.\ p_i\ \in \{0, 1\}$$

where $w1$ and $w2$ are arbitrary weights, pi is a variable indicating whether feature F_i is chosen during PCA, and the function $g(i)$ is an arbitrary mapping that includes any other metric that may affect the final score.

Some of the challenges this project encountered included measuring correlation and/or covariance between discrete and continuous variables. Several methods exist for dealing with measuring variables of different types. Chi-squared tests measure the correlation of discrete variables (13).

Negatively correlated fitness function

Another option is to use a negatively correlated fitness function. Breiman (2) discuss how highly correlated data

TABLE 1 | Results for RF, bagging-RGF, NC-RGF, and PFS-RGF.

Averages for 30 or less trees	Recall (%)	Precision (%)	F-Measure (%)	Runtime
Algorithms				
Random forest	85.61	79.02	81.25	277.42
Random genetic forest	86.54	77.41	79.81	229.58
Negatively correlated random genetic forest	86.55	77.30	79.60	225.64
Pre-emptive random genetic forest	56.20	56.92	29.59	191.84
Averages for 30 to 60 trees				
Random forest	87.15	80.23	82.67	310.49
Random genetic forest	87.61	78.24	80.88	954.38
Negatively correlated random genetic forest	87.53	78.19	80.78	656.04
Pre-emptive random genetic forest	55.16	56.69	27.29	408.52
Averages for 100 or more trees				
Random forest	87.33	80.08	82.57	456.51
Random genetic forest	87.73	77.93	80.54	980.46
Negatively correlated random genetic forest	87.76	78.08	80.61	988.99
Pre-emptive random genetic forest	55.62	56.00	27.90	869.94
Overall averages				
Random forest	86.61	79.69	82.09	343.83
Random genetic forest	87.29	77.84	80.42	642.92
Negatively correlated random genetic forest	87.17	77.75	80.20	560.56
Pre-emptive random genetic forest	55.79	56.68	28.49	515.50

tends to perform less well when used with RF (21). Breiman (2) explain that the randomness injected in RF should minimize the mean correlation between all generated trees.

Since trees are generated by the combination of sample distribution and features, correlated features may tend to produce the same types of trees. Thus, marking highly correlated features as bad may improve the performance of the generated models. The fitness function is the trivial opposite of the previously mentioned correlation function:

$$f() = 1 - C$$

Experimentation

The data

Experimentation was done using the UNSW-NB15 (12) dataset. This dataset, published in 2015, is a hybrid of real-world network data and simulated network attacks and is comprised of 49 features and 2.5 million rows.

There are 2.2 million rows of benign traffic and 300,000 rows of attack traffic that comprise nine different modern attack categories: fuzzers, reconnaissance, shellcode, analysis, backdoors, DOS, exploits, worms, and generic. Of the attack categories, some categories had an extremely small number of attacks; for example, analysis only makes up 0.8% of the dataset, backdoor only makes up 1% of the dataset, shellcode only makes up 0.5%, and worms make up less than 0.5%.

Hence, due to the highly unbalanced nature of classes, re-sampling techniques using both up and down sampling were used to balance the data. Approximately five hundred of each attack category were used in most trials.

Data processing

The features in this dataset are made up of: five flow features like source IP, destination IP, source port number, and so on; 13 basic features like source of destination bytes, destination of source bytes, source bits per second, and so on; content features like source TCP window advertisement value, destination TCP window advertisement value, source TCP base sequence number, and so on; 10 time-related features like row jitter, destination jitter, row start time, and so on; eleven additionally generated features; and a class label.

The dataset contained both nominal and numeric data. All nominal data were encoded for training and testing. The numeric data also contained data with varying ranges. All data was scaled using a Min/Max scaler.

The experimentation

The experiments were carried out over multiple automated trials (at least over 1000 trials for each run). Each trial consisted of extracting and preprocessing training/testing datasets. Once the data were extracted, three models were generated: RF (the classic RF), bagging-RGF, NC-RGF, and PFS-RGF. The same processed training sets were used for the runs on all four algorithms.

Model configuration

Each RF (regular and genetic) is configured with the same settings per trial. Each trial can vary the many properties of RFs, including the number of trees, bootstrap samples, etc. The optimal number of trees was determined via empirical results. Automation scripts trained and tested models with a range of trees between 30 and 100. Bootstrap samples were also varied using a percentage of the total dataset.

Results and discussion

Results for the classic RF, RGF (bagging-RGF), NC-RGF, and PFS-RGF for less than 30 trees, 30–60 trees, 100 or more trees, and overall averages are presented in terms of recall, precision, F-measure, and runtime (**Table 1**).

Recall, or Attack Detection Rate (ADR), is the effectiveness of a model in identifying an attack. The objective is to target a higher ADR. The ADR is calculated by:

$$\text{Recall} = \text{TP}/(\text{TP} + \text{FN}) \quad (1)$$

Precision is the positive predictive value, or the percentage of classified attack instances that are truly classified as attacks. Precision is calculated by:

$$\text{Precision} = \text{TP}/(\text{TP} + \text{FP}) \quad (2)$$

F-measure is the harmonic mean of precision and recall. The higher the F-measure, the more robust the classification model will be. The F-measure is calculated by:

$$\text{F} - \text{measure} = 2^{*}((\text{Precision}^{*}\text{Recall})/(\text{Precision} + \text{Recall})) \quad (3)$$

As can be seen from **Table 1** and **Figures 6–9**, it is clear that for 30 or fewer trees, the RGF and NC-RGF performed slightly better than RF in terms of recall or ADR, but the precision and F-measure of RF were slightly better than the RGF and NC-RGF. PFS-RGF performed the worst in all three categories—recall, precision, and F-measure. And there was a similar trend in 30–60 trees and 100 or more trees. And these trends are also reflected in **Figures 6–8**.

The highest recall, or ADR, was obtained by the NC-RGF Forest at 87.76% for 30 or fewer trees, and the highest

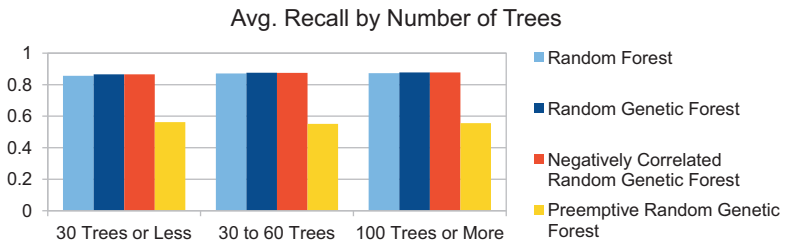

FIGURE 6 | Average recall by number of trees.

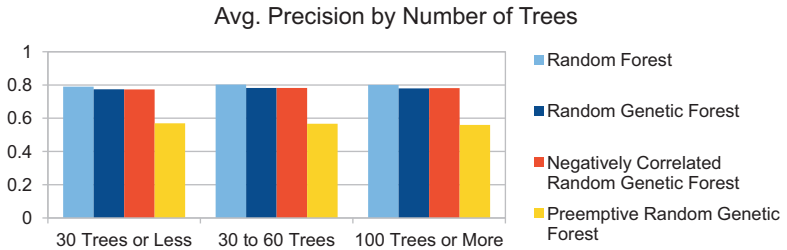

FIGURE 7 | Average precision by number of trees.

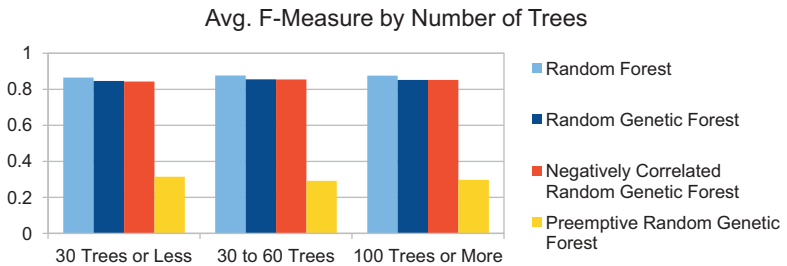

FIGURE 8 | Average measure by number of trees.

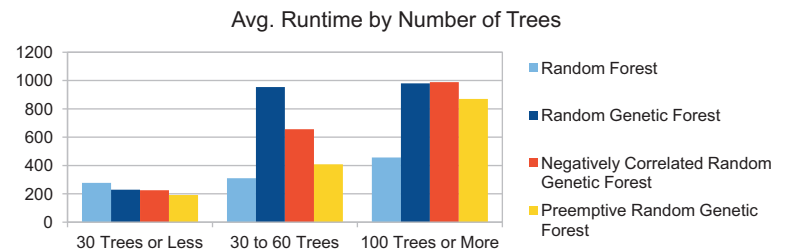

FIGURE 9 | Average runtime by number of trees.

precision and F-measure were 80.23 and 82.76%, respectively for RF for 30–60 trees.

In terms of overall averages, RGF performed the best and NC-RGF performed the second best in terms of recall, and RF performed the best in terms of precision and F-measure.

In terms of the overall runtime, RF performed the best, taking the least runtime, and RGF took the highest runtime. But for 30 or fewer trees, NC-RGF performed the best, with the lowest runtime amongst the tree algorithms (NC-RGF, RGF, and RF), and RGF performed the second best. Overall, the PFS-RGF had lower runtimes in a couple of the scenarios, and it did not perform well on the classification metrics of recall, precision, and F-measure.

Overall, from **Figures 6–9**, it can be seen that, of the four algorithms, RF, RGF, NC-RGF, and PFS-RGF, the performances of RF, RGF, and NC-RGF were very close, and RF performed well in terms of runtime in Spark's parallel big data environment.

Future work

With the results of RGF, we would like to further train RGF models with different types of fitness functions and

observe the results. Additionally, improving the runtime performance of each algorithm would be a priority for any further research. Some alternative methods could be explored. For example, a simple feature map could suffice as a fitness function.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Author contributions

SB: conceptualized the manuscript, responsible for guiding the research, and directing the formulation of the manuscript. TB did most of the implementations, presentations, analysis of the results, and wrote the first draft of the manuscript. Both authors contributed to the article and approved the submitted version.

References

1. Breiman, L., Friedman, J., Stone, C. J., and Olshen, R. A. (1984). *Classification and regression trees*. Boca Raton, FL: Chapman & Hall.
2. Breiman, L. (2001). Random forests. *Mach Learn.* 45.
3. Duroux, R., and Scornet, E. (2018). Impact of subsampling and tree depth on random forests. *ESAIM Probabil Stat.* 22, 96.
4. Azar, A. T., Elshazly, H. I., Hassanien, A. E., and Elkorany, A. M. (2014). A random forest classifier for lymph diseases. *Comput Methods Progr Biomed.* 113, 465–473.
5. Uriarte, R. D. (2006). Gene selection and classification of microarray data using random forest. *BMC Bioinform.* 7:3. doi: 10.1186/1471-2105-7-3
6. LeCun, Y., Bottou, L., and Bengio, Y. (1998). Gradient-based learning applied to document recognition. *Proc IEEE.* 86, 2278–2324.
7. Choi, S.-H., Shin, J., and Choi, M.-H. (2019). Dynamic nonparametric random forest using covariance. *Security Commun Network.* 19, 12. doi: 10.1155/2019/3984031
8. Bagui, S. S. (2021). Classifying UNSW-NB15 network traffic in the big data framework using random forest in spark. *Int J Big Data Intell Appl.* 2, 1–23. doi: 10.4018/ijbdia.287617
9. Han, J., Kamber, M., and Pei, J. (2012). *Data mining: concepts and techniques*, 3rd Edn. Waltham, MA: Elsevier Inc.
10. Scornet, E. (2017). Tuning parameters in random forests. *ESAIM Proc Surveys.* 60, doi: 10.1051/proc/201760144
11. Apache Hadoop. (n.d.). *Apache Hadoop*. Available online at: https://hadoop.apache.org/.
12. Cyber Range Lab of the Australian Centre for Cyber Security. (n.d.). Available online at: from https://www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-NB15-Datasets/ (accessed September 19, 2020).
13. Wackerly, D. D., Mendenhall, I. W., and Scheaffer, R. L. (2008). *Mathematical statistics with applications*, 7th Edn. Belmont, TN: Thompson Learning Inc.
14. Freund, Y., and Schapire, R. (1996). Experiments with a new boosting algorithm. *Mach Learn.* 148–156.
15. Holland, J. (1973). Genetic algorithms and the optimal allocation of trials. *SIAM J Comput.* 2.
16. Russell, S., and Norvig, P. (2010). *Artificial intelligence: A Moderan approach*, 3rd Edn. Upper Saddle River, NJ: Pearson Education Inc.
17. Genetic Algorithms. (n.d.). *(GeeksforGeeks*. Available online at: https://www.geeksforgeeks.org/genetic-algorithms/ (accessed May 2021)
18. Guller, M. (2015). *Big data analytics with spark*. New York, NY: Apress.
19. Elyan, E., and Gaber, M. M. (2015). A fine-grained random forests using class decomposition: an application to medical diagnosis. *Neural Comput.* 1–10.
20. Elyan, E., and Gaber, M. M. (2017). A genetic algorithm approach to optimising random forests applied to class engineered data. *Inform Sci.* 384, 220–234.
21. Breiman, L. (2002). Available online at: https://www.stat.berkeley.edu/~breiman/Using_random_forests_V3.1.pdf (accessed 06 2019).
22. Apache. (n.d.). *Cluster mode overview*. Available online at: https://spark.apache.org/docs/latest/cluster-overview.html (accessed December 2021).