|  |  |  |
| --- | --- | --- |
| **TECHNICAL UNIVERSITY OF KOŠICE**  **Faculty of Electrical Engineering and Informatics** | |  |
| **Random forest machine learning algorithm** | |  |
|  | |  |
| **2024** | **st. Yevhen Haidash** |  |
| **TECHNICAL UNIVERSITY OF KOŠICE**  **Faculty of Electrical Engineering and Informatics** | |  |
| **Random forest machine learning algorithm** | |  |
|  | |  |
| Study programme: | Informatika |  |
| Field of study: | Inteligentné systémy |  |
| Training workplace: | KKUI |  |
| Educator: | prof. Ing. Kristína Machová, Ph.D. |  |
| Consultant: | Ing. Miroslava Matejová |  |
|  | |  |
| **2024** | **st. Yevhen Haidash** |  |

**Abstract**

This paper explores the Random Forest machine learning algorithm, reviews the theoretical principle of the algorithm, and describes the practical process of implementing Random Forest in the Python programming language. The implementation code with all the functions used is thoroughly reviewed in the document, providing the reader with a complete understanding of the implementation. The program developed by the author has been tested on a real-world dataset, and its performance has been evaluated using common metrics. The author was able to achieve an average accuracy value of 0.71 in his program, which may indicate a successful implementation of the Random Forest algorithm. The paper also includes an analysis of the results obtained and conclusions about the advantages, disadvantages, and effectiveness of the algorithm in practical machine learning tasks.

**Keywords**

Machine learning, Random Forest, Decision trees, Bootstrap samples, Entropy, Variable importance, C4.5 algorithm, Python, Classification Algorithm

1. Theoretical description of the Random Forest algorithm
   1. General definition of Random Forest

Random Forest (hereafter referred to as RF) is a machine learning algorithm for classification and regression that works by constructing a set of decision trees from randomly generated samples of a dataset, in the context of RF these samples are called Bootstrap. The RF algorithm makes predictions based on the voting of individual trees. To understand how it works, we will consider the following concepts: decision tree, rule generation, Bootstrap sample

In order to generate random trees, you must first create a Bootstrap. A Bootstrap is a subset that consists of random rows and columns from the original dataset. First, 60% to 80% of the rows from the original dataset are randomly selected for the bootstrap, and it is important to note that these rows may be repeated. Then, a subset of unique columns is randomly selected from the resulting subset, with the target column always present. For the classification task, the number of selected columns is equal to the root of the number of attributes of the original dataset. For regression, on the other hand, typically one-third of the total number of attributes is selected.

A diagram of a company

Description automatically generated The next step after creating the Bootstrap is to build the decision tree (image 1). „ *A decision tree is a non-parametric supervised learning algorithm, which is utilized for both classification and regression tasks. It has a hierarchical, tree structure, which consists of a root node, branches, internal nodes and leaf nodes.*[*[1]*](https://www.ibm.com/topics/decision-trees)*.“* The root node contains the first conditional rule that splits the dataset. The branches of the decision tree are additional nodes after the root node that correspond to a particular condition or value and indicate which node or leaf to consider next. A leaf or leaf node is the final node that represents the result of a classification or prediction.

Image 1 Decision tree

* 1. Generating a decision tree

There are many algorithms to build a decision tree. The most commonly used are Iterative Dichotomizer 3 (ID3) and C4.5. The ID3 algorithm has one significant drawback - it only works with nominal values. Therefore, in this paper we will discuss the C4.5 algorithm, which is an improved modification of ID3. The C4.5 algorithm can handle both continuous and discrete values and can learn from data containing gaps.

* 1. Algorithm C4.5

C4.5 allows you to work with different types of data and solve the problem of data loss. The basic idea of the algorithm is to recursively find the best attribute to partition the data at each node. The attribute to be split is found using metrics such as entropy or Gemini index. The selected attribute splits the dataset into two or more subsets. This process is repeated until all objects in a node belong to the same class or the maximum tree depth is reached.

Consider the principle of finding the most informative discrete attribute for separation. Initially, we need to compute the global entropy of the dataset H(S), where p(i) is the probability of a particular value of the target attribute, see formula (1)

*( 1 )*

Now let us look at our metric by which we select the most informative attribute ( 2 ). Our metric is the gain ratio where (information gain) is a measure of how much partitioning the dataset S using attribute A will reduce the entropy of the classes at a decision tree node, and is thus the entropy of the subgroups created by partitioning the dataset S using attribute A.

*( 2 )*

is computed according to the following formula ( 3 ), where is the entropy of the dataset S after partitioning according to the attribute A.

*( 3 )*

Sequentially, is computed according to formula ( 4 ), where n is the number of subgroups formed by dividing attribute A, is the probability of finding an element in subgroup 𝑖, and is the entropy of subgroup 𝑖.

*( 4 )*

The subgroup entropy is given by ( 5 ), where is the number of classes in the subgroup 𝑖, is the probability of belonging to a class in the subgroup 𝑖

*( 5 )*

Finally, we compute using formula ( 6 ), where n is the number of subgroups formed by splitting according to attribute A, is the probability of selecting a subgroup 𝑖

*( 6 )*

The principle of evaluating the profit ratio for nominal attributes is slightly different from the previous one.

We first select a set of unique values in the nominal column and then sort this set in ascending order. From this set, we select intermediate values called threshold values. We then iterate over all the threshold values and evaluate their gain ratios using the same formulas. For example, for threshold N in formula (5), we will always have 2 subsets, namely rows with values < N and rows with values > N.

* 1. General principle of RF operation

Let us summarize all the steps and the main sequence of the RF algorithm. First, the required number of bootstrap subsets are created from the original dataset. Then, a decision tree is constructed from each subset using the C4.5 algorithm, which can also be called training because this algorithm adapts the rules to the original dataset. Once all the decision trees have been created, you can start predicting the new data. A new row is classified by each individual tree. Then the solution of our algorithm will be the class predicted by the largest number of trees.

1. Algorithm implementation in code

The author decided to write the RF algorithm in Python programming language. The implementation followed the functional programming paradigm, which emphasizes the use of functions. Therefore, 4 functions were first written to calculate the gain ratio using all of the above formulas. In addition to these functions, a recursive function to build a decision tree and a function to predict classes were also created. Using all these functions and the main block of code, the RF algorithm was implemented.

* 1. Description of application functions

The function **calculate\_entropy(column)** calculates the entropy of a column according to formula 1. Figure 2 shows that first a dictionary is created that counts the number of individual attributes in the column. The values of this dictionary are iteratively computed according to the above formula and added to the variable **sum**, which is an indicator of entropy and serves as the value returned by the function.

A computer screen with white text

Description automatically generated

figure 2. Calculate\_entropy

The function **calculate\_info\_gain** **(dataframe, datasetEntropy)** is used to calculate the information gain (formula 3). This function accepts the variables dataframe and datasetEntropy (figure 3), where the variable dataframe contains the column for which the information gain is to be calculated and the target column, and datasetEntropy is in turn the entropy of the whole dataset, i.e. the entropy of the target column. The function creates a set of unique attributes from the 1st column of the dataset and then iteratively creates subsets from the rows that correspond to the current attribute. It then calculates a probability from the length of this subset and the original data frame, which is then multiplied by the entropy of the target column of the subset, which in turn is calculated using the **calculate\_entropy** function

A computer screen with white text

Description automatically generated

Image 3 calculate\_info\_gain

The **get\_most\_informative\_feature(dataset: pd.DataFrame)** function accepts as input the original data frame or a subset of it. The function is a join function that combines **calculate\_info\_gain** and **calculate\_entropy** and is used to calculate the importance of information (formula 2). The function returns two values: the column name and the best threshold. If the most informative column contains discrete values, then the threshold will be None, but if the most informative column is of numeric type, the threshold takes the value of the most informative threshold for separation.

In the code it is implemented as follows: All columns of the data frame are searched except the target column. If the column is of a discrete type, its information importance is calculated using the two functions above (figure 4), - feature\_entropy, computed using the same **calculate\_entropy** function, since formulas 1 and 6 are almost identical. If the information importance of a particular column is greater than the current maximum, then the maximum is set to a new value and the column name is saved. If the current column is a discrete data type, the algorithm is augmented with a part that generates threshold values (figure 5). The algorithm creates a set of unique values in the column. It then creates threshold values for these values, i.e., the values that lie between the attribute values. Then, according to these thresholds, the column is split into two: a column with values higher than the thresholds and a column with values lower than the thresholds. These two columns are passed to formula number 4 to calculate the entropy of the data frame after splitting the thresholds. All other steps of the algorithm are similar to the discrete type column.

A screen shot of a computer

Description automatically generated

figure 4 get\_most\_informative\_feature 1

A screenshot of a computer program

Description automatically generated

figure 5 get\_most\_informative\_feature 2

The **built\_decision\_tree(dataset)** function takes as input a dataset or subset of data and recursively builds a decision tree from it, which is represented as a dictionary (figure 6). At the beginning of the function, there are several checks that are used to terminate the recursion in the current branch. A condition for recursion termination is a situation where only one class remains in the target column, or a situation where the dataset can no longer be partitioned, in which case the function returns the most frequent class in the target column as one of the final nodes of the dictionary. After these checks, the dictionary **tree = {best\_feature: {}}**, where the best\_feature is looked up using the **get\_most\_informative\_feature()** function. Then all possible values of the best column are added to the value under this key if it is a discrete type. If it is a numeric type, this partitioning is done using the **get\_subset\_number()** function, after which a fork is created on the subset with values below the threshold and with values above the threshold. Finally, after traversing through all nodes, the function returns a decision tree in the form of a dictionary.

A screen shot of a computer program

Description automatically generated

figure 6 built\_decision\_tree

The function **get\_subset\_number(dataset, feature, symbol, number)** is a helper function for the **built\_decision\_tree** function (figure 7). It takes the dataset to be partitioned, a feature of numeric type, a **symbol** that must be "<" or ">", and a number that is our threshold. The function splits the dataset into a subset with numeric attribute values less than the threshold or greater than the threshold, depending on the **symbol** parameter. The return value is the corresponding data frame.

A computer code with white text

Description automatically generated

figure 7. get\_subset\_number

And the last function, **predict(decisionTree, row)**, predicts the class for a given **row** by running it through the **decisionTree** (figure 8). Traversal through the tree is recursive until a leaf is reached. If the current tree fails to make a prediction, the value None is returned. Theoretically, this situation is possible because there is always the possibility that the subsets generated by the bootstrap will not have enough attributes to classify a particular example. In practice, with a large number of trees, such an event is too rare to significantly affect the result.

A computer screen shot of a program

Description automatically generated

figure 8 predict

* 1. Main part of the code

In the main code, it all starts with setting hyperparameters such as the dataset tree path, the size of the training part relative to the training part, the number of trees, and so on. Then incomplete rows are removed from the dataset and the target column defined by the hyperparameter is set last. For efficient training, the ratio of target classes in the dataset is also adjusted by removing the rows with the predominant class. After preprocessing the data, the dataset is blended and divided into training and testing parts.

In a cycle with the number of iterations equal to **countTrees**, 60% to 80% of the rows are randomly selected from the training set. Then a subset of columns, including the target column, is randomly selected from this subset. For classification, the code selects the number of columns equal to the root of the initial count. This subset is our bootstrap. In the next step, a decision tree is created from this bootstrap using the built\_decision\_tree function and added to the list of all trees. When this list is complete, the learning process is finished.

In the test part, all strings from the test dataset are iteratively traversed through all trees and their predictions are recorded in a dictionary, where the key is the class and the value of the key is the number of trees that predicted that class. Then the key in the dictionary with the highest value will be the prediction of our algorithm. This prediction is then checked against the tag and, depending on the correctness of the answer, it is added to the appropriate count, i.e., True Positive, False Positive, True Negative, and False Negative. At the end of the code, when all rows from the test dataset are predicted and checked, the accuracy of the algorithm is evaluated using common metrics. The general code pattern can be seen in the pseudocode (Fig. 9)

A screenshot of a computer

Description automatically generated

figure 9 Pseudo-code

1. Description of the dataset used
   1. Selected dataset

To evaluate the algorithm, we used the diabetes dataset (figure 10) from the online platform Kaggle.com. This dataset contains 8 columns with attributes and one target "Outcome" representing the diagnosis. This dataset is from real life, here is its description from the Kaggle platform[[2]](https://www.kaggle.com/datasets/ehababoelnaga/diabetes-dataset?rvi=1&select=Training.csv): „*The Diabetes Dataset contains information about individuals diagnosed with diabetes, including demographic attributes, medical history, and clinical measurements. This dataset serves as a valuable resource for studying diabetes management, risk factors, and predictive modeling for disease outcomes.*“

* 1. Description of attributes

Let's look at the individual attributes. Pregnancies - pregnancy with a value from 0 to 17, Glucose - glucose level from 0 to 179, **BloodPressure** - blood pressure from 0 to 122, **SkinThickness** - skin thickness with a value from 0 to 63, Insulin - insulin level from 0 to 846, BMI - body mass index with a value from 0 to 67. 1, **DiabetesPedigreeFunction** - a function that measures the value of diabetes in the pedigree and has a value ranging from 0.08 to 2.42, Age - age of the patient, with values ranging from 21 to 81 in the dataset, and finally the target attribute Outcome, which has a value of 0 - negative diagnosis and 1 - positive diagnosis.

A screen shot of a computer

Description automatically generated

figure 10. Diabet dataset

* 1. Code dataset usage properties

In my code I used a larger version of this dataset, which had 2460 rows. Of these 2460 rows, 1500 were negative and 960 were positive. This ratio indicates that you should balance the number of positive and negative rows before using it. In addition, all rows in the data set are complete, which means that the data set has already been processed.

1. Evaluation of the algorithm and conclusions
   1. Metrics used

In the RF algorithm implementation code, 4 metrics were used to evaluate the predictions (Fig. 11), namely Precision, Accuracy, Recall and F1-measure. All these metrics are calculated using the following values: TP(True positive) - the number of positive classes classified as positive, FN(False negative) - the number of positive classes classified as negative, TN(True negative) - the number of negative classes that were classified correctly, and FP(False positive) - the number of negative classes that were misclassified as positive.

Accuracy measures how often the model correctly identifies positive outcomes. Correctness is used to show how often the model tracks all positive outcomes. Recall reflects the overall performance of the classifier. F1 - This is a harmonic average of Accuracy and Correctness, which provides a balanced accuracy between the two metrics. The implementation of these metrics and the calculation of the required values is clearly shown in the figure (figure 11).

A computer screen with white and blue text

Description automatically generated

figure 11 Metrics in code

* 1. Results obtained

After implementing the algorithm, the author proceeded to its evaluation. The evaluation was carried out on a dataset that contained 1920 rows after data preparation and class balancing. Of these, 70% (1344 rows) were grouped into a training set and the remaining 30% (576 rows) were grouped into a test set. The values of some hyperparameters are also worth mentioning. The number of columns for the bootstrap was 4 and the number of rows was 941. The number of trees was set to 100.

The following results were obtained (figure 12):

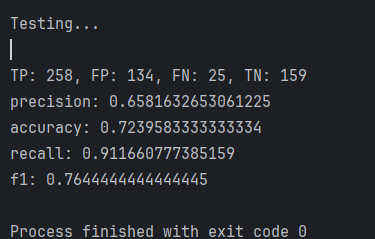


figure 12 results

* 1. Conclusion

Based on the results obtained, it can be said that the author has succeeded in implementing the Random Forest machine learning algorithm using C4.5. The code is written in Python, is flexible and can be used with any other data sets. The algorithm can work with both discrete and numeric column values. It also provides features to optimize training, such as the ability to remove uninformative rows and to balance the number of rows of different target classes.

The precision value is approximately 66%, which means that about two-thirds of the positive examples detected by the algorithm are true positives. The accuracy value shows that the algorithm correctly classifies examples 72% of the time. The recall value is approximately 91%, which means that the algorithm detects most of the true positive examples. The 1 rate is approximately 76%, which means that the algorithm achieves an acceptable balance between accuracy and completeness.

List of images

[Obr. 1 Decision tree 4](#_Toc118285416)

[Obr. 2 Calculate\_entropy 4](#_Toc118285417)

[Obr. 3 Сalculate\_info\_gain 4](#_Toc118285418)

[Obr. 4 get\_most\_informative\_feature 1 4](#_Toc118285419)

[Obr. 5 get\_most\_informative\_feature 2 4](#_Toc118285420)

[Obr. 6 built\_decision\_tree 4](#_Toc118285421)

[Obr. 7 get\_subset\_number 4](#_Toc118285422)

[Obr. 8 predict 4](#_Toc118285422)

[Obr. 9 Pseudo-code 4](#_Toc118285422)

[Obr. 10 Diabet dataset 4](#_Toc118285422)

[Obr. 11 Metrics in code 4](#_Toc118285422)

[Obr. 12 results 4](#_Toc118285422)

List of literature used

1. https://www.ibm.com/topics/decision-trees
2. <https://www.kaggle.com/datasets/ehababoelnaga/diabetes-dataset?rvi=1&select=Training.csv>
3. <https://kristina.machova.website.tuke.sk/prezentacieSU/>
4. <https://www.youtube.com/watch?v=J4Wdy0Wc_xQ&ab_channel=StatQuestwithJoshStarmer>
5. https://www.youtube.com/watch?v=gkXX4h3qYm4&t=10s&ab\_channel=IBMTechnology

**Contents**

[1. Theoretical description of the Random Forest algorithm 4](#_Toc174115274)

[1.1. General definition of Random Forest 4](#_Toc174115275)

[1.2. Generating a decision tree 4](#_Toc174115276)

[1.3. Algorithm C4.5 5](#_Toc174115277)

[1.4. General principle of RF operation 6](#_Toc174115278)

[2. Algorithm implementation in code 7](#_Toc174115279)

[2.1. Description of application functions 7](#_Toc174115280)

[2.2. Main part of the code 13](#_Toc174115281)

[3. Description of the dataset used 15](#_Toc174115282)

[3.1. Selected dataset 15](#_Toc174115283)

[3.2. Description of attributes 15](#_Toc174115284)

[3.3. Code dataset usage properties 15](#_Toc174115285)

[4. Evaluation of the algorithm and conclusions 16](#_Toc174115286)

[4.1. Metrics used 16](#_Toc174115287)

[4.2. Results obtained 16](#_Toc174115288)

[4.3. Conclusion 17](#_Toc174115289)

[List of images 18](#_Toc174115290)

[List of literature used 19](#_Toc174115291)