Introduction

Machine Learning plays a crucial role in data inference and in finding models that, when fed with sets of observations, allow to make predictions on new data.

The power of this tool is the applicability to many different tasks, from predicting numerical values to assigning categories to observations according to some features or characteristics.

Images classification is one of the more interesting scopes: training the algorithm with pixels’ values and the correspondent labels, it will output predictors capable of recognizing and classifying the pictures, assigning them the correct tag.

Random Forest algorithm, built assembling many tree predictors, allowing for some bias in the model in order to reduce the variance and the overfitting of the individually taken Decision Tree classifiers, performs really well when tuned with optimal parameters’ values.

This report shows the analysis made after having implemented from scratch the Random Forest algorithm on a training set and a test set storing pixels’ observations associated to 10 different labels.

First of all an overview about the theoretical framework of the algorithm is presented, secondly the importance of the hyperparameters’ tuning is highlighted, then, after a brief introduction of the dataset, the code is commented in depth. Finally the implementation on the dataset is explained and the accuracy of the model is analyzed, also focusing the attention to the previously mentioned hyperparameters.

Random Forest

The Random Forest is an ensemble machine learning method that allows to perform both classification and regression on datasets. It uses Decision Trees as building blocks to construct a more powerful predictive model.

Decision Tree classifiers split the dataset according to some rules. They are characterized by a root, that is the node where starts the splitting of the dataset, each internal node represents a test on a feature and each leaf is a different class label. If an observation has a value for that feature greater than a predetermined threshold, then the observation is conduced to a certain child node, otherwise, if it is smaller, the observation is routed to another child node and this process is iterated until the observation does not reach a leaf node. Therefore, depending on the tests’ answers, the paths will be different and the observations are routed to different labels.

The feature and the threshold on which the data are split on are decided according to the best split criteria: all the possible splits at each step are considered in order to find the one that has the highest information gain and so the largest improvement in the entropy value for the node. The entropy is a measurement of the homogeneity or purity of the node: if in a node there are only observations with the same label, then that node is said “pure”, since it does not contribute to the training error.

Decision Trees work well for the data they are built on, but not equally well when they have to make predictions on new data. Indeed these classifiers are prone to overfit the data: their training error is usually really small, especially if they are deep and if they are grown until a pure node is reached. They are an inflexible instrument that learns the data by fitting it closely and so it is led to learn all the information coming from the data, also the misleading or noisy parts. Moreover, Decision Trees are characterized by variability and instability since also a small change in the data or in the parameters leads to a big change in the result. Of course Decision Trees have also many advantages as the easy interpretability and representation and they work pretty well on stable contexts.

In order to combine the advantages of Decision Trees and the need for more stable models, with more predictive power, Random Forest has been introduced: starting from the simplicity of the decision trees, this ensemble method allows for more flexibility and so improves the model’s overall accuracy and stability.

The way the Random Forest gains in flexibility is actually adding some randomness in the process:

* the sample on which the random forest is built on is not the original one, but a bootstrapped sample: each tree in the forest is learnt on a sample selected randomly and with replacement from the original one;
* the Random Forest randomly selects only a subset of the features for each splitting of a node.

The ratio behind this method is combining different predictors in order to gain information, but also decorrelating them in order to avoid misleading results due to outliers or anomalies. Building different trees using these random elements leads to a larger variety of information and consequently to improvements in robustness when making predictions.

Since about 1/3 of the dataset does not enter in the Random Forest due to the bootstrapping, we are artificially including some bias in the model but reducing the variance so that the result is a more accurate classifier.

So the main advantage of using Random Forest is the reduced overfitting, while the main disadvantage is the complexity of the model that results in a large computational cost.

Hyperparameters tuning

Even if Random Forest is more robust than Decision Trees and so it is also less depending by hyperparameters, in order to improve the performance of this ensemble method, a tuning operation is needed.

The parameters that have to be fixed are the maximum depth, the minimum sample split, the number of randomly selected features to consider when splitting the nodes of the trees, the number of trees in the forest and the observations in the bootstrap sample.

The maximum depth is the longest path from the root of the tree to the leaf node. In this case the maximum depth has been set to 10, so the trees in the forest won’t grow over the 10th node to split. This parameter’s tuning is needed to handle overfitting. Indeed as the maximum depth increases, the performance over the training data is improved and so the training error is reduced, however it is not rare that the test error has a different behavior. It tends to decrease in a first moment, but then it starts increasing and this means the model is overfitting.

The minimum sample split is the minimum number of samples needed in a node to split it further. The default value has been set to 2, that means it is required that a node that is not pure has to split again if there are still 2 observations with different labels in the node. Increasing this value, the model is prevented from overfitting, but when too large values are established, the model starts to underfit since the information loss is too high with respect to the gain in generalization.

The number of features to pick among, when determining the split, are usually a subset chosen from the total number of features equal to the square root of the total, since it has been proved that over this number, the model starts to overfit. In this dataset the total number of features is 784 and has been chosen a subset of 25 randomly selected features.

The number of estimators when increased leads to a more accurate result since we are adding the overall information given to the algorithm, but at the same time it increases also the time complexity of the model. So a balance is needed and the analysis at the end will show how this hyperparameter affects the test error and so the accuracy of the model.

Another parameter that helps reducing the computational complexity of the Random Forest is the number of observations in the bootstrap sample: selecting only a subsample of the original dataset for the bootstrapping allows to decrease it without losing too much in performance.

Other parameters that could be tuned, but that are not considered in this specific case, are the maximum leaf nodes and the minimum number of samples in a leaf. The first one restricts the growth of the tree by checking the number of leaf nodes after a split: if there are more leaves than the specified number, then the algorithm stops. If this constraint is too small the tree will underfit the data, on the other hand, if it is too large it will overfit.The second one specifies the minimum number of samples that we require to be present in a node to be considered as a leaf node. Also this hyperparameter helps preventing the overfitting problem as the parameter value decreases.

The FashionMNIST Database

This database consists of a training set of 60.000 observations and a test set of 10.000 observations. Each observation has 784 features and an associated label. The features of the observations represent pixels of 28x28 images and so the value they can assume goes from 1 to 255. Each image is classified with one of these 10 labels:

* 0 for T-shirt/top
* 1 for Trouser
* 2 for Pullover
* 3 for Dress
* 4 for Coat
* 5 for Sandal
* 6 for Shirt
* 7 for Sneaker
* 8 for Bag
* 9 for Ankle boot

*(Source:* [*https://research.zalando.com/welcome/mission/research-projects/fashion-mnist/*](https://research.zalando.com/welcome/mission/research-projects/fashion-mnist/)*)*

The aim of the algorithm is classifying the observation in one of these labels and so learning the machine with the pixels and the relative classifications in order to predict the content of grey scale pictures.

Algorithm from scratch

In this code can be distinguished three main parts: the Decision Tree algorithm, the Random Forest algorithm and the analysis of the accuracy when changing the number of estimators in the forest.

Decision Tree algorithm

The tree is grown starting from the root and selecting at each node the best split according to the information gain that it will bring to the predictor. The main loop seeks for all the possible features and all the possible thresholds in each feature in order to obtain all the possible splits and the corresponding information gains. The tree is built recursively by looking for the best split at each node until a stopping criteria is met: for example the trees can be cut at some maximum depth, or the growth can be stopped when inside a node there is a certain number of samples among which computing the majority vote, or again it can be stopped if inside a node there is just one possible label. Once reached a leaf node according to one of these criteria, the most common label in the node defines the class associated with that node.

Then the model has been fed with the training set, in order to enable it to make proper predictions, accordingly to what learned in the previous step. In particular for each observation the algorithm looks at the values of each feature and goes down along the tree by moving toward left or right according to the thresholds defined when the tree has been built. When a leaf node is reached, the label associated to that node is assigned to the observation and this occurs for each observation so that we obtain the predictions for that sample.

An in depth explanation of the algorithm follows.

Firstly an helper function called *scaled\_entropy* has been created: it allows to compute the entropy of a set of labels’ values using the traditional formula. We used the bincount method from Numpy in order to count over the occurrences of each class and then, dividing each occurrence by the total length of the labels’ vector, we get the array of probabilities used to compute the entropy.

Another helper function has been defined in order to find the most common label in a node: the *most\_frequent\_label* function exploits the Counter method from collections in order to take the index of the label that has the greatest number of occurrences, then it selects only the first element of the first tuple in the list of frequencies. This function will be helpful both for the Decision Tree classifier and the Random Forest, since for both the value assigned to a leaf node is computed by majority vote, among the labels in the node in the first case, or among the labels output by the single predictors in the forest.

Also an helper class, the *Node class*, has been created, in order to store information for the nodes. In the middle of the tree, a node is characterized by the best split feature and the best split threshold and the resulting left child node and right child node. Instead a leaf node is associated to the most common label for the node.

The *\_\_init\_\_* method takes as arguments the feature, the threshold, the left child node and the right child node and the value associated to the node.

Another helper function that has been defined is the *leaf\_node*, that returns the Boolean ‘True’, if the value in the node is not none and so if we are in a leaf; this is useful when, after the tree has been grown, each observation is run along the tree and eventually assigned to a leaf node and so a label.

Then the *DecisionTree Class* stores the methods defined to grow and predict the classifier.

We define the *\_\_init\_\_* method for this class taking arguments the minimum number of samples required to further split the tree, the maximum depth of the tree and the number of features. In this case we set as default values 2 for the minimum number of observations in a node required for a split, 10 for the maximum depth of the tree and 25 for the number of features to consider in the greedy search for the best split and the best threshold values. The choice to set 25 as default value has been made in order to reduce the computational cost of the classifier; since there are 784 features, the algorithm could look over the whole set of features and, for each of these, it would look over all the possible features’ values in order to find the best split. In order to save computations, the greedy search is made on a subset of 25 features selected randomly and this is actually one of the two ways in which some randomness is introduced in the Random Forest. In the \_\_init\_\_ method is also stored the root, with none value at the beginning, it will be needed when the observations will traverse the tree for making predictions.

Then, the *split* function has been defined, taking as input the columns of the features’ matrix and a threshold value for splitting the node. It returns left indexes as the indexes of those observations that have values for a feature smaller or equal than the threshold, and right indexes as all the values greater than the threshold.

The *build\_tree* method takes as parameters the features’ matrix X, the labels’ array y and the depth of the tree that is set to 0 at the beginning. In this function, the stopping criteria have been set: they stop the growth of the tree when met, otherwise it continues. So if we reach the maximum depth, or if there is only a class label in the labels’ array, or again if the number of samples is less than the minimum required to split a node, then the function returns the value associated to this node that is the most common label in the node. On the other hand if at least one of the stopping criteria has not been met yet, we continue growing the tree by selecting randomly the indexes for the features’ subset we defined in the ‘n\_feats’ parameter. Once selected the features’ indexes to consider, we can start the greedy search among all the possible values that the indexed features randomly selected can assume. As previously mentioned, the criteria to select the thresholds values’ is the greatest value for the information gain, so specific functions have been defined in order to facilitate the research: the Information\_Gain function and the Best\_criteria function.

The *Information\_Gain* function takes as inputs the labels array, all the features’ columns and a split threshold. The information gain is defined as the difference between the parent node’s entropy and the weighted average of the children nodes’ entropy. In particular we have to compute the parent node’s entropy using the previously defined function, then a split is generated using the threshold value set as parameter. Then it computes the weighted average of the children’ entropy by dividing the numerosity of the labels in the left and in the right side after the split by the overall length of the labels’ array; these two ratios are set as weights for the left node entropy and the right node entropy.

The *Best\_criteria* method takes as input the X features’ matrix, the y labels’ array and finally the randomly selected features’ indexes. For each randomly selected feature, the function takes each value that it assumes and computes the information gain that splitting the node at each of these thresholds would provide. If the information gain is greater than the best gain we set as 0, then the new best gain is the actual information gain, the best split is the actual feature index associated to this information gain and the best threshold is the correspondent value assumed by that feature. The best gain has been set to 0 in order to consider each split, also the smallest possible: indeed if a parent node has maximum variability of the labels – as it occurs at the beginning for this dataset – then the entropy takes its greatest possible value; if after the split, the proportion among the labels is the same and so there still is maximum variability in the node and same level of entropy, then the information gain will be 0. The function returns the values of the split index and the split threshold.

Once defined the function to operate the greedy search according to the best criteria, the tree can be grown. A split is generated using the split function, but taking as parameters the best feature and the best threshold, that are respectively the split index and the best threshold returned by the Best\_criteria function. Then the function proceeds with a recursion where the build\_tree function is implemented again on the observations that are on the left child generated by the splitting of the node – so those observations having best feature’s value smaller or equal than the best threshold, taken with their correspondent labels and setting the depth of the tree one step greater, until the maximum depth is reached and the growth is stopped. Another recursion do the same thing on the right hand side of the split. So each node is split according to the best feature and the best threshold for that specific node and left and right children are nodes defined by the recursions until a leaf node is found.

Then the *fit* method is required to apply the rules defined above to a certain matrix X of variables and a certain target vector. It takes as inputs the features’ values and the labels’ values and it starts growing the tree from the root - that initially is Nonetype - applying to the root the build\_tree function.

The last thing for the DecisionTree class is to define the predict function.

First the *along\_tree* function has been defined: it takes a single observation and a node and it checks if a stopping criteria has been met verifying if the node is a leaf node. If this is true, then the function returns the value associated with the leaf node. Otherwise if the value of the observation for the feature at the left node is smaller or equal than the threshold we stored for that node, then we use again a recursion to apply the along\_tree method to the left node. If instead the value of the observation for the features in the right node is greater than the threshold, we apply he function to the right node features. We recursively continue going down along the tree until we reach a leaf node and the along\_tree function returns a value associated with that leaf.

The *predict* function simply implements the along\_tree function for all the observations in the data matrix X given as input.

Once created the single Decision Tree classifier, a Random Forest can be built to ensemble a set of classifiers and reduce the overfitting that can affect them when taken individually.

Actually, in the Decision Tree grown by this algorithm the overfitting is reduced by the fact we don’t use all the features to split a node, but only a random sample of 25 features. Anyway we introduce another element of randomness with the Random Forest algorithm: the sample on which each classifier in the forest is built on, is a random subsample of the original one, taken with replacement.

Random Forest Algorithm

First of all a *Random Forest Class* has been created together with an function for this class.

The *Bootstrap\_sample* function takes as input the matrix of the features’ values X and the array of the labels y and it returns the randomly chosen indexes for a bootstrapped sample from X and y. In particular, it should perform a random choice among the overall indexes of the sample with replacement, so that it creates a new sample with the same dimensions and the same features’ value for each observation, but where the observations can appear also more than once and in different order than before. However, in order to reduce the computational cost, in this case the bootstrapping is used to perform a random selection with replacement on a subset of 6000 observations from the original X matrix.

The *\_\_init\_\_* method initializes the class and takes as arguments the number of trees we want to populate the forest and all the parameters needed in the Decision Tree \_\_init\_\_ function and so the minimum number of observations required for the split, the maximum depth of the trees and the number of features to consider when splitting a node, for more randomness and in order to save computation. In this function all the parameters’ values are stored, but also an empty list that we want to fill with the trees of the forest each time they are grown.

Also in this case the *fit* method takes as input the matrix of the features’ values X and the array of labels y. Up to the number of trees in the forest that have been chosen in the \_\_init\_\_ function, the fit method builds trees with the previously defined build\_tree function in the Decision Tree class and stores them in the list “ trees” that is empty at the beginning. The trees that are build with the Random Forest class fit method are actually different from those of the single DecisionTree fit method, since we use the bootstrapped sample obtained with the bootstrap\_sample function defined above in order to add some randomness in the forest.

To make predictions the *predict* method is defined: it takes as input the observations we want to classify and it returns the predicted labels of these observations according to the classifier defined in the fit method. For each tree learner in the previously filled list of bootstrapped trees, we want to make a prediction of the labels using the predict method of the DecisionTree class. The majority vote is computed among the array of predictions output by each tree. The result is a Numpy array containing many arrays, one for each tree classifier, storing the predicted labels for each observation; we want to take the majority vote among the first element of all the arrays, then among the second element of all the arrays and so on. So the axis of the Numpy array are swapped to obtain a new object containing arrays storing the values of the predictions for each observation. Now we can compute the majority vote: for each array inside the tree\_preds array, we take the most frequent label using the same function defined for the DecisionTree class and the final output is a new vector containing the list of the most common labels predicted for each observation in the input matrix X.

In order to understand which is the performance of the algorithm, a method to calculate the error, or equivalently the accuracy, is needed. The function *accuracy* starts assigning value 0 to the initial accuracy and for each correct prediction adds 1 to the accuracy value, then the result is scaled dividing by the length of the labels’ array. Once obtained the accuracy value, the error can be computed as its complementary value.

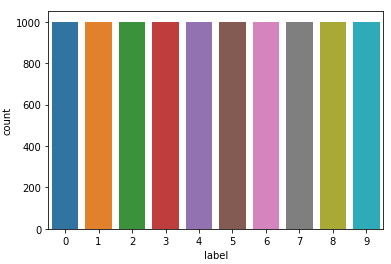
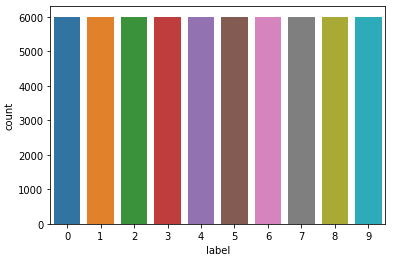
Algorithm on the FashionMNIST Dataset

Before proceeding with the application of the Random Forest algorithm to the dataset, we perform some data analysis and data visualization. Firstly the presence of missing values in the data has been checked and once verified their absence, we can show the images represented by each array of pixels records.

The left side square is the training set collection, while the square on the right is the test set.

Then we check the distribution of the observations among the 10 classes using the *countplot* function from seaborn to show that the observations are perfectly balanced among the classes, both in the training set (on the left) and in the test set (on the right).



Now we can proceed with the real implementation of the algorithm on this dataset.

The Random Forest algorithm built so far has been applied: the model has been fit on the training data and the predictions for the training labels have been compared to the real values, in order to check the accuracy of the model on the same data it is built on. Without tuning the parameters’ values, the training error computed as misclassification rate on the model with a bootstrapped subsample with size 6000 and number of estimators set to 150, is 17%. The model is not perfect, as we could imagine, since many adjustments have been taken with the main purpose of avoiding overfitting.

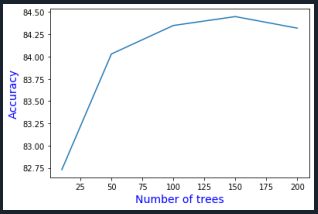
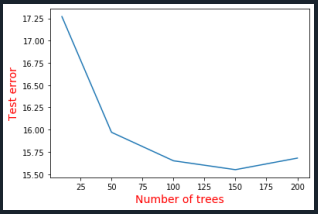
In order to check if the bias introduced in the model actually reduce the test error, the analysis has been focused on the test set and we observed how the predictions output by the Random Forest differ from the test set labels.

For the first time the Random Forest has been implemented by using the whole dimension of the sample in the bootstrap function as number of observations to randomly select. In that case the accuracy of the test set is 85%.

However, for all the other implementations a subset of 6000 observations has been chosen, so only the 10% of the training set to train each tree in the forest, in order to reduce the time needed for the process and so analyze the path of the test set when changing the number of trees in the forest. The result with 150 estimators – the same number we used for checking the training error – is an accuracy of 84.45%, so an acceptable reduction in precision for a great reduction in time-complexity.

As expected, the Random Forest results an accuracy value larger than the accuracy of the single tree predictor: indeed if the Decision Tree is run for the prediction of the training set labels, it will output a model with about the 82% of accuracy and so a training error of 18%. The test error for the single classifier is instead 21%.

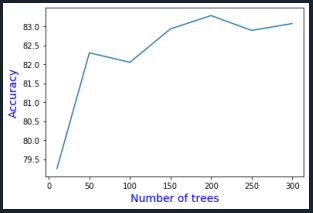
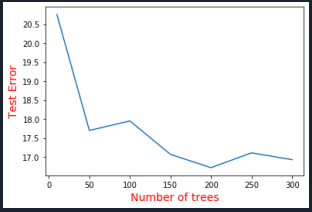
Using a bootstrapped subsample of 6000 observations and iteratively running the Random Forest algorithm, the pattern of the test set when changing the number of predictors in the forest can be analyzed. For the first time, only 10 trees have been included in the forest and the level of the accuracy on the predictions of the test set is 82.73%. Then we proceeded by repeating this method for a range of estimator from 50 to 200 with steps of 50 and we plotted the results.

It is clear that 150 is the optimal number of estimators in the forest since the line reaches a peak in correspondence of this value and the accuracy level of the predictions is 84.45%.

Of course we could have tested many more different values for the number of estimators, but the choice has been made accordingly to the computational cost already high, considering that for 200 estimators the accuracy slightly decreases again and also considering that the value seemed to be pretty much stable since 100 estimators.

In order to try if the result would have changed, using as bootstrapped sample of 1200 observation randomly selected among the 60000 (only the 2% instead of the 10%), we selected for the number of trees in the forest a range of values up to 300, again with steps of 50. Computing the accuracy for each new model we obtained that these predictors are less accurate than the previous ones, even those with more estimators in the forest. Also in this case, when using more than 200 predictors, the accuracy starts to decrease.

The peak, reached in correspondence of 200 estimators, is the maximum level of accuracy – for the range we considered – of 83.28%. So again the result is slightly worse than the one we obtained for a larger bootstrapped subsample. The best model is then the ensemble method taking as parameters a number of estimators equal to 150 and a bootstrap subsample of the 10% of the total observations.

Results

This analysis has shown how Tree predictors and in particular Random Forests can be useful in classification tasks. We have tried to build a model for labeling pictures according to the pixels’ values. Firstly we tested the model with the single Decision Tree and then we improved its performance using the Random Forest.

Hyperparameter tuning is crucial for the accuracy of the model and in particular we kept fix the maximum depth of the trees in the forest and the number of features to consider when splitting a leaf. On the other hand we have tried to optimize the values of the number of estimators seeking a compromise between the increase in the memory footprint due to the complexity of the model and the decrease of the test error and so the improvement of the accuracy level.

The resulting model with a bootstrapped subsample of the observations and 150 tree predictors in the forest led to an accuracy of the 84.45% of images correctly classified.

It is important to clarify the awareness that even better models could have been reached if the total number of observation were included in the bootstrapping process instead of a subsample of the training set and if also the other parameters’ values would have been optimized.