Random Forests : object-oriented design and implementation in Python

Object-oriented programming, MatCAD

Joan Serrat

Abstract

This practical is about doing a modular and extensible design and implementation of random forests. This was an extremely popular and effective method for classification in machine learning before this field was taken by storm by deep neural networks a few years ago. Nowadays it is still a good solution for problems where one can easily handcraft *features*.

We will start with a brief background on classification and then explain this method. It is important that you understand the context because you will have to not only design and implement the classifier but also the classes necessary to train and test it through a common strategy called k-fold cross–validation. The reward is a versatile implementation of a classifier that readily achieves at least 90% accuracy on MNIST, a well known dataset of handwritten digits.

To get the referenced Python scripts go to the `snippets` folder at https://v2.overleaf.com/read/dfjyxpnybjgp.

1 Introduction

In machine learning, the goal of classification is to assign a label or class identifier to certain input data. The most common representation for "data" is a vector of *features*, that is, a vector of n dimensions where each coordinate represents a certain measurement, potentially relevant. Relevance means discriminating power, that is, capacity to differentiate feature vectors from one class to another.

The typical workflow in a (pre deep learning) classification problem is the following:

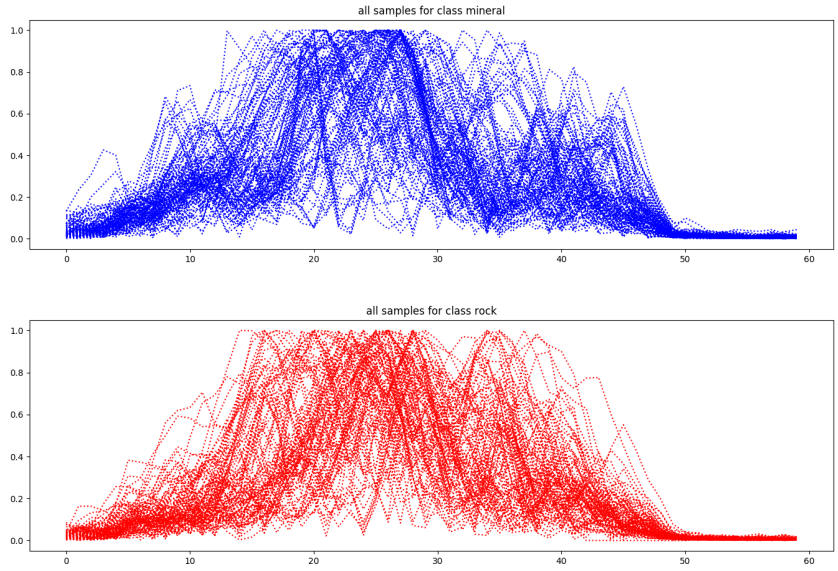

Figure 1: All 208 samples of the "Sonar" dataset, with two classes, data being vectors of 60 components. Source code at `sonar_figures.py` and the dataset itself at `sonar.all-data.csv`

1. **Build or get a dataset**. A dataset is made of samples of the different classes to distinguish, the more samples and the more balanced among the classes, the better. A sample has two parts: some data and a label. The data part can be a scalar, a vector of values, an image, a video .... The label is the identifier of one of the classes (like an integer number or a string). The set of labels of a dataset is also called the *groundtruth*. For websites gathering all kinds of datasets, see for instance the UCI repository, the Scikit-learn example datasets, Kaggle, or the long list in this Github repository.

   In this practical we are going to work with at least two datasets. One is from UCI and is called "Sonar" (figure 1). It's not of much interest apart from the fact it is used to illustrate a certain implementation of random forests that we will take as a starting point. See the script `sonar_figures.py` for a description. The second is MNIST, a famous dataset among deep learning beginners (figure 2).

2. **Feature extraction**. The data part of a sample is not normally the best for classification, so one can transform it into a vector of *features*, a representation from which we expect a better performance from the classifier. For instance, in MNIST dataset of figure 2, the data of a sample

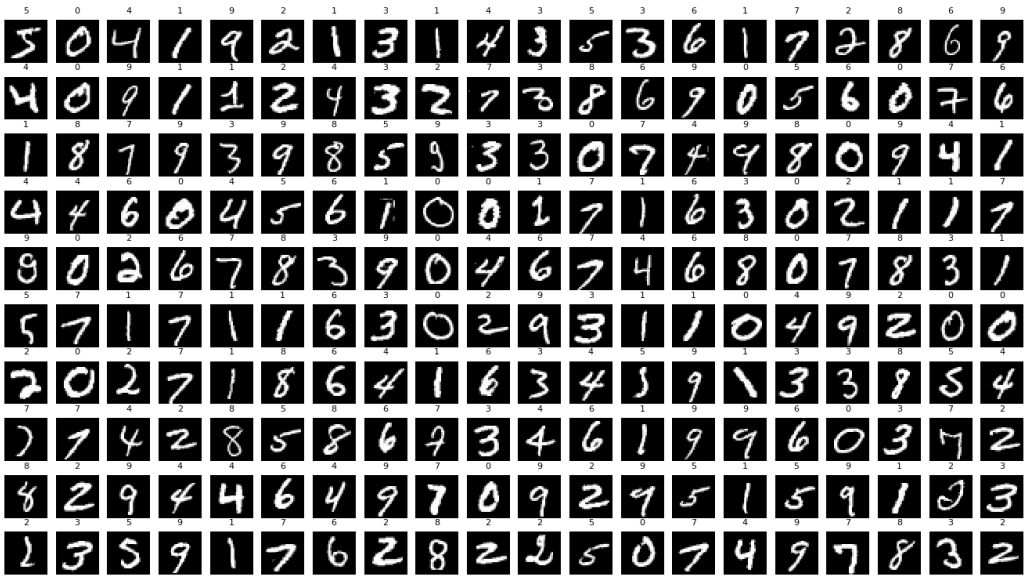

Figure 2: 200 samples for the ten classes of the MNIST dataset. It consists of 70,000 samples, 60,000 for training and 10,000 for testing, each being an image of size 28×28 pixels (or if flattened, a vector of 784 components), 256 grey levels. Source code at `mnist_figures.py` and `mnist.py`. The dataset in dictionary form, saved with pickle, is at http://www.cvc.uab.es/people/joans/tmp/mnist.pkl.zip

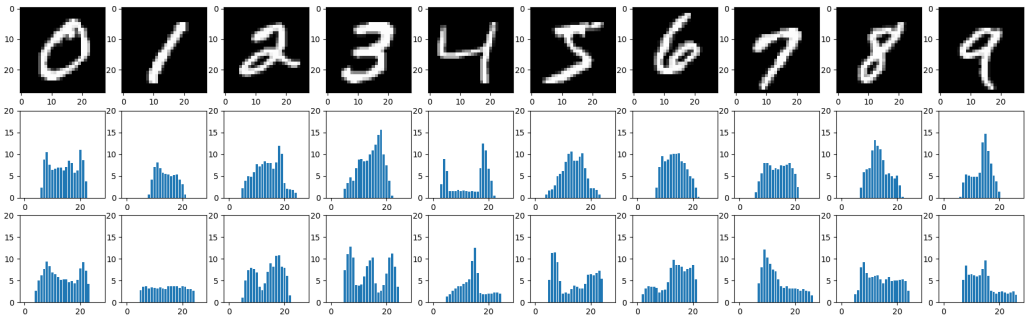

Figure 3: Vertical (middle row) and horizontal (bottom) projections of some digits. Source code at `mnist_figures.py` and `mnist.py`

is the value of its 784 pixels. We may suppose that a classifier can perform better if, instead of using these values, it works with concatenation of the vertical p_x and horizontal p_y projections of the image (figure 3):

$$\begin{aligned} p_x^j &= \sum_{i=0}^{27} x_{i,j},\ j = 0 \ldots 27 \\ p_y^i &= \sum_{j=0}^{27} x_{i,j},\ i = 0 \ldots 27 \\ x &= [\, p_x^0,\, p_x^1 \ldots p_x^{27},\, p_y^0,\, p_y^1 \ldots p_y^{27} \,] \end{aligned}$$

No matter how are they written, this new representation for ones will normally be quite different from the representations for zeros, eights, threes etc. though not so much for sevens, and similarly for the other classes. Feature extraction means thus to compute a feature representation x for each sample and put all the representations into a matrix X (one row per sample), and corresponding class labels as an integer into a vector y. For m samples, n features and c classes we'll have an X of size $m \times n$ and y of length n, with $y_i \in \{0, 1 \ldots c-1\}$.

Feature extraction can improve classification accuracy, but we won't spend time trying to find good features for our datsets. Instead, we are going to directly work with the raw data of each sample as features. Thus, for Sonar and MNIST we will have 60 and 784 features per sample, respectively.

3. **Training**. To train a classifier means to learn the parameters of some function/algorithm that maps a new feature vector into a valid class number, $f : \mathbb{R}^n \to \{0, 1 \ldots c-1\}$. For a certain dataset, it is normal practice

fold	indices train	indices test
1	0, 1, 2, 3, 5, 7, 8, 9, 10	4, 6, 11
2	0, 1, 3, 4, 5, 6, 7, 9, 11	2, 8, 10
3	0, 2, 3, 4, 5, 6, 8, 10, 11	1, 7, 9
4	1, 2, 4, 6, 7, 8, 9, 10, 11	0, 3, 5

Table 1: Cross-validation folds for 4 folds and 12 samples. The number of test samples of each fold is $12/4 = 3$. We have randomly permuted the list of indices $[0 \dots 11]$ before making the folds, just in case the original samples were not totally independent but somehow grouped. Doing this is important for the Sonar dataset.

(see next point) to split samples into two subsets, train and test. It is the train set that is used for learning, that is, a subset of the rows of X and y, say 70% samples for training and 30% for testing, randomly chosen.

4. **Assessment**. Compute a measure of how accurate are the labels predicted by the classifier. In order to get an unbiased assessment, and get an idea of the generalization ability of the classifier, the samples used for evaluation must be different from those used for training. Otherwise, it is too easy for the classifier because it has already "seen" the samples you want to label, and somehow, may remember the solution. So, before training you randomly split the samples into train and test subsets. The simplest evaluation measure is the ratio of right class predictions over total number of predictions (therefore, number of samples in the test set). This is a relative measure: in a c classes problem, just producing a uniform random label gives $1/c$ accuracy (so, 50% accuracy in a problem with two classes).

It may happen that our dataset is small, or simply that we would like to have more test samples to better evaluate the accuracy of the classifier. However, we have no time or no means to get more samples. A solution is to repeat the train-test split of samples several times, say k times, with the available data such that the test sets are disjoint. Each time run the train and evaluation processes, and at the end average the obtained accuracies. This is know as k-fold cross validation, being a fold each one of these splits. Suppose we have 12 samples and want to perform 4-fold cross-validation. Then we train and test with the splits of table 1, where numbers are the indices of samples.

For an extended discussion of the former concepts you can read any machine learning textbook, but a more practical source is the documentation

and tutorials of Scikit-learn, a very well designed and documented machine learning library for Python.

2 The random forest algorithm

A decision tree is a binary tree where each non-leaf node contains a feature index and threshold value related to that feature. Suppose a certain sample x has to be classified by an already trained decision tree. x enters into the root node, which has index i and value v. If $x[i] < v$ then x is sent to the left child, else to the right one. The process continues until x reaches a leave node. Leave nodes just store the class label to be assigned.

While classification is easy to explain, training is not so simple. These two video tutorials explain it very well, one is on how to train a decision tree, in Python, from scratch, the other is on decision trees in general. They are less than 10 minutes long, play them and then come back.

Training means to decide two things. The first is which feature index $f \in \{0, 1, \dots m-1\}$ with m number of features, and value v to store at each non-leave node, and 2) whether a node should be a leave or not. To do 1) one takes the set of s samples $(X_i, y_i), i \in I = \{1 \dots s\}$ entering the present node, and evaluates a certain criterion for all the features f and all the corresponding values $v \in X_i[f], i \in I$ (in all the samples). The goal is to minimize the criterion with respect to f and v, which is a measure of how heterogeneous are the labels of the two subsets resulting from dividing $y_i, i \in I$ into $y_j, j \in J$ and $y_k, k \in K$ such that $J = \{j \in I \mid X_j[f] < v\}$ and $K = \{k \in I \mid X_k[f] \geq v\}$. The less diverse are $\{y_j, j \in J\}$ and $\{y_k, k \in K\}$, the better are f and v. One way to measure it is through a so called Gini index. Once f and v are found, $(X_j, y_j), j \in J$ are sent to the left child node and $(X_k, y_k), k \in K$ to the right, and each node is trained the same way.

The second aspect to decide is when to split a node —make its two children—, or make it a leave node. If we have reached a maximum tree level, or if the number of samples is below a threshold, then we do so. The label to assign in case we reach this node during testing is the most repeated label among the incoming samples.

A random forest is a collection of decision trees. See this continuation video on random forests. The problem of decision trees is that they tend to overfit, that is, they perform well on the training samples but not that well on new, unseen samples (doesn't *generalize*). A solution to this problem is to train a collection of different, independent trees, get a prediction from each one of them and then take as result the most frequent output label. This is called an ensemble of experts. The way to produce such diverse

trees is to introduce randomization in the process of learning them. Two things are randomized: the samples entering the root node, and the features considered at each node. One parameter of the algorithm is the number samples (relative to the number of training samples) entering the root node of each decision tree. For each tree, this number of samples are independently drawn *with replacement* (thus, possibly with repetitions) from the training set. Another parameter is the number (or fraction) of features to consider at each node when minimizing the criterion. This number of features are again independently drawn *with replacement* from the list of all the features.

Other good references are the Random forest entry at Wikipedia, same for the Gini index and the documentation in Scikit-learn.

Note: there is no need to understand everything to the utmost detail, you can adapt the code in the non OO implementation or elsewhere, for instance for the Gini index. What we are interested the most is in the object-oriented design.

3 Random forests for regression

Given an input feature (or raw data) vector x, a classifier produces a label or class identifier $\hat{y} \in \{0 \dots c-1\}$, being c the number of classes. Regression is another task where the output is now a real number, so the continuous analogue of a classifier. To learn a regressor with a training set $(X_i, y_i), i = 1 \dots m, X_i \in \mathbb{R}^n, y_i \in \mathbb{R}$ means to learn a scalar function of n variables. It turns out that with a slight modification we can adapt random forests to make regression. Everything is the same except

- at leave nodes we replace the label by some summary of the samples reaching it during training, like the mean of the y_i's.
- the split criterion to minimize, for instance the sum of y's variances of the left and right subsets, which is called MSE (minimum squared error)

That simple. Figure 4 shows an example.

4 What to do

In this section we suggest a list of tasks to address the whole problem of design and implementation, ordered sequentially. Importantly, we also identify some directions in which our design may change in the future.

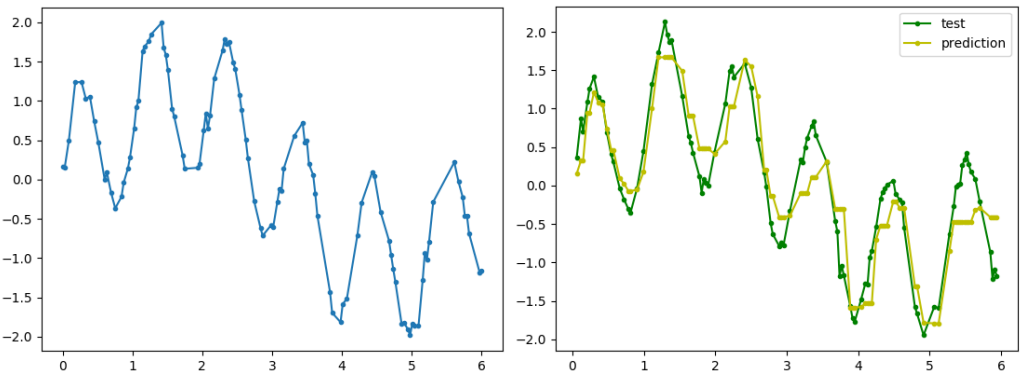

Figure 4: Left: training set of points. Right: test points and prediction at them. Source code at `points_regression.py`

4.1 Get to know random forests

Download the non object-oriented implementation `rfmachinelearningmastery.py` and the UCI Sonar dataset file. Run it to check it works. Study the code and identify the different steps of the algorithm in it. You will realize how messy the code is:

- it is based on the flat structure provided by many functions
- mixes all the responsibilities, no clear separation of the random forest, train, test process, representing the dataset
- bad names for variables and functions that make it difficult to understand

The implementation is specific for this Sonar dataset, because it heavily depends on the way it is represented once read from the CVS file: a dataset is a *list* of lists (rows), with the class label as the last element of each row.

Worse, there is not either a proper representation for the decision trees. Nodes of such trees are dictionaries with fields misleadingly named `index`, `values` and `groups`. Actually, `groups` is first the split of samples and then (!) is replaced by the left and right children.

In addition, it misses two responsibilities we are interested in, cross-validation and regression.

4.2 A first design

Make a first design, at the moment not considering all the responsibilities but only the core ones: representing the random forest and how to train and

test it, hence how to represent the pair (X, y) and its split into left and right subsets. Set aside cross-validation, computation of feature importance, and regression, but bear in mind that they will be introduced later. And since we won't want our design to suffer dramatic changes as a consequence of it, make a design that can later be gracefully extended in these directions.

Do consider, however, a random classifier against which to compare the future working random forest implementation. A random classifier takes (X, y) and, given a sample x, predicts its class $\hat{y}$ by just looking at the probability distribution (frequency of occurrence) of labels y, without regard of x. If there were 3 classes 0, 1 and 2 occurring in y 70%, 10% and 20% of times respectively, prediction could be implemented as

```
import numpy as np
ypred = np.random.choice(3, p=[0.7, 0.1, 0.2])
```

So, what to do ? Here's an incomplete list of tasks :

- Identify responsibilities and assign them to a first tentative set of classes. Draw them in an UML class diagram with as much detail as you can. For instance, decide what's the input, result and process for each method. Think of which messages will be sent (method calls) so as to achieve the main functionalities, like train and test the model.
- As a reference, have a look at the Scikit-learn library, which implements many classifiers and regressors. This library is well designed, with consistent names and class interfaces exhibiting only what the user needs to get the job done. Don't miss the documentation of random forests.
- Start small and iterate : repeatedly design a little, program a little.
- Do the design by hand at first and then write it in PlantUML using the web server `www.plantuml.com/plantuml`.
- You can reuse the not OO code, but with meaningful names for methods and variables
- Don't yet apply coding style conventions beyond names, and trace execution first with `print` and then with loggers
- Assess accuracy on the sonar dataset only, or some other classic like the https://en.wikipedia.org/wiki/Iris_flower_data_set also in UCI. Do not try MNIST yet.

The result of this first step should be a working code for random forests with a unique split criterion (Gini index). This step is the longest and most difficult: we have to jump from nothing to something well structured and working. Once achieved, the next steps will be easier.

4.3 Second design

We are going to implement just one split criterion for classification, the Gini index, like in the non OO implementation. However, we wish our design can be *easily* augmented with more criteria in the future, like Importance Gain or others the users will program. This means we want to minimize the necessary changes besides adding the new criterion implementation, ideally we would like to do nothing at all.

Also we want to introduce in the design the concept of k-fold cross-validation, and be able to assess the accuracy per fold and the average for all folds.

4.4 Regression

One of the nice properties of random forests is that they also are well suited to regression and we only need to perform a few small changes to convert our classifier into a regressor. So, the goal now is to extend our design with a random forest regressor. This implies to identify what a random forest classifier and a regressor have in common and move it somewhere to avoid, of course, redundancy (the "don't repeat yourself" principle).

The regression and classification methods we are implementing are both supervised, that is, they need samples to train. So the interfaces these two entities should offer are maybe quite similar. Moreover, random forests are for us just the beginning. In the future we intend to have a collection of supervised classifiers and regressors (for instance, SVM, k-nearest neighbour etc. see how many are them in the Scikit-learn library).

Note also that datasets for classification and regression may be different, they may have different responsibilities and/or one same responsibility be implemented in a different way in each case.

4.5 Feature importance

One of the advantages of decision trees over other models is that they are very easy to interpret, and as a consequence, random forests are also. In a decision tree we have, at each node, the specification of a decision rule: which feature to choose, and which threshold to apply to it in order to follow the

left or right branch. One way to interpret a random forest is to measure how important is each of the features. And a simple way to derive importance indices is to take all the nodes from all the trees and count how many times each feature is used. Actually, the count should be weighted by the number of samples on each subset (features higher in the tree are more important because they matter more, they take the first decisions), but we are not going to do so for the sake of simplicity.

To compute the importance of the features we obviously need to traverse the trees and do something at each node. There is also another thing we want to do in the sense of interpreting the random forest, and is to print all its trees. Again, it is necessary to traverse each one and print the information at each node. In the future we plan to have other processes to perform on the trees, like different ways to measure the importance, make a graphical representation of the nodes and the forest etc. We want to extend the design in this sense without having to change much or nothing, apart from adding these capabilities somewhere.

4.6 Coding style

Your code should follow the PEP-8 style guide for Python. Probably not 100%, but adhering to the conventions for names and comments is mandatory. Also, you have to run the style checker of your IDE (see section 5) and try to minimize the broken rules it points out.

4.7 Optimization

What's the bottleneck of the algorithm ? MNIST training will take forever unless you discover it. But then a simple change tailored to the case when features are raw images can speed up the training incredibly. Hint: good Python IDEs include a profiler, a tool that records how execution time is spent during a run. Launch it and locate the most critical parts.

Now you can use more trees and samples, and reach more than 90% accuracy only training for 10 minutes with all the 60K samples and assessing with all the 10K test samples! My result: with 20 trees, no feature extraction but raw image as representation, 92 % accuracy. However, do not spend much time trying to increase it. This is not a contest to get the highest accuracy, other types of classifiers which also learn the features surpass 99%. Also, there are a few irreducible errors because of impossible samples like those in figure 5.

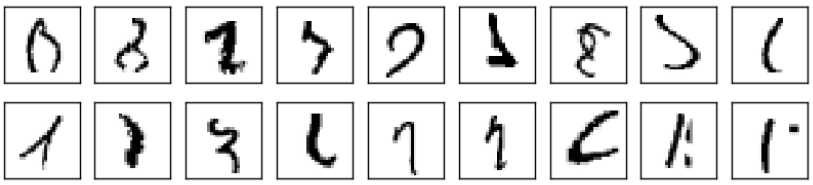

Figure 5: Can you read these digits?

5 Development

There are many IDEs (integrated development environment) for Python3. We recommend you Spyder3. There are versions for the three major computer platforms. An alternative is PyCharm (community edition). Development is easier with an IDE, and we will need two tools they normally include, a code style checker and a profiler.

The easiest way to set up the Spyder IDE *and* the required libraries of scikit-learn and scikit-image etc. is to install anaconda `www.anaconda.com` .

6 Grading

Your final version will be assessed according to the following grading scheme. Grades are quantized to 1 point, that is, "up to 2 points" means 0, 1 or 2 points.

0 points. At least one of the following : non existing paper design, even though the code does the job there are not enough classes or not well assigned responsibilities, none or too few design patterns identified and used, some important OO principles not respected.

up to 3 points. Basic design correct, follows OO principles, classification just works, most design patterns for classification identified and used. Once you get to this level, you can add the points from the following items. We recommend to follow the same order.

up to 2 points. Design and implementation of feature importance.

up to 2 points. Extension of random forests for classification to regression.

up to 2 points. Coding style adherence (with emphasis in comments and names) and good usage of assertions and loggers.

1 point. Optimization for MNIST and visualization of feature importance. Or some other improvement proposed by the students and agreed upon with the instructor.

A Python commands you should know

This is a list of useful functions and class methods for you to study in some reference manual of the IDE online help:

- `float`, `int` (truncate) to cast a variable to a type , `round`
- methods of the `list` class : `append`, `remove`, + operator, `extend`, `len`
- expressions `[x**2 for x in range(10)]`, `3 not in [1,2,4,5]` is `True`
- methods of a `numpy` array : `shape`, `ravel`, `transpose`, `var`, `astype`, `sum` (of a numeric but also of a boolean array), `sqrt`, `mean`. Functions `arange`, `linspace`, `sort`, `argsort`, `unique`, some methods like `sort` are also functions.
- in `numpy.random` : `seed`, `rand`, `randint`, `choice`, `permutation`
- most frequent element in a list or array `x`, `max(set(x), key=x.count)`
- dictionaries : methods `update`, `keys`, `items`, new dictionary {}
- `pass`, `break`, `raise NotImplementedError`
- file input/output : in `numpy` package `load`, `save`, `savez` for arrays, in the `pickle` package `load` and `dump` for objects
- `matplotlib.pyplot` package : see the scripts to generate the figures in this document, see also the gallery in matplotlib.org