Designing a non-overfit model for solving object categorization problems

How does reducing the model complexity help to solve the problem of overfitting?

A Computer Science Extended Essay

Vadim Zaripov

3982 words

Contents

1 Introduction

Machine learning is one of the fastest-growing areas of computer science. ML algorithms can solve problems of object categorization, clusterization, and regression.

However, many complex ML models, such as decision trees, are prone to the overfitting problem: when there is too little data to train a model, the model can "correspond too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably"[1]. In other words, the model is so complex that it can handle nearly each training object separately, without evaluating the general dependency, and therefore perform inaccurately when working with previously unseen data.

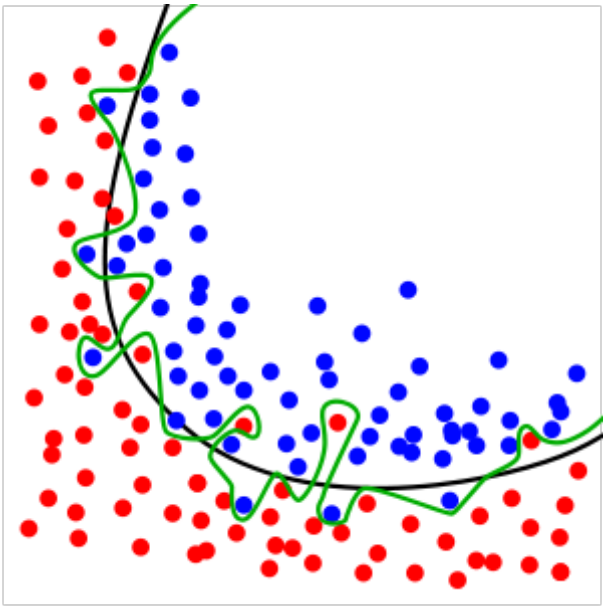


Figure 1: Overfitting model solving object categorization problem[2]

Figure 1 presents an example of an overfitting model. Its goal is to separate blue points from red points by a green line. Although it does so perfectly for the existing points, it is unable to find an actual pattern in the point's positions (this pattern is represented by the black line in the figure).

The essay describes a logistic regression tree categorization model, discusses the op-

[1] "Overfitting — Meaning of Overfitting by Lexico," Lexico Dictionaries — English, n.d., https://www.lexico.com/definition/overfitting

[2] Chabacano, "Diagram Showing Overfitting of a Classifier," Wikimedia Commons, February 1, 2008, https://commons.wikimedia.org/wiki/File:Overfitting.svgfile.

timisation of its parameters, and compares this model with a decision tree to see how reducing the model complexity help to reduce overfitting. Moreover, the essay will compare the model with the random forest, which is also used to solve the overfitting problem.

This essay is an attempt to design a categorization model which will solve the overfitting problem by simplifying the model. This essay will explore how reducing the number of model parameters would help to increase the accuracy of the model.

1.1 Problem statement

The algorithm should process an object in order to predict its class.

Object. An object is anything that has to be classified. Every object is determined by the set of parameters (numbers), which form a vector (in this essay this vector is defined as x).

Class. Every object belongs to exactly one class. For example, animals can be classified into families, and every animal belongs to only one family.

For the model to be accurate, the data is needed. The model will be trained by a set of objects (**training sample**) whose classes are already known. This set will provide the model with the essential data needed for assigning a class label to each object.

1.2 Metrics of the model quality

There already is a variety of methods that allow solving the categorization problem. That is why metrics of the successive model must be developed to make sure that it is useful for some purposes.

Below there are requirements which the model fulfill: number of model parameters and accuracy.

1.2.1 Number of model parameters

As the main goal of the model is to try to solve the overfitting problem by reducing the number of its parameters, restrictions on the number of parameters should be imposed. In this essay, the required number of model parameters would need to be significantly

smaller than the number of parameter's of other models used for comparison (the random forest and the decision tree). This is needed to see how reducing the number of parameters of a model affects its overfitting chances.

1.2.2 Accuracy

In object categorization problems, the accuracy of a model on a sample is equal to the proportion of a correctly categorized objects in this sample. No matter how efficient and simple an algorithm is, it will not be useful if not accurate. The successful model has to me competitive with the existing ones, and it would be considered successful if the difference between its accuracy would at worst be 10% less than the accuracies of other models used for comparison (decision tree and random forest).

As the algorithm is trained on train data, its performance cannot be evaluated based on the same data, as some information from this data is already in the model. Objects unseen by the model are needed to estimate its accuracy. Therefore, all objects from the dataset are usually randomly split into **training sample** and **testing sample**. The model is trained on training sample and then tested on testing sample.

1.3 Hypothesis

The hypothesis is that the model designed will have less overfitting on small datasets (having less than 1000 objects) than more complex models. This means that while potentially performing worse when classifying training sample, the accuracy on testing sample will be higher. Moreover, the difference between training and testing sample's accuracy will also be smaller.

This hypothesis is supported a number of existing research papers, including the work by Xue Ying[3] as well as the article by Douglas M. Hawkins, who says that overfit models violate the principle of parsimony by having more parameters than necessary, which leads

[3]Ying, Xue. "An Overview of Overfitting and Its Solutions." Journal of Physics: Conference Series 1168, no. 2 (February 2019): 022022. https://doi.org/10.1088/1742-6596/1168/2/022022

to "adding irrelevant predictors can make predictions worse because the coefficients fitted to them add random variation to the subsequent predictions"[4].

1.4 Variables used

Here are some notations that will be used throughout the essay:

- M is the number of classes
- N is the number of training objects
- x is the vector that consists of object's parameters
- d is the number of parameters describing an object ($d = \dim x$)

2 Background information

This essay is an alternative look at an already developed field of machine learning. Therefore it will rely on some already well-known approaches, which this section will examine.

2.1 Logistic regression

The main problem of the essay is already solved in the case of two classes, it is called logistic regression[5]. This essay would extensively use its ideas.

Logistic regression finds such a function $f(x)$ (where x is a vector of object's parameters), that if $f(x) = 0$, the object belongs to the first class, and if $f(x) = 1$ – to the second class.

Let this function be determined like that:

[4]Hawkins, Douglas M. "The Problem of Overfitting." ChemInform 35, no. 19 (May 11, 2004). https://doi.org/10.1002/chin.200419274

[5]Galarnyk, Michael. "Logistic Regression Using Python (Scikit-Learn)." Towards Data Science, 29 Apr. 2020, towardsdatascience.com/logistic-regression-using-python-sklearn-numpy-mnist-handwriting-recognition-matplotlib-a6b31e2b166a

$$f(x) = \begin{cases} 0, \text{ if } \dfrac{1}{1+e^{\langle \omega, x \rangle}} < 0.5 \\ 1, \text{ if } \dfrac{1}{1+e^{\langle \omega, x \rangle}} \geq 0.5, \end{cases}$$

where $\langle \omega, x \rangle$ is a dot product of vector x and a constant vector of weights ω.

It can be seen that $f(x)$ is determined by this vector ω. Generally, ω contains coefficients for object's parameters as $\langle \omega, x \rangle$ is a linear function of d parameters. The rest of the f function is needed in order for its values to be between 0 and 1.

Usually, the x vector is complemented by the element 1 in the end. This way, the last element of ω is a free term of the linear function $\langle \omega, x \rangle$. Having this free term allows to cover all possible linear functions. That is why by the vector x will be further considered vector of object's parameters complemented by the element 1.

The only thing which is needed is to find such ω that it will give the most accurate predictions. In order to understand how good a chosen vector ω is for a particular case, a **loss function** $\xi(\omega)$ is introduced:

$$\xi(\omega) = \tfrac{1}{N} \cdot \sum |y_i - f_\omega(x_i)|,$$

where N is the total number of elements in the training sample, y_i is a label of i-th element's class (0 or 1), and $f_\omega(x_i)$ is the prediction of the model for this object.

This function shows the percentage of the objects which were classified incorrectly. The only thing which the algorithm has to do is to minimize this function: find such ω that $\xi(\omega)$ is minimal.

This is done by **gradient descent**, which is a sequence of the following steps:

- The algorithm calculates a **gradient** of $\xi(\omega)$ in point ω. The gradient of a function in a certain point is a vector whose direction indicates the direction in which function grows faster and whose length indicates how fast it grows. In other words, the gradient is similar to the derivative, but in the case of a function that takes more than one parameter.

- The algorithm calculates new ω by moving the last one to the direction opposite to the gradient (direction of the fastest decrease) by a value proportional to the gradient's magnitude.

By performing these steps many times, the point ω would move to the direction where the function $\xi(\omega)$ decreases and eventually would end up in a point, where $\xi(\omega)$ is minimal, which is exactly what is needed for logistic regression.

2.1.1 Graphical representation of logistic regression

Consider the d-dimensional space. In this space, every object is a point, and, as it was already mentioned, $\langle \omega, x \rangle$ is a linear function of $\dim x$ parameters. $\langle \omega, x \rangle = 0$ determines a multidimensional plane in this space, and $|\langle \omega, x \rangle|$ is the distance from a point x to this plane. The sign of $\langle \omega, x \rangle$ indicates in which of half-spaces divided by the plane the point is.

The regression predicts the class to be 0 if:

$$
\begin{gathered}
f(x) = 0 \\
\Updownarrow \\
\tfrac{1}{1+e^{\langle \omega, x \rangle}} < 0.5 \\
\Updownarrow \\
e^{\langle \omega, x \rangle} > 1 \\
\Updownarrow \\
\langle \omega, x \rangle > 0
\end{gathered}
$$

This means that an object will be classified as 0 if and only if the point in multidimensional space with the coordinate x is on one particular half-space by which the plane $\langle \omega, x \rangle$ divides the space.

This means, that the logistic regression finds the most accurate plane which separates points representing objects of one class from points representing objects from the other.

2.2 Decision trees

Another machine learning model which is important for the essay is a decision tree. The principle of its working is easier to understand, but the way it is trained is actually not that obvious. However, as it is not explicitly used in this essay, this section would mostly focus on the way it calculates predictions, not the way it is trained.

A decision tree is a binary tree, to each of whose leaves a certain class is assigned (one class can be assigned to multiple leaves). In other vertexes, conditions are placed (every condition determines if an object goes to the left child or to the right child). Therefore, an object starts from the root and through the sequence of conditions gets to a leaf. The class assigned to this leaf would be a tree's prediction for this object[6].

In decision trees, every condition is a comparison between one object's parameter and a constant value. As determining an object's class precisely might require many conditions, the depths of decision trees are usually big, meaning that in some cases it is prone to overfitting. The model which this article is trying to create should contain less information.

2.3 Random forests

Decision trees are prone to overfitting as their structure is extremely flexible (their depth can be high, which allows to handle all training objects separately). One of the methods used to reduce overfitting is training a set of different decision trees and uniting them in an ensemble called the **random forest**. In a random forest, the prediction is found by calculating the prediction for each decision tree in an ensemble and then choosing the class for which more trees voted.

Research shows[7] that ensemble models are efficient in reducing the chance of overfitting, which makes random forests particularly interesting for this essay.

However, this model has some significant drawbacks compared to the logistic regres-

[6]Chollet, François. Deep Learning with Python. Shelter Island (New York, USA), Manning, Cop, 2018

[7]Dietterich, Thomas G. "Ensemble Methods in Machine Learning." Multiple Classifier Systems, 2000, 1–15. https://doi.org/10.1007/3-540-45014-9_1

sion tree, which this essay describes. It is computationally expensive, which means that it requires a lot of time to train a model and compute its predictions. Moreover, it is difficult to interpret[8]. Therefore, if logistic regression tree shows similar accuracy, it will be better for solving the categorization problem on small datasets.

3 Method

3.1 Model description

This essay will work with the **logistic regression tree**. This model is similar to the decision tree, as it has a similar structure. It is a binary tree with the classes assigned to leaves.

The main reason decision trees overfit is the high number of comparisons (vertexes), which allows to handle every training object separately. That is why in this model the number of vertexes is limited: each class can be assigned only to one leaf. Therefore, the number of leaves will be constant, as well as the overall number of vertexes.

However, such a strong limitation will significantly impact the model accuracy. For example, a root vertex splits all classes in two separate groups by only one comparison, which cannot be done properly. Therefore, conditions stored in each vertex should be more complex.

In the model of this essay, logistic regressions play the role of such conditions. For example, the regression in the root considers first and second halves of classes as two big classes and is trained to separate these two groups.

This allows conditions in vertexes to be more complex, and an algorithm to be much more accurate. However, the pure exploration of whether or not the model fits the established metrics will be given further.

[8]Vihar Kurama, "Random Forests: Consolidating Decision Trees," Paperspace Blog, February 16, 2020, https://blog.paperspace.com/random-forests/amp/

3.2 Optimization of model parameters

In the logistic regression tree model, logistic regressions are not the only thing determining how the model will perform. It is also dependent on the permutation of classes in leaves a lot.

Consider a problem of 4 classes categorization, where objects are distributed as presented in the figure 2. Here objects are determined by **only 2 parameters** (their x and y coordinates on a plane), but this example may be escalated to higher-dimensional objects.

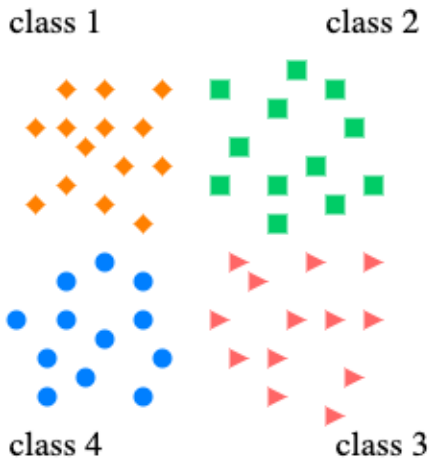


Figure 2: Possible objects' distribution

If classes are organized in the tree as presented in the figure 3a, the root regression would have to separate classes 1 & 3 from classes 2 & 4. It is impossible to do by logistic regression, as it separates classes with a linear function as explained in 2.1.1.

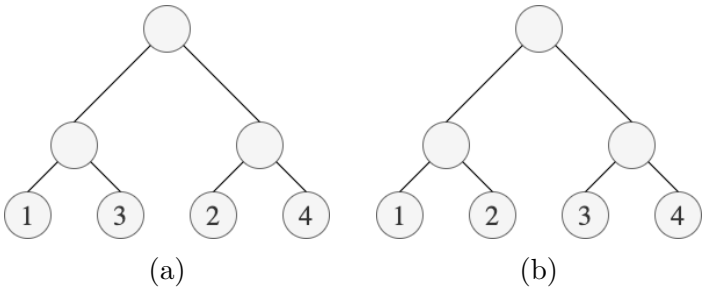


Figure 3: Different permutations of classes in leafs

However, if classes are organized as presented in the figure 3b, the root regression would have to separate classes 1 & 2 from classes 3 & 4, which can be done by logistic

regression (figure 4).

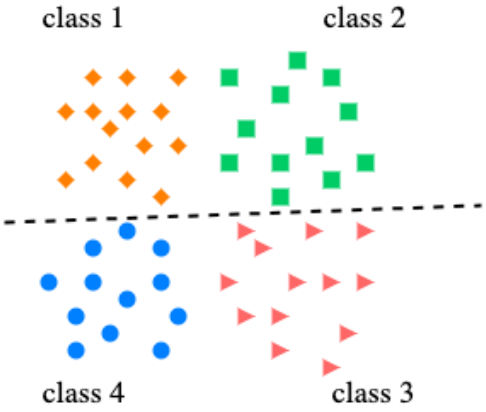


Figure 4: Objects 1 and 2 are separated from objects 3 and 4 by a linear function

This means, that organizing classes randomly is not an appropriate solution. When there are not many classes the problem can be solved by trying every permutation and finding the one that gives the best accuracy.

However, there are $M!$ permutations of M classes, and as the factorial function grows fast, such a method would take a long time even for a small number of classes.

In this essay, the permutation of classes in leaves of logistic regression tree is going to be optimized using **genetic algorithms**. These algorithms mimic the process of natural selection in an attempt to evolve solutions to otherwise computationally intractable problems.

Any genetic algorithm comprises of the following steps: initialization, selection, crossover, and mutation[9].

3.2.1 Initialization

This involves generating a random population of possible solutions to the problem. In our case, the set of random classes' permutations is generated.

[9]Johnson, Becky. "Genetic Algorithms: The Travelling Salesman Problem." Medium, 15 Nov. 2017, medium.com/@becmjo/genetic-algorithms-and-the-travelling-salesman-problem-d10d1daf96a1

3.2.2 Selection

A sample of permutations is taken from the population and placed in the mating pool (a set that is used to create the next population). This can be done in a number of different ways but, in general, permutations that achieve better accuracy have a higher probability of being selected to enter the mating pool.

A further design consideration in selection is to decide whether to ensure that the best solutions are carried to the next generation with certainty. This is known as **elitism**.

In this particular problem, a new logistic regression tree model would be trained on each object in the population; then the accuracy of each would be estimated using **k-fold cross-validation**[10]. In this method, all the objects are randomly split into k equal groups (in this model $k = 5$). Then, a model is trained and tested k times, each time with one of the groups playing the role of testing sample, and the others – training sample. The final accuracy is an average of these k results. Cross-validation is a more credible way to estimate the accuracy.

The algorithm uses **truncation selection**, meaning that the mating pool comprises of the most accurate 10% of the initial population.

3.2.3 Crossover

In biology, the crossover is the term given to the mechanism by which the chromosomes of a new individual are created from a combination of its parents' chromosomes.

In this step of the algorithm, a new population will be formed by randomly taking pairs of permutations from the mating pool and producing new permutations from them.

In this algorithm, the crossover step will combine pairs of permutations using the **order crossover** method[11]. It combines two permutations A and B by taking a random subsequence of A, copying it to the answer permutation, and then moving all other elements which did not appear in this sub-sequence in order that they appear in B. This

[10]Chollet, François. Deep Learning with Python. Shelter Island (New York, USA), Manning, Cop, 2018

[11]Koohestani, Behrooz. "A Crossover Operator for Improving the Efficiency of Permutation-Based Genetic Algorithms." Expert Systems with Applications, vol. 151, Aug. 2020, p. 113381, 10.1016/j.eswa.2020.113381

process would produce a new valid permutation, containing some features from both A and B.

3.2.4 Mutation

In biology, mutation refers to random errors in genetic information from one generation to the next. In a genetic algorithm, mutations are random changes in the population.

In this algorithm, the mutation is achieved by randomly changing pairs of classes in the permutation. After this step is done, the algorithm returns to the selection step of the genetic algorithm, but with the new population.

Each such cycle is called an **epoch**. After many epochs, the population would comprise of objects that give the best accuracy, and selecting the best of them all would solve the problem of finding the optimal permutation of classes in leaves of the logistic regression tree model.

4 Computational complexity

In this section, the number of computational operations needed to calculate the prediction will be estimated. For that purpose Big O notation will be used.

4.1 O notation

In computer science, Big O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows. It is formally defined this way:

If f and g are functions from positive integers to non-negative real numbers, then $f(n) = O(g(n))$ if there exist positive integers M and n_0 such that $f(n) < Mg(n)$ for all $n > n_0$[12].

In other words, O estimates the function growth.

[12]Kumar, Paritosh. "Time Complexity of ML Models." Medium, 31 Mar. 2021, medium.com/analytics-vidhya/time-complexity-of-ml-models-4ec39fad2770

4.2 Complexity of computing model prediction

The complexity of calculating a logistic regression's prediction is $O(d)$, where d is a number of object's parameters ($d = \dim x$), because each parameter should be multiplied by a certain coefficient, and then these values should be summarized.

However, predictions of more than one regression should be calculated in order to classify an object. The model tree has $O(M)$ leaves, where M is a number of classes (it can be slightly bigger when the number of classes is not an exact power of 2). Therefore, the depth of this tree is $O(\log M)$ because after going down each vertex the number of leaves reduces by 2.

As each level of the tree requires calculating a logistic regression's prediction, the overall complexity of the algorithm is $O(d \log M)$.

4.3 The number of model parameters

Each logistic regression has $O(d)$ parameters. The number of nodes in a binary tree is $2^0 + 2^1 + 2^2 + ... + 2^h = 2^{h+1} - 1 = O(2^h)$, where h is the depth of a tree[13]. Notice, that as the number of leaves of a tree doubles when the depth increases by 1, and if the number of leaves is M, then $O(2^h) = O(M)$. This means that the tree described in the model has $O(M)$ nodes, and $O(M)$ logistic regressions.

Notice, that each logistic regression has $O(d)$ parameters, meaning that the overall number of parameters of a model is $O(Md)$.

Such a level of complexity is better than the complexity of many other models (for instance, the number of parameters in a decision tree is at worst proportional to the size of the training sample[14]). Moreover, another good side is that regressions are trained separately, independently (one regression is not used for training another), reducing the chance of overfitting. Therefore, requirement of a number of model parameters (section 1.2.1) is satisfied.

[13] "Relationship between Number of Nodes and Height of Binary Tree," GeeksforGeeks, January 15, 2018, https://www.geeksforgeeks.org/relationship-number-nodes-height-binary-tree/

[14] "Machine Learning - Number of Nodes in an Unpruned Decision Tree." Cross Validated, n.d. https://stats.stackexchange.com/questions/114806/number-of-nodes-in-an-unpruned-decision-tree

5 Testing and comparison with existing models

5.1 Testing the model accuracy and overfitting

The model is targeted at working with small datasets, as its main goal is to minimize overfitting and reduce its computational complexity. Therefore, the performance of this model should be compared to others' for datasets of different sizes.

Different datasets for solving the categorization problem can be generated using Python Scikit-learn library[15]. Each sample is described by 3 features ($d = 3$), and the number of classes generated is 4 ($M = 4$). The number of decision trees in a random forest is 10.

For each dataset size, 10 different datasets are generated. Each dataset is split into training and testing samples (75% of a dataset objects are training sample and 25% – testing sample). For each model, the model accuracy is evaluated on both groups. Then, the average accuracies among all datasets of this particular size are calculated. This procedure of generating 10 datasets and calculating the average accuracy helps to reduce the impact of a particular dataset's features and evaluate the general tendency specific to this dataset size.

The difference between the accuracy on training and testing samples will estimate whether or not the model has overfitted: if it performs much worse on a data it has not seen, then overfitting occurred.

This data can be observed in figure 5.

[15] "Sklearn.datasets.make_blobs," scikit-learn, https://scikitlearn.org/stable/modules/generated/sklearn.datasets.make_blobs.html

# of objects in a dataset	Decision tree			Random forest			Logistic regression tree		
	Average train accuracy, A_{dtrain}	Average test accuracy, A_{dtest}	$A_{dtrain} - A_{dtest}$	Average train accuracy, A_{ntrain}	Average test accuracy, A_{ntest}	$A_{ntrain} - A_{ntest}$	Average train accuracy, A_{ltrain}	Average test accuracy, A_{ltest}	$A_{ltrain} - A_{ltest}$
200	1.000	0.566	0.434	0.977	0.606	0.371	0.681	0.604	0.077
250	1.000	0.624	0.376	0.983	0.668	0.315	0.732	0.686	0.046
300	1.000	0.551	0.449	0.978	0.633	0.345	0.681	0.641	0.04
350	1.000	0.643	0.357	0.980	0.690	0.290	0.723	0.699	0.0240
400	1.000	0.587	0.413	0.982	0.629	0.353	0.667	0.643	0.024
450	1.000	0.621	0.379	0.984	0.650	0.334	0.701	0.682	0.019
500	1.000	0.627	0.373	0.985	0.686	0.298	0.709	0.689	0.02
550	1.000	0.614	0.386	0.983	0.688	0.295	0.722	0.696	0.026
600	1.000	0.603	0.397	0.982	0.648	0.334	0.683	0.661	0.022
650	1.000	0.598	0.402	0.980	0.647	0.333	0.684	0.667	0.016
700	1.000	0.639	0.361	0.983	0.698	0.285	0.704	0.694	0.011
750	1.000	0.679	0.321	0.983	0.730	0.254	0.750	0.721	0.029
800	1.000	0.663	0.337	0.986	0.705	0.282	0.733	0.696	0.037
850	1.000	0.624	0.376	0.984	0.676	0.309	0.721	0.685	0.036
900	1.000	0.632	0.368	0.985	0.696	0.289	0.716	0.700	0.016
950	1.000	0.658	0.342	0.984	0.704	0.280	0.717	0.704	0.013
1000	1.000	0.600	0.400	0.982	0.653	0.328	0.683	0.668	0.016

Figure 5: Test data[16]

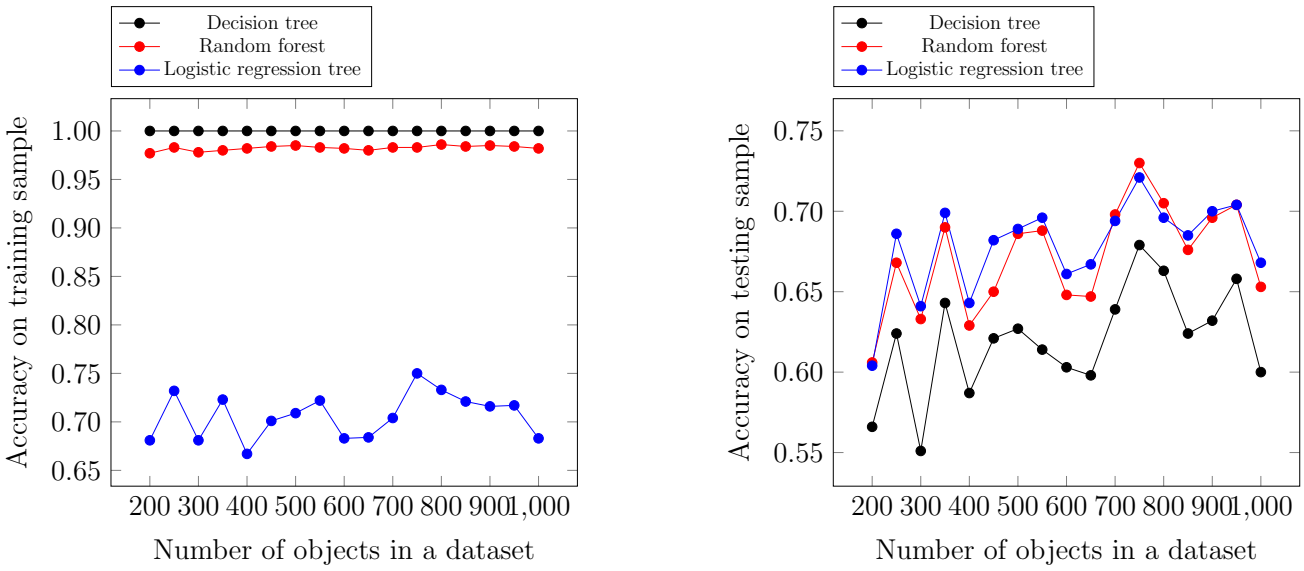


Figure 6: Accuracy of models on train and test data vs. size of a dataset

[16]The data is taken from the code (code block [8]) – refer to Appendix

As it can be seen in the figure 6, on the testing data the model performs worse than a decision tree and a random, because these two models have significantly more parameters. As a result, for example, decision tree reaches 100% accuracy as its depth is unlimited. However, when tested on the data which was not seen by the model in the training stage, the logistic regression tree shows better accuracy than the decision tree and similar accuracy to the random forest, which proves that it is an efficient model for reducing overfitting.

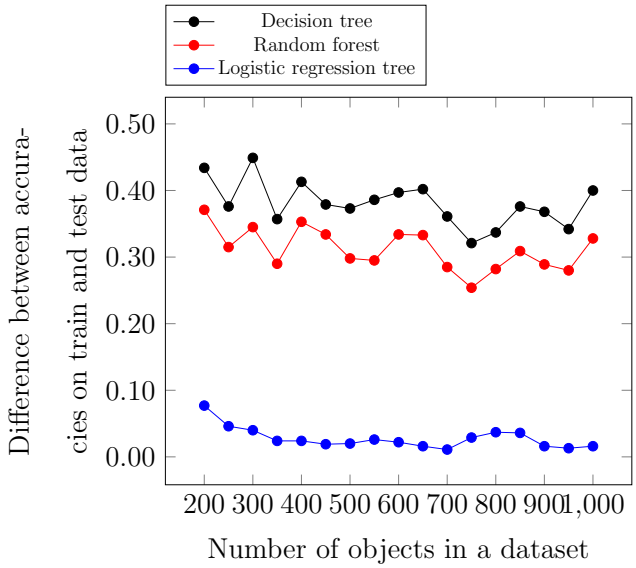


Figure 7: The difference between accuracies on train and test data vs. the number of objects in a dataset

The graph presented in the figure 7 shows how overfit the model is. Notice, that the decision tree is the most overfitted model, as its number of parameters is not restricted. Random forest uses a big number of complex models, and while each overfits, it does so in a different way, meaning that as an ensemble they reduce overfitting. This is also evident in the figure.

Logistic regression tree, on the other hand, does not overfit on small datasets: its performance on testing data is always extremely close to the performance on training data. This shows, that on small datasets the model is efficient.

However, it is important to understand that overfitting of decision trees can be reduced by imposing some limitations on these models (e.g. limiting the depth of a tree).

Such properties of a model are called **hyperparameters**. However, this optimization of hyperparameters requires some skills and experience to find a balance to make a model complex enough to succeed, while not making it too complex to overfit. On the other hand, logistic regression tree does not require hyperparameter optimisation.

5.2 Testing the genetic algorithm's success

The work of the genetic algorithm can be observed by monitoring best accuracy in the population with the increase of epochs passed (figure 8).

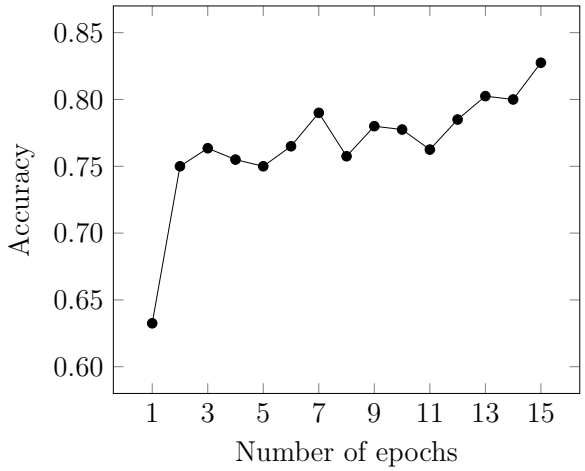


Figure 8: Accuracy vs. epoch[17]

The best accuracy in the population tends to grow from epoch to epoch, allowing to find the best permutation of classes in leaves without checking all the possible options.

5.3 Sources of error and further research

While the results prove the hypothesis that the complexity of a model and the size of a training dataset is directly related to the chance of overfitting, there are multiple factors that could impact the validity of these results.

1. There is no clear dependence between the accuracy and the dataset size, which shows that the structure of the data, not the size, is the main factor of the model

[17]Data for this graph is taken from the code (code block [6]) – refer to Appendix

accuracy. To better see the relationship between the dataset size and the accuracy, more datasets should be generated for each size.

2. The number of parameters of all models was dependent on the dataset size. The better approach would be to fix the complexities of the models used in the experiment in order to better see the dependence on the dataset size, not the number of parameters.

5.4 Comparison with literature

Existing research also demonstrates the validity of these results. For instance, an article "Comparing different supervised machine learning algorithms for disease prediction" analyses different articles that compare ML models for predicting diseases[18].

Logistic regression in 2 classes categorization performed not worse than a decision tree in 66% of the cases (2 of 3 articles), and performed better than a random forest in 71% of the cases (5 of 7 articles).

Moreover, even in two articles where the accuracy of logistic regression was worse, the difference between accuracies was negligible: 0.88 for logistic regression and 0.89 for random forest in one article, and 0.755 and 0.803 in another article. These results show that the big number of parameters in a random forest does not give it competitive advantage when training on small datasets.

As most of the datasets in this research were in the order of thousands of objects, the research proves the hypothesis that for small datasets more simple models can perform better.

6 Conclusion

The model successfully reaches the initial goal and helps to avoid overfitting via limiting its complexity. The results of the experiment successfully match the hypothesis, proving

[18]Shahadat Uddin et al., "Comparing Different Supervised Machine Learning Algorithms for Disease Prediction," BMC Medical Informatics and Decision Making 19, no. 1 (December 2019), https://doi.org/10.1186/s12911-019-1004-8

that the smaller number of parameters of a model reduces the chance of overfitting.

According to the figure 6 the accuracy of a less complex model (logistic regression tree) on the testing sample is better in case of small datasets. Moreover, the data shown on the figure 7 demonstrates that the difference between accuracies on train and test data is also smaller for a more simple model. Therefore, the initial hypothesis is confirmed.

The model presents a new view on machine learning algorithms, showing how multiple ideas, such as logistic regressions and decision trees, can be combined to create a new model.

7 Bibliography

Chabacano. “Diagram Showing Overfitting of a Classifier.” Wikimedia Commons, February 1, 2008. https://commons.wikimedia.org/wiki/File:Overfitting.svg#file.

Dietterich, Thomas G. “Ensemble Methods in Machine Learning.” Multiple Classifier Systems, 2000, 1–15. https://doi.org/10.1007/3-540-45014-9_1.

François Chollet. Deep Learning with Python, Second Edition. Shelter Island, Ny Manning Publications, n.d.

Galarnyk, Michael. “Logistic Regression Using Python (Scikit-Learn).” Medium, April 29, 2020. https://towardsdatascience.com/logistic-regression-using-python-sklearn-numpy-mnist-handwriting-recognition-matplotlib-a6b31e2b166a.

Hawkins, Douglas M. “The Problem of Overfitting.” ChemInform 35, no. 19 (May 11, 2004). https://doi.org/10.1002/chin.200419274.

Johnson, Becky. “Genetic Algorithms: The Travelling Salesman Problem.” Medium. Medium, November 15, 2017. https://medium.com/@becmjo/genetic-algorithms-and-the-travelling-salesman-problem-d10d1daf96a1.

Koohestani, Behrooz. “A Crossover Operator for Improving the Efficiency of Permutation-Based Genetic Algorithms.” Expert Systems with Applications 151 (August 2020): 113381. https://doi.org/10.1016/j.eswa.2020.113381.

Kumar, Paritosh. “Time Complexity of ML Models.” Medium, March 31, 2021. https://medium.com/analytics-vidhya/time-complexity-of-ml-models-4ec39fad2770.

Kurama, Vihar. “Random Forests: Consolidating Decision Trees.” Paperspace Blog, February 16, 2020. https://blog.paperspace.com/random-forests/amp/.

“Machine Learning - Number of Nodes in an Unpruned Decision Tree.” Cross Validated, n.d. https://stats.stackexchange.com/questions/114806/number-of-nodes-in-an-unpruned-decision-tree.

“Overfitting — Meaning of Overfitting by Lexico.” Lexico Dictionaries — English, n.d. https://www.lexico.com/definition/overfitting.

“Relationship between Number of Nodes and Height of Binary Tree.” GeeksforGeeks, January 15, 2018. https://www.geeksforgeeks.org/relationship-number-nodes-height-binary-tree/#: :text=Total%20number%20of%20nodes%20will.

“Sklearn.datasets.make_blobs.” scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html#sklearn.datasets.make_blobs.

Ying, Xue. “An Overview of Overfitting and Its Solutions.” Journal of Physics: Conference Series 1168, no. 2 (February 2019): 022022. https://doi.org/10.1088/1742-6596/1168/2/022022.

8 Appendix – code

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import KFold
import random
import itertools
```

[2]:
```
# Print iterations progress
def printProgressBar (iteration, total, prefix = '', suffix = '', decimals = 1,␣
↪length = 100, fill = '#', printEnd = "\r"):
    """
    Call in a loop to create terminal progress bar
    @params:
        iteration   - Required  : current iteration (Int)
        total       - Required  : total iterations (Int)
        prefix      - Optional  : prefix string (Str)
        suffix      - Optional  : suffix string (Str)
        decimals    - Optional  : positive number of decimals in percent complete␣
↪(Int)
        length      - Optional  : character length of bar (Int)
        fill        - Optional  : bar fill character (Str)
        printEnd    - Optional  : end character (e.g. "\r", "\r\n") (Str)
    """
    percent = ("{0:." + str(decimals) + "f}").format(100 * (iteration /␣
↪float(total)))
    filledLength = int(length * iteration // total)
    bar = fill * filledLength + '-' * (length - filledLength)
    print(f'\r{prefix} |{bar}| {percent}% {suffix}')
    # Print New Line on Complete
    if iteration == total:
        print()
```

The model

[3]:
```
class LogisticRegressionTree:
    leaves = np.array([]) # leaves of the tree (permutation of class labels)
    regs = np.array([]) # logistic regressions

    _cnt_classes = 0
    _cnt_classes_round = 0
    _regs_cnt = 0

    _segments = np.array([(0, 0)])

    '''

    --- PUBLIC METHODS ---

    These methods are used outside of the class by a user.
    They include methods for fittig, predicting, and scoring the accuracy.

    '''
```

```
    def fit(self, x, y, stop_acc = 0.9, use_genetic = True, population_size = 100,␣
↪epochs = 100, mutation_cnt = 1, percent_selected = 0.1,
            kfold_n_splits = 5, kfold_shuffle = True): #the main function which␣
↪sets up a model and performs training
        self._cnt_classes = np.amax(y) + 1
        self._cnt_classes_round = 1
        while(self._cnt_classes_round < self._cnt_classes):
            self._cnt_classes_round *= 2
        self.leaves = np.arange(self._cnt_classes_round)

        for i in range(2*self._cnt_classes_round - 2):
            self._segments = np.concatenate((self._segments, np.array([(0, 0)])))

        self._regs_cnt = self._cnt_classes_round - 1

        self._setup(0, 0, self._cnt_classes_round)

        if(use_genetic):
            self.leaves, self.regs, curve = self._fit_genetic(x, y, stop_acc,␣
↪population_size, epochs, mutation_cnt, percent_selected, kfold_n_splits,␣
↪kfold_shuffle)
            return {'curve': curve}
        else:
            self.leaves, self.regs = self._fit_brootforce(x, y, stop_acc,␣
↪kfold_n_splits, kfold_shuffle)
            return {'curve': None}



    def predict(self, X): # method for predicting classes for a set or objects
        return [self._predict_for_permutation(self.leaves, self.regs, x) for x in X]

    def score(self, x, y, kfold_n_splits = 5, kfold_shuffle = True): # method for␣
↪evaluating model's accuracy on test data
        cnt_all = len(x)
        cnt_correct = 0
        for i in range(cnt_all):
            if(self._predict_for_permutation(self.leaves, self.regs, x[i]) == y[i]):
                cnt_correct += 1

        return cnt_correct / cnt_all

    def get_params(self): # method that returns parameters of a model
        res = {}
        res['coefs'] = []
        for reg in self.regs:
            res['coefs'].append((reg.coef_ + [reg.intercept_]))
        res['leaves'] = self.leaves
        return res


    '''

    PRIVATE METHODS

    These methods are not supposed to be used outside of the model,
    they are used by public methods.
```

```
    '''



    def _fit_brootforce(self, x, y, stop_acc, kfold_n_splits, kfold_shuffle): #␣
↪training method, which checks all permutations
        max_accuracy = 0.
        leaves = []
        total = sum(1 for _ in itertools.permutations(self.leaves))
        cnt = 0
        #printProgressBar(0, total, suffix="Permutation " + str(0) + "/" +␣
↪str(total) + ", acc: " + str(0), length = 50)
        for permutation in itertools.permutations(self.leaves):
            acc = self._score_permutation(permutation, x, y, kfold_n_splits,␣
↪kfold_shuffle)
            if(acc >= stop_acc):
                print("\n ACCURACY REACHED \n")
                return leaves, permutation
            if(acc > max_accuracy):
                max_accuracy = acc
                leaves = permutation
            cnt += 1
            printProgressBar(cnt, total, suffix=max_accuracy, length = 50)
        regs = self._fit_permutation(leaves, x, y)
        return leaves, regs


    def _fit_genetic(self, x, y, stop_acc, population_size, epochs, mutation_cnt,␣
↪percent_selected, kfold_n_splits, kfold_shuffle): # training method, which␣
↪performs the genetic algorthm
        population = [np.random.permutation(self.leaves) for i in␣
↪range(population_size)]
        leaves = self.leaves
        mx_acc = 0.
        curve = []
        for i in range(epochs):
            scores = [(self._score_permutation(population[i], x, y, kfold_n_splits,␣
↪kfold_shuffle), i)
                      for i in range(population_size)]
            scores.sort(reverse = True)
            printProgressBar(i, epochs, suffix="Epoch " + str(i) + "/" +␣
↪str(epochs) + ", acc: " + str(scores[0][0]), length = 50)
            curve.append(scores[0][0])
            if(scores[0][0] >= mx_acc):
                mx_acc = scores[0][0]
                leaves = population[scores[0][1]]

            if(scores[0][0] > stop_acc):
                print("\n ACCURACY REACHED \n")
                break

            pool = []
            for k in range(min(population_size,␣
↪int(percent_selected*population_size + 1))):
                pool.append(scores[k][1])
```

```
            new_population = []
            for j in range(population_size):
                first, second = random.sample(pool, 2)
                first = population[first]
                second = population[second]
                ans = [-1]*len(first)

                l, r = random.sample(range(len(ans)), 2)

                for k in range(l, r + 1):
                    ans[k] = first[k]

                buffer = []
                for k in range(len(first)):
                    if(second[k] not in ans):
                        buffer.append(second[k])

                for k in range(len(first)):
                    if(ans[k] == -1):
                        ans[k] = buffer[0]
                        buffer.pop(0)

                for k in range(mutation_cnt):
                    i1, i2 = random.sample(range(len(ans)), 2)
                    ans[i1], ans[i2] = ans[i2], ans[i1]

                new_population.append(ans)

            population = new_population

        regs = self._fit_permutation(leaves, x, y)
        return leaves, regs, curve


    def _setup(self, cur, l, r): #initializing segments of leaves each regression␣
↪is responsible for
        self._segments[cur] = (l, r)
        if(l == r - 1):
            return
        m = (l + r + 1) // 2
        self._setup(2*cur + 1, l, m)
        self._setup(2*cur + 2, m, r)


    def _fit_permutation(self, leaves, x, y): # fitting regressions for a concrete␣
↪permutation
        return np.array([self._fit_regression(leaves, ind, x, y) for ind in␣
↪range(self._regs_cnt)])

    def _fit_regression(self, leaves, reg_ind, x, y): # fitting a particular␣
↪regression for a concrete permutation
        l, r = self._segments[reg_ind * 2 + 2]
        new_y = np.array([ int(k in leaves[l:r]) for k in y ])
        reg = LogisticRegression()
        reg.fit(x, new_y)
        return reg
```

```
    def _predict_for_permutation(self, leaves, regs, x): # making a prediction for␣
↪a concrete permutation
        l = 0
        r = self._cnt_classes_round
        cur_reg = 0
        while(l + 1 < r):
            if(regs[cur_reg].predict([x]) == 0):
                cur_reg = 2*cur_reg + 1
                r = (l + r + 1) // 2
            else:
                cur_reg = 2*cur_reg + 2
                l = (l + r + 1) // 2
        return leaves[l]

    def _score_permutation(self, leaves, x, y, kfold_n_splits, kfold_shuffle): #␣
↪scoring a permutation based on training data
        #print(leaves, '\n')
        kf = KFold(n_splits=kfold_n_splits, shuffle=kfold_shuffle)

        cnt = 0
        sum_ = 0.

        for train_index, test_index in kf.split(x):
            X_train, X_test = X[train_index], X[test_index]
            y_train, y_test = y[train_index], y[test_index]

            regs = self._fit_permutation(leaves, X_train, y_train)
            cnt_all = len(X_test)
            cnt_correct = 0
            for i in range(cnt_all):
                if(self._predict_for_permutation(leaves, regs, X_test[i]) ==␣
↪y_test[i]):
                    cnt_correct += 1

            cur_res = cnt_correct / cnt_all
            #print(cnt_correct, cnt_all, 'hello')

            cnt += 1
            sum_ += cur_res
        if(sum_ / cnt > 1.):
            print(leaves)
        return sum_ / cnt
```

Testing genetic algorithm's work

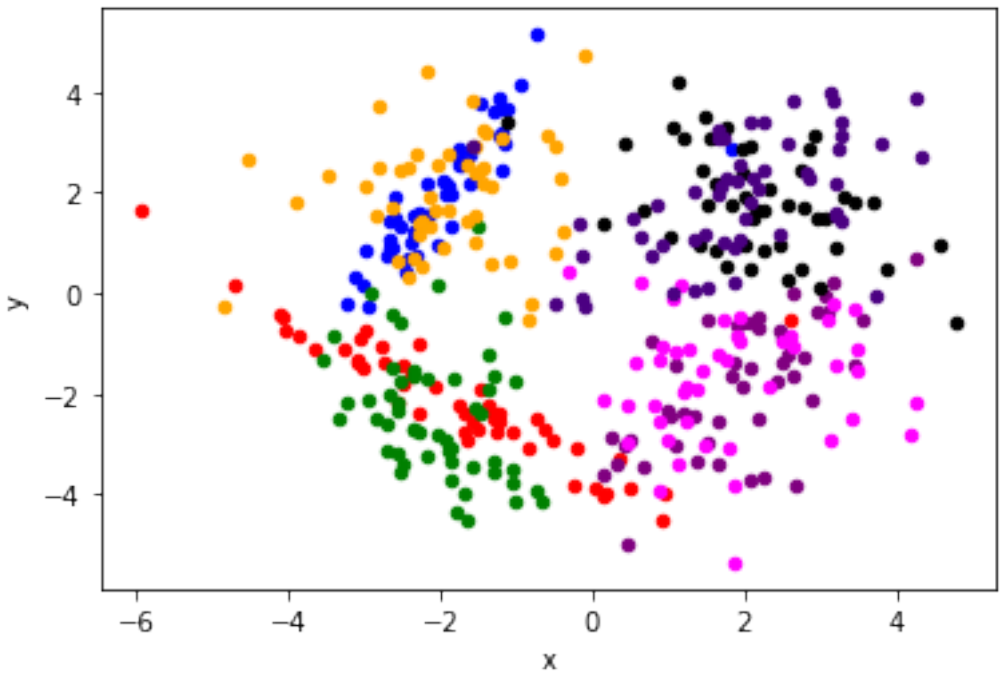
[4]:
```
from sklearn.datasets import make_blobs
from sklearn.datasets import make_classification
from matplotlib import pyplot
from pandas import DataFrame
```

Generating data for testing

[5]:
```
X,y = make_classification(n_samples=500, n_features=3, n_informative=3,
                          n_redundant=0, n_repeated=0, n_classes=8,␣
↪n_clusters_per_class=1,
                          class_sep=2, random_state=23)

X_train, X_test, y_train, y_test = X[:400], X[400:], y[:400], y[400:]

df = DataFrame(dict(x=X_train[:,0], y=X_train[:,1], label=y_train))
colors = {0:'red', 1:'blue', 2:'green', 3:'orange', 4:'purple', 5:'black', 6:
↪'magenta', 7:'indigo', 8:'beige', 9:'pink'}
fig, ax = pyplot.subplots()
grouped = df.groupby('label')
for key, group in grouped:
    group.plot(ax=ax, kind='scatter', x='x', y='y', color=colors[key])
pyplot.show()
```



Fitting the model on train data (15 epochs)

[6]:
```
model = LogisticRegressionTree()
curve = model.fit(X_train, y_train, population_size = 10, epochs = 15, use_genetic␣
↪= True, stop_acc = 0.97)
```

```
 |----------------------------------------------------| 0.0% Epoch 0/15, acc:
0.6325000000000001
 |###-------------------------------------------------| 6.7% Epoch 1/15, acc: 0.75
 |######----------------------------------------------| 13.3% Epoch 2/15, acc:
0.7625
 |##########------------------------------------------| 20.0% Epoch 3/15, acc:
0.7549999999999999
 |#############---------------------------------------| 26.7% Epoch 4/15, acc:
0.75
 |#################-----------------------------------| 33.3% Epoch 5/15, acc:
```

```
0.7649999999999999
 |####################------------------------------| 40.0% Epoch 6/15, acc:
0.79
 |########################--------------------------| 46.7% Epoch 7/15, acc:
0.7575000000000001
 |############################----------------------| 53.3% Epoch 8/15, acc:
0.78
 |################################------------------| 60.0% Epoch 9/15, acc:
0.7775000000000001
 |####################################--------------| 66.7% Epoch 10/15, acc:
0.7625
 |########################################----------| 73.3% Epoch 11/15, acc:
0.7849999999999999
 |############################################------| 80.0% Epoch 12/15, acc:
0.8025
 |################################################--| 86.7% Epoch 13/15, acc:
0.8
 |####################################################| 93.3% Epoch 14/15, acc:
0.8275
```

Testing the model's accuracy

[7]:
```
from sklearn.datasets import make_blobs
from sklearn.datasets import make_classification
from pandas import DataFrame
from sklearn.tree import DecisionTreeClassifier
from random import randint
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
```

[8]:
```
acc = []
for i in range(200 , 1001, 50):
    train_acc1 = []
    train_acc2 = []
    train_acc3 = []

    test_acc1 = []
    test_acc2 = []
    test_acc3 = []
    for j in range(10):
        X,y = make_blobs(n_samples = i, n_features=3, centers=4, cluster_std=5)
        cnt = 3*i//4
        X_train, X_test, y_train, y_test = X[:cnt], X[cnt:], y[:cnt], y[cnt:]

        decisionTreeClf = DecisionTreeClassifier()
        decisionTreeClf = decisionTreeClf.fit(X_train,y_train)
        test_acc1.append(metrics.accuracy_score(y_test, decisionTreeClf.
↪predict(X_test)))
        train_acc1.append(metrics.accuracy_score(y_train, decisionTreeClf.
↪predict(X_train)))

        rndForestClf = RandomForestClassifier(n_estimators = 10)
        rndForestClf = rndForestClf.fit(X_train, y_train)
        test_acc2.append(metrics.accuracy_score(y_test, rndForestClf.
↪predict(X_test)))
        train_acc2.append(metrics.accuracy_score(y_train, rndForestClf.
↪predict(X_train)))
```

```
        model = LogisticRegressionTree()
        curve = model.fit(X_train, y_train, use_genetic = False)
        test_acc3.append(metrics.accuracy_score(y_test, model.predict(X_test)))
        train_acc3.append(metrics.accuracy_score(y_train, model.predict(X_train)))

    acc.append([train_acc1, train_acc2, train_acc3, test_acc1, test_acc2,␣
↪test_acc3])
```

[9]:
```
avg_acc = []
for i in acc:
    avg_acc.append([(sum(x)/len(x)) for x in i])
```

[10]:
```
avg_acc
```

[10]:
```
[[1.0,
  0.9766666666666668,
  0.6806666666666666,
  0.566,
  0.6059999999999999,
  0.6040000000000001],
 [1.0,
  0.9834224598930481,
  0.7320855614973262,
  0.6238095238095238,
  0.6682539682539683,
  0.6857142857142857],
 [1.0,
  0.9782222222222222,
  0.6813333333333333,
  0.5506666666666666,
  0.6333333333333334,
  0.6413333333333334],
 [1.0,
  0.9797709923664121,
  0.7232824427480917,
  0.6431818181818183,
  0.6897727272727272,
  0.6988636363636364],
 [1.0, 0.9823333333333334, 0.6666666666666666, 0.587, 0.629, 0.643],
 [1.0,
  0.9839762611275965,
  0.7008902077151336,
  0.6212389380530973,
  0.6495575221238937,
  0.6823008849557523],
 [1.0,
  0.9847999999999999,
  0.7090666666666665,
  0.6272,
  0.6864000000000001,
  0.6887999999999999],
 [1.0,
  0.9827669902912621,
  0.7216019417475728,
  0.6144927536231883,
```

```
 0.6876811594202898,
 0.6956521739130435],
[1.0,
 0.9815555555555555,
 0.6831111111111111,
 0.6026666666666667,
 0.648,
 0.6613333333333332],
[1.0,
 0.9796714579055441,
 0.6839835728952772,
 0.598159509202454,
 0.6466257668711656,
 0.667484662576687],
[1.0,
 0.9834285714285714,
 0.7043809523809523,
 0.6394285714285715,
 0.6982857142857143,
 0.6937142857142857],
[1.0,
 0.9834519572953738,
 0.7496441281138789,
 0.6787234042553192,
 0.7297872340425532,
 0.7207446808510638],
[1.0,
 0.9861666666666666,
 0.7328333333333334,
 0.663,
 0.7045000000000001,
 0.6955],
[1.0,
 0.9841444270015698,
 0.7210361067503923,
 0.6239436619718309,
 0.6755868544600938,
 0.6854460093896714],
[1.0,
 0.985037037037037,
 0.7158518518518519,
 0.6315555555555555,
 0.6964444444444445,
 0.6995555555555556],
[1.0,
 0.9837078651685391,
 0.7169943820224719,
 0.6579831932773109,
 0.7042016806722688,
 0.7042016806722688],
[1.0,
 0.9816,
 0.6834666666666667,
 0.6003999999999999,
 0.6531999999999999,
 0.6676000000000001]]
```