Metadata of the chapter that will be visualized in SpringerLink

Book Title	Artificial Intelligence XXXV	
Series Title		
Chapter Title	Confidence in Random Forest for Performance Optimization	
Copyright Year	2018	
Copyright HolderName	Springer Nature Switzerland AG	
Corresponding Author	Family Name	**Senagi**
	Particle	
	Given Name	**Kennedy**
	Prefix	
	Suffix	
	Role	
	Division	Department of Information Technology
	Organization	Dedan Kimathi University of Technology
	Address	Nyeri, Kenya
	Division	
	Organization	LIASD, University of Paris8
	Address	Saint-Denis, France
	Email	kennedy.senagi@dkut.ac.ke
Author	Family Name	**Jouandeau**
	Particle	
	Given Name	**Nicolas**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	LIASD, University of Paris8
	Address	Saint-Denis, France
	Email	n@ai.univ-paris8.fr

Abstract

In this paper, we present a non-deterministic strategy for searching for optimal number of trees (*NoTs*) hyperparameter in Random Forest (RF). Hyperparameter tuning in Machine Learning (ML) algorithms optimizes predictability of an ML algorithm and/or improves computer resources utilization. However, hyperparameter tuning is a complex optimization task and time consuming. We set up experiments with the goal of maximizing predictability, minimizing *NoTs* and minimizing time of execution (*ToE*). Compared to the deterministic algorithm, e-greedy and default configured RF, this research's non-deterministic algorithm recorded an average percentage accuracy (*acc*) of approximately 98%, *NoTs* percentage average improvement of 29.39%, average *ToE* improvement ratio of 415.92 and an average improvement of 95% iterations. Moreover, evaluations using Jackknife Estimation showed stable and reliable results from several experiment runs of the non-deterministic strategy. The non-deterministic approach in selecting hyperparameter showed a significant *acc* and better computer resources (i.e. cpu and memory time) utilization. This approach can be adopted widely in hyperparameter tuning, and in conserving utilization of computer resources i.e. green computing.

Keywords (separated by '-')

Machine Learning - Random Forest - Hyperparameter tuning - Number of trees

Confidence in Random Forest for Performance Optimization

Kennedy Senagi[1,2(✉)] and Nicolas Jouandeau[2]

[1] Department of Information Technology, Dedan Kimathi University of Technology, Nyeri, Kenya
kennedy.senagi@dkut.ac.ke
[2] LIASD, University of Paris8, Saint-Denis, France
n@ai.univ-paris8.fr

Abstract. In this paper, we present a non-deterministic strategy for searching for optimal number of trees ($NoTs$) hyperparameter in Random Forest (RF). Hyperparameter tuning in Machine Learning (ML) algorithms optimizes predictability of an ML algorithm and/or improves computer resources utilization. However, hyperparameter tuning is a complex optimization task and time consuming. We set up experiments with the goal of maximizing predictability, minimizing $NoTs$ and minimizing time of execution (ToE). Compared to the deterministic algorithm, e-greedy and default configured RF, this research's non-deterministic algorithm recorded an average percentage accuracy (acc) of approximately 98%, $NoTs$ percentage average improvement of 29.39%, average ToE improvement ratio of 415.92 and an average improvement of 95% iterations. Moreover, evaluations using Jackknife Estimation showed stable and reliable results from several experiment runs of the non-deterministic strategy. The non-deterministic approach in selecting hyperparameter showed a significant acc and better computer resources (i.e. cpu and memory time) utilization. This approach can be adopted widely in hyperparameter tuning, and in conserving utilization of computer resources i.e. green computing.

Keywords: Machine Learning · Random Forest
Hyperparameter tuning · Number of trees

1 Introduction

RF was first introduced by Breiman [2]. It is an ensemble classifier that builds many decision trees from the same dataset using bootstrapping and randomly sampled variables. Predictions at the leaves are combined to form a single prediction; for each tree. When performing classifications, the input query instances traverse each tree, which casts its vote for a class. RF considers the class with the most votes as the answer to a classification query [2]. The inception of RF has led to development of many RF libraries and diverse usage of RF on a variety

M. Bramer and M. Petridis (Eds.): SGAI-AI 2018, LNAI 11311, pp. 1–15, 2018.
https://doi.org/10.1007/978-3-030-04191-5_31

of datasets. RF has been implemented in several software libraries: scikit-learn, SPRINT and Random Jungle. It has been studies and applied in diverse domain [11,12,15].

RF is a supervised ML algorithm that requires tuning to improve predictability, speed and utilization of computer resources. Tuning RF involves adjusting hyperprameters. Hyperparameters can be set by default or configured manually. In default settings, the ML algorithm sets hyperprameter values while manually setting requires users to set specific values. Hyperparameters specify interoperability of the underlying model. When adopting ML algorithm to specific datasets, hyperparameter tuning can be cumbersome and time consuming [6,14].

Default hyperprameter values do not give better results compared to tuned RF. Many theories have been put forward to optimize hyperparameter RF including grid search and Bayesian optimization [14]. Some researchers focus on optimizing specific hyperprameters [1,10,11]. For instance, Bernard et al. [1] discusses ways of selecting the number of features to consider in node splitting and Liu et al. [10] derived a nonparametric algorithm to estimate the categorical distributions of the internal nodes. $NoTs$ has been studied too [11].

This research focused on maximizing *acc* while minimizing $NoTs$ and ToE in RF. We did experiments on different datasets and saw that most datasets had maximum *acc* and minimum ToE between 2 to 512 $NoTs$, the *fertile region*. We formulated a non-deterministic algorithm that maximized *acc* while minimizing $NoTs$ and ToE in the fertile region. Moreover, we did further experiments using grid search, e-greedy algorithms, and compared results.

In this paper, Sect. 2 covers related works, Sect. 3 discusses experiments, results and analysis and Sect. 4 concludes the paper.

2 Related Works

Tuning ML systems can be time consuming and at times inaccurate and difficult. To solve this, a hyperparameter optimization strategy is proposed inspired by analysis of boolean function focusing on high-dimension datasets. The algorithm recorded at least an order of magnitude of speedup than Hyperband and Bayesian Optimization and outperform Random Search 8x [7].

Sensitive was more on $NoTs$ but less sensitive to the number of features in node split. MapReduce could made parameter optimization feasible on a massive scale [6]. RF grows trees rapidly and setting up a large $NoTs$ (e.g. 1000) is okay. If there are many variables, trees can grow more (up-to 5000) [3]. *acc* increased when $NoTs$ in RF was doubled. However, there was a threshold beyond which there was no significance gain in *acc* [11].

A full Bayesian treatment expected improvement parameter tuning, and algorithms (e.g. ANN) for dealing with variable time regimes and running experiments in parallel. Results of this experiment surpassed a human expert at selecting hyper-parameters on the competitive CIFAR-10 dataset; beating the state of the art by over 3%. SVM was used as a case study algorithm [14].

XGBoost algorithm proposed candidate splitting points according to percentiles of feature distribution, then maps the continuous features into buckets

split, aggregates the statistics and finds the best solution among proposals based on the aggregated statistics [4].

Optimizing parameters of evolutionary algorithm values is a challenging activity. CMA-ES tuning algorithms gave better results in terms of utility. Tuning parameters of evolutionary algorithms does pay off in terms of performance, better other intuitions and usual parameter setting conventions [13].

A selection of supplemental training datasets was used in fine-tuning a high-performing ANN model. Natural Language Processing system ability is improved after being evaluated by Item Response [9].

3 Experiments, Results and Analysis

In this research, we considered 20 standardized datasets collected from UCI Machine Learning [5] and Kaggle [8] website, namely: Balance Scale (1), Breast Cancer Wisconsin - Original (2), Car Evaluation (3), Habermans Survival (4), Pen-Based Recognition of Handwritten Digits (5), Website Phishing (6), Yeast (7), Banknote Authentication (8), Contraceptive Method Choice (9), Diabetic Retinopathy Debrecen (10), EEG Eye State (11), Pima Indians Diabetes (12), Wine Quality - White (13), Wine Quality (14), Breast Cancer Wisconsin (Original) (15), Dota (16) Handwriting Verification Test (17), Ionosphere (18), Plant Leaf Classification (19) and Seeds (20). All experiments were run 20 times and results averaged. $NoTs$ (θ) was varied, as we measured accuracy (acc) and time of execution (t). Computer was: Intel(R) Xeon(R) CPU E5-4610 0 @ 2.40 GHz.

3.1 Considering 2 to 4096 Number of Trees

The parameter space was composed of a finite set of sorted even $NoTs$; 2 to 4096. RF predictability was evaluated by acc defined in Eq. 1; where n are samples, $\hat{y}_i$ is the predicted label and y_i is the original label. The results of acc and t are tabulated in Tables 1 and 2 respectively.

$$acc(y, \hat{y}) = \frac{1}{n} \sum_{i=0}^{n-1} 1(\hat{y}_i = y_i) \tag{1}$$

Table 1 shows acc increasing steadily with an increase in $NoTs$, then flattens. RF classification employs bagging principles, where a committee of trees, cast a vote for the predicted class. However, RF classifier introduces modifications in bagging where it builds a large collection of de-correlated trees, and then averages them. When the $NoTs$ become huge, we see RF acc varying insignificantly meaning the average acc of de-correlated trees vary insignificantly. Therefore, increasing the $NoTs$ increases acc, but there is a threshold where, increasing $NoTs$ does not contribute to a significantly positive acc. Maximum acc values are bolden in Table 1. Besides the 6^{th} dataset having its average maximum acc at 2048 $NoTs$, the other datasets had their average accuracies between 2 and

Table 1. Accuracy (percentage) of RF with θ trees for 20 datasets (DS)

DS	Number of trees											
	2	4	8	16	32	64	128	256	512	1024	2048	4096
1	80.3	81.9	83.0	82.4	84.6	**85.6**	84.6	84.0	84.0	84.0	84.6	84.6
2	91.7	93.7	97.1	**98.0**	97.6	97.6	97.6	97.1	97.1	97.1	97.1	97.1
3	86.3	85.5	83.6	83.8	**84.8**	84.4	84.6	84.4	84.8	84.8	84.6	84.6
4	76.1	79.3	75.0	76.1	**79.3**	79.3	78.3	78.3	78.3	79.3	78.3	79.3
5	92.5	96.8	98.3	98.6	98.4	98.9	99.0	**99.1**	99.0	99.1	99.1	99.1
6	81.5	86.9	86.2	87.4	85.7	87.4	87.9	88.4	87.7	87.9	**88.7**	88.2
7	48.6	47.8	52.9	57.3	56.5	59.5	**59.8**	58.8	58.8	58.5	58.5	58.8
8	96.6	**97.8**	97.6	97.6	97.3	97.6	97.8	97.8	98.1	97.8	97.8	97.8
9	46.4	48.4	49.1	**51.6**	49.5	49.8	51.1	49.5	50.7	51.4	50.9	51.1
10	61.3	64.7	65.3	65.0	**69.9**	66.5	67.6	67.9	68.2	67.1	67.9	67.3
11	77.9	84.2	87.9	89.3	91.3	**92.7**	92.0	92.2	92.2	92.1	92.3	92.2
12	66.7	71.0	74.9	74.5	76.6	76.6	76.6	75.8	**77.5**	76.6	77.1	77.1
13	54.9	59.4	64.7	64.6	65.7	65.9	67.1	**67.3**	67.1	66.6	67.3	67.4
14	54.4	69.7	63.3	67.3	69.2	69.2	69.6	**70.2**	69.8	69.2	69.8	69.8
15	96.5	96.6	97.0	**97.2**	97.2	97.2	97.2	97.2	97.2	97.2	97.2	97.2
16	73.6	77.1	79.9	83.8	84.9	**86.0**	86.6	86.8	86.2	87.1	87.5	87.1
17	84.9	88.7	91.5	89.6	91.5	**92.5**	92.5	91.5	92.5	92.5	92.5	92.5
18	74.9	70.1	75.8	77.5	76.2	76.2	75.8	**76.6**	77.5	77.5	77.5	76.6
19	92.5	93.2	93.2	93.8	93.8	93.7	93.6	**94.1**	93.8	94.1	94.1	94.1
20	82.5	**90.5**	87.3	84.1	87.3	85.7	87.3	87.3	87.3	87.3	87.3	87.3

512 trees. These variations could have been caused by the randomness of RF. This research identified the range of 2 to 512 $NoTs$ to be the *fertile region*.

Table 2 shows ToE increasing steadily with an increase in $NoTs$. This tells us that building more $NoTs$ in RF demand more computing resources. We also observed a relative significant change in ToE; the threshold values are in bold.

Furthermore, this research notes that, different datasets gave different values of acc and ToE with the different $NoTs$. This is could be as a result of different dataset having different complexities (i.e. features, number of records and classes) and the random nature of RF. Therefore, it is important we maximize acc and minimize ToE (i.e. minimize computer resources utilized) in the fertile region.

3.2 Considering 2 to 512 Number of Trees

In Sect. 3.1, we defined the fertile region where we observed a lower ToE and maximum accuracies. In this region, we can avoid searching in regions (>512)

Table 2. Time of execution (sec) of RF with θ trees for 20 datasets (DS)

DS	Number of trees											
	2	4	8	16	32	64	128	256	512	1024	2048	4096
1	0.21	0.21	0.22	0.23	0.25	**0.30**	0.51	0.90	1.60	3.29	6.49	12.45
2	0.21	0.21	0.22	0.23	0.25	**0.30**	0.50	0.90	1.59	3.09	5.98	12.35
3	0.21	0.21	0.22	0.23	0.25	**0.30**	0.50	1.00	1.80	3.39	6.79	13.57
4	0.21	0.21	0.22	0.23	0.25	**0.30**	0.50	0.80	1.60	3.30	5.99	12.06
5	0.21	0.21	0.22	0.23	**0.26**	0.41	0.60	1.10	2.20	4.01	8.23	15.87
6	0.21	0.21	0.22	0.23	0.25	**0.30**	0.50	0.89	1.89	3.46	6.42	13.14
7	0.21	0.21	0.22	0.23	0.26	**0.30**	0.50	1.00	1.88	3.71	7.13	14.06
8	0.21	0.21	0.22	0.23	0.25	**0.30**	0.50	0.90	1.69	3.17	6.64	12.76
9	0.21	0.21	0.22	0.23	0.25	**0.30**	0.50	1.00	1.89	3.47	6.95	13.70
10	0.21	0.21	0.22	0.23	0.25	**0.30**	0.50	0.90	1.79	3.47	6.93	14.41
11	0.21	0.21	0.22	0.33	**0.46**	0.71	1.20	2.40	4.70	9.18	18.27	36.62
12	0.21	0.21	0.22	0.24	0.25	**0.30**	0.50	0.79	1.68	3.46	6.43	12.64
13	0.21	0.21	0.22	0.23	**0.25**	0.51	0.70	1.40	2.69	5.28	10.45	21.20
14	0.21	0.21	0.22	0.23	0.25	0.40	**0.60**	1.10	2.09	3.76	7.33	14.96
15	0.26	0.32	0.26	0.31	0.50	**0.62**	1.19	1.93	3.73	6.74	14.13	27.27
16	0.26	0.28	0.29	0.31	**0.47**	0.77	1.03	1.94	3.11	6.54	12.95	26.09
17	0.26	0.29	0.29	0.31	**0.37**	0.57	0.95	1.71	2.94	5.07	10.21	19.48
18	0.24	0.26	0.29	0.34	**0.35**	0.62	0.90	1.60	2.60	5.31	10.51	19.33
19	0.25	0.26	0.29	0.33	**0.31**	0.71	0.89	1.50	3.00	5.49	10.43	20.885
20	0.26	0.26	0.29	0.37	**0.37**	0.55	1.13	1.78	3.033	5.59	11.128	20.389

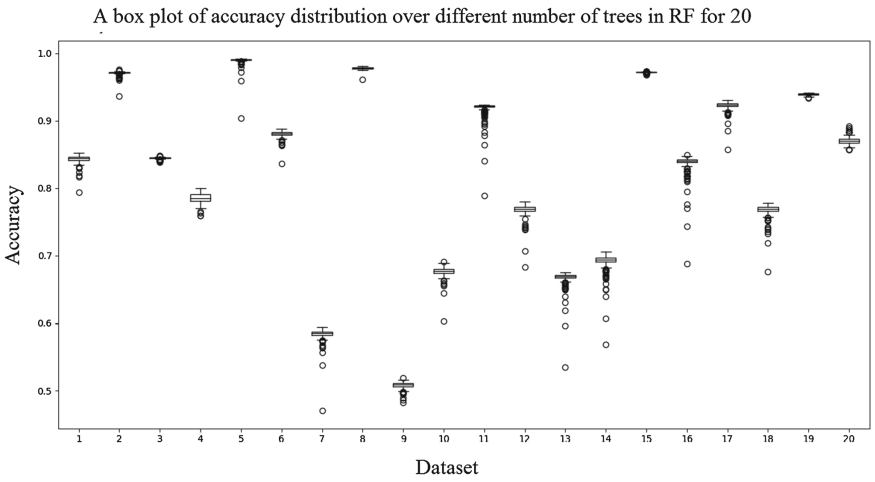

Fig. 1. Number of trees against datasets of accuracy in RF for 20 datasets

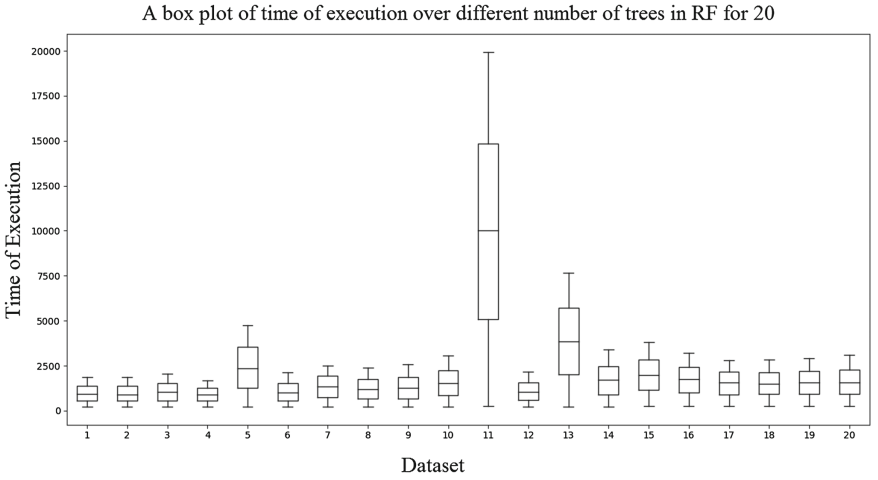

Fig. 2. Number of trees against datasets of time of execution in RF for 20 datasets

that showed higher ToE and significant flattening of acc. Within the fertile region parameter space, we defined a finite set of sorted even $NoTs$, θ. We configured, trained and tested RF with the respective θ and recorded acc and t; results are shown in Figs. 1 and 2.

Figure 1 shows a box plot of accuracy distribution for $NoTs$ against datasets across 20 datasets in the fertile region. Some datasets had a low inter-quartile range, low difference between the low and maximum points and more outliers below the lower whiskers. Some box plots also recorded some outliers above the upper whisker. A low difference in quartile ranges means there was a low variation in acc from the median. However, the outliers inform us that, some accuracies values were very far away from the median. Nonetheless, we see different acc distribution on different datasets. We therefore need to have a strategy that will dynamically search the best acc.

Figure 2 is a box plot of $ToEs$ distribution for $NoTs$ against datasets across 20 datasets in the fertile region. We observed the lower whisker having almost the same ToE; there could be some $NoTs$ that give almost the same minimum ToE. In most datasets, the lower whiskers being shorter than the upper whiskers. A shorter lower whisker means lower ToE were closer to the median.

From the above, we formulated: deterministic, non-deterministic, e-greedy and default configured RF (having 8 $NoTs$) algorithmic approaches to search an minimum $NoTs$ hyperparameter that maximized acc.

(a) Deterministic Hyperparameter Search

The deterministic search algorithm is outlined in Algorithm 1. This algorithm's goal was to maximize acc. Note that, $\exists acc_{max} \in acc$. The deterministic algorithm is greedy and exhaustive (linear search) and returns acc_{max}, with its corresponding NoT (θ_{best}) and ToE (t). Results are in Tables 4, 5 and 6.

Algorithm 1. The Deterministic Hyperparameter Search

```
procedure DeterministicSearch(train, test)
    t_i ← CurrentTime()
    T ← [θ_1, θ_2, θ_3, ..., θ_n]
    acc_max ← 0
    for each θ in T do
        rf ← RandomForest(θ, train)
        acc_new ← Accuracy(rf, test)
        if acc_new > acc_max then
            (acc_max, θ_best) ← (acc_new, θ)
    time_spent ← CurrentTime() − t_i
    return (acc_max, θ_best, time_spent)
```

Table 3. Percentage of samples and average standard deviation across 20 datasets

DS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Percentage	90	80	30	10	20	30	40	20	50	50	10	10	20	20	10	30	30
Average STD $\times 10^{-4}$	1.1	0	1.1	3.9	0.5	1.1	1.4	0.5	0	1	1	3.9	2.6	1.2	0.5	2.4	1.1
Modal percentage	10																

(b) The Non-deterministic Hyperparameter Search Algorithm

This research was interested in maximizing *acc* and minimizing $NoTs$. Tables 1 and 2 shows different *acc* with different $ToEs$. Table 2 shows more $NoTs$ require more ToE (i.e. memory and cpu time). With this observation, we ought to maximize *acc* and minimize $NoTs$ to conserve computing resources i.e. green computing. This research therefore formulated a non-deterministic approach to converge close/to maximum *acc* and minimum $NoTs$. The algorithm is outlined in Algorithm 2, where $\theta_i = random(\in \mathcal{T}), \psi_1 = 1 + \frac{lim}{100}$, *and* $\psi_2 = 1 - \frac{lim}{100}$.

Experiment results of the non-deterministic algorithm are in Table 3. We see a significantly low average standard deviation across the percentage samples. This means there was a small variation in *acc* across the percentage samples. We considered 10% sample because it requires lesser CPU resources in generating random $NoTs$, using the function, Generate( ) . Fewer $NoTs$ means lower ToE.

The goal of the non-deterministic algorithm was to maximize *acc* and minimize *t* through randomization. In this algorithm we assumption that, $\exists acc_{best} \in acc$ has θ_{best}. Note that the function Generate( ) returns 10% of elements in the parameter space and elements are sampled without replacement. We iterate through the random selected $NoTs$ as we configure RF. We considered percentage upper bound and lower bound of the acc_{best}. If acc_{rand} falls in the upper boundary, then $acc_{best} \leftarrow acc_{rand}, \theta_{best} \leftarrow \theta_{rand}$ and we *break*, with the assumption that we do not anticipate further percentage Δ acc_{best}. If acc_{rand} falls in the lower boundary and θ_{rand} is less than θ_{best}, then $acc_{best} \leftarrow acc_{rand}, \theta_{best} \leftarrow \theta_{rand}$ and we also *break*, with the assumption that we have an insignificant Δ acc_{best} and we have a better t_{best}. Moreover, if acc_{rand} falls above the upper boundary, then $acc_{best} \leftarrow acc_{rand}, \theta_{best} \leftarrow \theta_{rand}$, and we continue looping with the

Algorithm 2. The Non Deterministic Hyperparameter Search

```
vals = []
procedure GENERATE()
    while LEN(vals) ≤ 26 do
        val = 2 + rand()%512
        if val is not in vals then
            add val in vals
    return val
procedure NONDETERMINISTICSEARCH(train, test)
    t_i ← CURRENTTIME()
    acc_rand, θ_rand, acc_best, θ_best, count ← 0
    T ← GENERATE( )
    for each θ_rand in T do
        rf ← RANDOMFOREST(θ_rand, train)
        acc_rand ← ACCURACY(rf, test)
        if count == 0 then
            (acc_best, θ_best) ← (acc_rand, θ_rand)
        if ψ1.acc_best > acc_rand > ψ2.acc_best then
            if acc_rand < acc_best then
                if θ_rand < θ_best then
                    (acc_best, θ_best) ← (acc_rand, θ_rand)
                    break
            else
                (acc_best, θ_best) ← (acc_rand, θ_rand)
                break
        else if acc_rand > ψ1.acc_best then
            (acc_best, θ_best, count) ← (acc_rand, θ_rand, 0)
        count ← count + 1
        if count >= 10 then break
    time_spent ← CURRENTTIME() − t_i
    return (acc_best, θ_best, time_spent)
```

assumption that we anticipate further percentage Δ acc_{best}. Lastly, we *break* when iteration counts are 10% of the parameter space. Finally, we set the percentage boundary as 1% to increase the algorithm's *acc*. A larger percentage boundary will reduce the algorithm's *acc*. Results are in Tables 4, 5 and 6.

(c) RF configured with Default Parameters

RF took 8 number of trees by default as the NoT.

(d) The e-greedy Algorithm

A multi-bandit strategy is a mathematical model used to reason about how to make a decision when we have many actions to take and imperfect information about the rewards you would receive after taking these actions. Multi-armed bandit problem use the analogy of considering arms (options) to select in order

to maximize a reward. e-greedy is an example of a multi-bandit algorithm. e-greedy uses the concept of exploiting and exploring. This algorithm is greedy in the sense that it exploits the best action at that time. However, the exploiting is regulated by the epsilon value that allows it to explore (trying out other options). We calculate the probability that an event will occur and match it with the epsilon value ϵ, set to 0.1, this guides us on when to exploit or explore. Exploiting considers the rewards for each arm/option and picks the arm with the highest reward, while exploring randomly chooses any arm. The arm/option that was either exploited or explored accumulates an reward [16]. This research looked at the problem of minimizing NoT using multi-armed bandit problem, whereby a user anticipates to chose an NoT that maximizes *acc* (exploiting) and can also considering other possible $NoTs$ could maximize *acc* (exploring). We initialize a counter and rewards arrays with lengths equivalent to $NoTs$ length. The chosen NoT increase it's count value, in it's index in the counter array c. Cumulative occurrences is got by summing up the counter array, C. These values are used in calculating the probability of an event will occur p, where $p = c/C$. If p is less than ϵ we choose to exploit, otherwise we explore. In exploring, we choose a random NoT from the other $NoTs$. After either exploiting or exploring, each NoT reward (in this case *acc*) is averaged in it's element index in the rewards array. This rewards array was used in considering future arms to exploit. We set-up experiments with 500 iterations.

3.3 Performance Comparisons: Number of Trees, Accuracy and Time of Execution

Table 4 contains results and analysis of minimum $NoTs$, selected by deterministic and non-deterministic hyperparameter search algorithms. The table also has average probabilities for e-greedy predicting θ_{best}. We observe a considerably good percentage improvement of $NoTs$ in the non-deterministic algorithm. At some instances, for example, in datasets 8 and 13, the non-deterministic algorithm was able to perfectly converged to the minimum $NoTs$ with 26 and 2 iterations respectively. Moreover, as observed in Table 4, about 65% of the datasets used less than 50% (i.e. less than 5% of random values in the search space) of random values while iterating, to converge close/to maximum *acc* and minimum $NoTs$. Generally, the percentage $NoTs$ improvement was 29.39% and the average number of iterations used were 11.8. We have average probabilities of e-greedy selecting the correct θ_{best} for each dataset. We were not able to get a good visualization of e-greedy learning performance (in terms of probabilities) across 500 iterations for the 20 datasets. However, we averaged the probabilities and we see e-greedy had an average 0.83 i.e. e-greedy can select θ_{best} quite well.

Table 5 shows accuracy recorded from running deterministic, non-deterministic and default configured RF. The default configured RF had a mean percentage difference of -4.9 while the non-deterministic and e-greedy algorithms had the same and considerably better percentage change of -1.81. In non-deterministic algorithm, datasets 2, 8 and 13 recorded a zero percentage change in *acc*.

Table 4. Recorded minimum number of trees (θ_{best}) and iterations for deterministic and non-deterministic algorithms, and e-greedy average probability of predicting θ_{best}

DS	Deterministic	Non-deterministic			e-greedy
	θ_{best}	θ_{best}	θ % improvement	Iteration	Avg probability of selecting θ_{best}
1	26	32	−23.08	5	0.821
2	46	48	−4.35	26	0.703
3	116	46	60.34	26	0.958
4	70	18	74.29	26	0.949
5	48	16	66.67	26	0.821
6	216	26	87.96	26	0.774
7	118	34	71.19	3	0.819
8	44	44	0	26	0.778
9	48	42	12.5	2	0.825
10	18	10	44.44	4	0.821
11	196	50	74.49	26	0.723
12	164	10	93.9	2	0.822
13	46	46	0	2	0.82
14	150	50	66.67	3	0.813
15	32	32	−1.133	11	0.808
16	500	46	−30.250	2	0.785
17	138	20	−7.118	3	0.761
18	26	26	−0.444	3	0.78
19	2	2	0.895	11	0.913
20	4	10	0.800	2	0.787
μ	100	30.40	29.39	11.8	0.814

Table 6 has results and analysis of time of execution of deterministic, non-deterministic, e-greedy algorithms and default configured RF. The ratio of deterministic: non-deterministic algorithms, deterministic:e-greedy, deterministic: default configured RF are calculated. Their averages are also calculated. Default-configured RF records a very high average ratio of 6223. Non-deterministic and e-greedy algorithms record relatively high ratios of 415 and 110 respectively. As discussed in this section, the deterministic algorithm is exhaustive and selects the minimum $NoTs$ that had the maximum acc. With these results, we benchmark the non-deterministic algorithm, e-greedy and default configured RF. The non-deterministic algorithm uses the principle of randomization, heuristics and terminating policies as outlined in Algorithm 2. With this strategy, the non-deterministic algorithm recorded ≈97% average acc, and could run at an average of 415.92 faster, on an average of 11.8 iterations. Using the strategy formulated in Algorithm 2, the non-deterministic algorithm recorded 100% acc at three instances, and recorded zero $NoTs$ percentage improvement on two instances. Moreover, in the non-deterministic algorithm, we recorded $NoTs$ that are below the $NoTs$ threshold (64 trees), that showed a significant

Table 5. Maximum accuracy (acc_{best}) for deterministic, default configured RF and non-deterministic, and average *acc* for e-greedy recorded across 20 datasets (DS)

DS	Deterministic	Default configured RF		Non-deterministic		e-greedy	
	acc_{max}	acc_{best}	% Δ	acc_{best}	% Δ	*Avg acc*	% Δ
1	0.862	0.819	−4.99	0.856	−0.70	0.847	−1.74
2	0.976	0.971	−0.51	0.976	0.00	0.971	−0.51
3	0.85	0.846	−0.47	0.846	−0.47	0.849	−0.12
4	0.815	0.761	−6.63	0.804	−1.35	0.8	−1.84
5	0.993	0.973	−2.01	0.99	−0.30	0.988	−0.50
6	0.897	0.855	−4.68	0.887	−1.11	0.883	−1.56
7	0.601	0.552	−8.15	0.593	−1.33	0.588	−2.16
8	0.985	0.976	−0.91	0.985	0.00	0.98	−0.51
9	0.538	0.48	−10.78	0.505	−6.13	0.514	−4.46
10	0.711	0.627	−11.81	0.682	−4.08	0.685	−3.66
11	0.925	0.89	−3.78	0.919	−0.65	0.91	−1.62
12	0.797	0.74	−7.15	0.736	−7.65	0.775	−2.76
13	0.681	0.636	−6.61	0.681	0.00	0.669	−1.76
14	0.71	0.654	−7.89	0.679	−4.37	0.692	−2.54
15	0.973	0.97	−0.31	0.973	0.00	0.972	−0.10
16	0.88	0.826	−6.14	0.863	−1.93	0.863	−1.93
17	0.942	0.909	−3.50	0.933	−0.96	0.925	−1.80
18	0.797	0.735	−7.78	0.771	−3.26	0.773	−3.01
19	0.942	0.936	−0.64	0.94	−0.21	0.94	−0.21
20	0.919	0.883	−3.92	0.903	−1.74	0.887	−3.48
μ	0.840	0.802	−4.933	0.826	−1.812	0.826	−1.814

change in $ToEs$, as discussed in Sect. 3.1. This means the formulated strategy worked quite well. Considering dataset 2, we note that 0% percentage *acc* change, was got with more $NoTs$ (48 trees instead of 46 trees) but at 34.76 times faster. These shows 100% accuracies got, at more number trees but takes a shorter searching time. This makes the strategy formulated in this research relevant. Despite the 1% boundary policy and breaking policies strategies, 65% of the datasets recorded less than 1% change in percentage *acc*. The other 35% scored fairly good results too. Generally, a shorter ToE means the process will take a shorter time in memory and shorter cpu time, when tuning RF. We also observed the non-deterministic algorithm run an average of 4.6% iterations (i.e. 11.8 of 256 iterations in the parameter space). This is an improvement in iterations by 95.3%. Therefore, the non-deterministic algorithm can improve utilization of computing resources while maintaining a significant *acc*.

Table 6. Average time of execution (sec) recorded across 20 datasets (DS)

DS	Deterministic	Default configured RF		Non-deterministic		e-greedy	
	i (sec)	t (sec)	Ratio	t (sec)	Ratio	t (sec)	Ratio
1	224.11	0.03	7470	1.22	183.7	2.16	103.75
2	217.97	0.02	10899	6.27	34.76	7.16	100.91
3	239.22	0.03	7974	6.45	37.09	7.28	104.92
4	216.26	0.02	10813	6.43	33.63	7.16	100.12
5	282.42	0.07	4035	6.38	44.27	7.59	118.17
6	235.94	0.03	7865	6.25	37.75	7.96	104.4
7	249.68	0.04	6242	0.78	320.1	1.33	107.16
8	230.44	0.03	7681	6.34	36.35	7.9	104.75
9	246.37	0.03	8212	0.51	483.08	2.32	106.19
10	246.37	0.04	6159	0.94	262.1	2.26	109.01
11	622.88	0.29	2148	1.02	610.67	2.76	225.68
12	227.91	0.03	7597	0.46	495.46	2.25	101.29
13	360.73	0.11	3279	0.59	611.41	2.46	146.64
14	260.52	0.05	5210	0.77	338.34	2.31	112.78
15	230.42	0.15	1536	0.26	886.23	2.37	97.22
16	218.4	0.13	1680	0.29	753.1	2.26	96.64
17	195.28	0.03	6509	0.25	781.12	2.16	90.41
18	213.35	0.03	7112	0.28	761.96	2.25	94.82
19	227.81	0.04	5695	0.28	813.61	2.3	99.05
20	190.49	0.03	6350	0.24	793.71	2.16	88.19
μ	256.83	0.06	6223.3	2.30	415.92	3.82	110.61

As explained in this section, e-greedy algorithm heavily relies on the idea that, the probability of the chosen NoT should be less than the ϵ value, for it to exploit that (selected) arm. Otherwise, the algorithm explores (randomly selects other arms). Across the 20 datasets, on average, the probability that e-greedy will select the optimal arm is 0.814, as shown in Table 4. At some point, because of the exploring and exploiting ideas, e-greedy misses to select the best arms, that's why Table 5 shows e-greedy having a average acc score of 0.826. Nevertheless, in the same table, considering the deterministic algorithm selected the best acc value, e-greedy falls short of an averagely of -1.8% from the best score, and runs 110.61 faster than the deterministic algorithm. Comparing e-greedy and the non-deterministic, coincidentally, we see both of them having the same accuracy across the 20 datasets. However, the non-deterministic algorithm runs 415 times faster compared to the deterministic approach while e-greedy that run at 110 times faster than the deterministic approach. Generally we see the non-deterministic algorithm performing better than the e-greedy algorithm.

The RF algorithm configured by default $NoTs$ (8 trees) showed good results too. It recorded ≈94.5% average accuracy change and very good ToE ratio of 6223. Table 2 told us that fewer $NoTs$ give lesser times of execution. Comparing these results with the non-deterministic algorithm, Table 6 has a higher $ToEs$ because of the higher number of iterations (average of 30.40 across the 20 datasets). If we benchmark our results with the deterministic algorithm, RF algorithm configured by default $NoTs$ runs 6223 faster and give 94.5% chances to get the average best acc, on average. While the non-deterministic approach runs 415 faster and give 98.19% chances to get the average best acc, on average.

3.4 Evaluation Using Jackknife Estimation

Jackknife is used to evaluate the quality of the prediction of computational models. It uses resampling to calculate standard deviation error and estimate bias of a sample statistic, as shown in Eqs. 2 and 3 [15]. Table 7 shows Jackknife results across the 20 datasets. Different datasets record different values of Bias-Corrected Jackknifed Estimates. Standard error is used for null hypothesis testing and for computing confidence intervals. This is why confidence intervals deviating insignificantly. We also see the bias-corrected Jackknifed estimate

Table 7. Jackknife estimates for deterministic and non-deterministic algorithms

DS	Bias-corrected jackknifed estimate			Confidence interval					
				Deterministic		Non-deterministic		e-greedy	
	Deterministic	Non-deterministic	e-greedy	Lower	Upper	Lower	Upper	Lower	Upper
1	0.86	0.85	0.84	0.86	0.87	0.85	0.85	0.84	0.85
2	0.98	0.98	0.97	0.98	0.98	0.98	0.98	0.97	0.97
3	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85
4	0.82	0.79	0.8	0.82	0.82	0.79	0.80	0.79	0.8
5	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99
6	0.89	0.88	0.88	0.89	0.89	0.88	0.89	0.88	0.88
7	0.61	0.59	0.59	0.60	0.61	0.59	0.59	0.59	0.59
8	0.99	0.99	0.98	0.99	0.99	0.98	0.99	0.98	0.98
9	0.53	0.52	0.51	0.53	0.53	0.51	0.52	0.51	0.51
10	0.71	0.69	0.68	0.71	0.71	0.69	0.69	0.68	0.69
11	0.93	0.91	0.91	0.93	0.93	0.91	0.92	0.9	0.92
12	0.80	0.77	0.77	0.79	0.80	0.77	0.78	0.77	0.78
13	0.68	0.67	0.66	0.68	0.68	0.66	0.67	0.66	0.67
14	0.71	0.69	0.69	0.71	0.71	0.68	0.69	0.68	0.69
15	0.97	0.97	0.97	0.97	0.97	0.97	0.97	0.97	0.97
16	0.86	0.88	0.86	0.86	0.87	0.88	0.88	0.85	0.87
17	0.93	0.94	0.92	0.93	0.94	0.94	0.95	0.92	0.93
18	0.77	0.80	0.77	0.76	0.78	0.79	0.80	0.77	0.78
19	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94
20	0.90	0.92	0.88	0.89	0.91	0.91	0.93	0.88	0.89
μ	0.84	0.83	0.82	0.83	0.84	0.83	0.83	0.82	0.83

deviating minimally because the standard error were zero across all the records. In the bias-corrected jackknifed estimate column, the non-deterministic algorithm records 0.83 compared to the e-greedy that recorded 0.82. Generally, the non-deterministic algorithm predictions are stable and reliable.

$$Var(\theta) = \frac{n-1}{n}\sum_{i=1}^{n}(\bar{\theta}_i - \bar{\theta}_{jack})^2, \quad \bar{\theta}_{jack} = \frac{1}{n}\sum_{i=1}^{n}(\bar{\theta}_i) \tag{2}$$

$$\bar{\theta}_{BiasCorrected} = N\bar{\theta} - (N-1)\bar{\theta}_{jack} \tag{3}$$

4 Conclusion

Hyperparameter tuning is a complex optimization task and time consuming. In this research, we formulated a non-deterministic strategy in searching an optimal $NoTs$ hyperparameter in RF algorithm. The goal of this strategy was to maximize predictability, minimizing $NoTs$ and minimizing $ToEs$. We compared experiment results with the deterministic algorithm, e-greedy and default configured RF. The non-deterministic strategy recorded significantly good results in maximizing *acc*, minimizing $NoTs$ and minimizing searching time. Moreover, evaluations using Jackknife Estimation show that its predictions are stable. The non-deterministic strategy had a significant *acc* levels and better utilization of cpu processing and time in memory. This research can be adopted in algorithms hyperparameter search and in green computing to preserve computing resources.

References

1. Bernard, S., Heutte, L., Adam, S.: Influence of hyperparameters on random forest accuracy. In: Benediktsson, J.A., Kittler, J., Roli, F. (eds.) MCS 2009. LNCS, vol. 5519, pp. 171–180. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02326-2_18
2. Breiman, L.: Random forests. J. Mach. Learn. **45**(1), 5–32 (2001). https://doi.org/10.1023/A:1010933404324
3. Breiman, L., Cutler, A.: Random forests manual v4.0 (2017). https://www.stat.berkeley.edu/~breiman/Using_random_forests_v4.0.pdf
4. Chen, T., Guestrin, C.: XGBoost: a scalable tree boosting system. In: 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 785–794. ACM (2016). https://doi.org/10.1145/2939672.2939785
5. Dheeru, D., Karra Taniskidou, E.: UCI machine learning repository (2017). http://archive.ics.uci.edu/ml
6. Ganjisaffar, Y., Debeauvais, T., Javanmardi, S., Caruana, R., Lopes, C.V.: Distributed tuning of machine learning algorithms using MapReduce clusters. In: 3rd Workshop on Large Scale Data Mining: Theory and Applications. ACM (2011). https://doi.org/10.1145/2002945.2002947
7. Hazan, E., Klivans, A., Yuan, Y.: Hyperparameter optimization: a spectral approach. arXiv preprint arXiv:1706.00764 (2017)
8. Kaggle: Kaggle datasets. https://www.kaggle.com/datasets

9. Lalor, J., Wu, H., Yu, H.: Improving machine learning ability with fine-tuning (2017)
10. Liu, X., et al.: Semi-supervised node splitting for random forest construction. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 492–499 (2013). https://doi.org/10.1109/CVPR.2013.70
11. Oshiro, T.M., Perez, P.S., Baranauskas, J.A.: How many trees in a random forest? In: Perner, P. (ed.) MLDM 2012. LNCS (LNAI), vol. 7376, pp. 154–168. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31537-4_13
12. Senagi, K., Jouandeau, N., Kamoni, P.: Using parallel random forest classifier in predicting land suitability for crop production. J. Agric. Inform. **8**(3), 23–32 (2017). https://doi.org/10.17700/jai.2017.8.3.390
13. Smit, S.K., Eiben, A.E.: Comparing parameter tuning methods for evolutionary algorithms. In: IEEE Congress on Evolutionary Computation, pp. 399–406. IEEE (2009). https://doi.org/10.1109/CEC.2009.4982974
14. Snoek, J., Larochelle, H., Adams, R.P.: Practical Bayesian optimization of machine learning algorithms. In: Advances in Neural Information Processing Systems, pp. 2951–2959 (2012)
15. Wager, S., Hastie, T., Efron, B.: Confidence intervals for random forests: the jackknife and the infinitesimal jackknife. J. Mach. Learn. Res. **15**(1), 1625–1651 (2014)
16. White, J.: Bandit Algorithms for Website Optimization. O'Reilly Media, Inc., Farnham (2013)

Author Queries

Chapter 31

Query Refs.	Details Required	Author's response
AQ1	This is to inform you that corresponding author has been identified as per the information available in the Copyright form.	
AQ2	Per Springer style, both city and country names must be present in the affiliations. Accordingly, we have inserted the city names "Nyeri" and "Saint-Denis" in affiliations "1" and "2". Please check and confirm if the inserted city names are correct. If not, please provide us with the correct city names.	