Machine Learning: algorithms, Code Lecture 5

Bagging, forests, CV and Trees with weights

Nicky van Foreest

May 26, 2021

Contents

1 Overview

- Last lecture
  - Recursion and Trees
- This lecture
  - Bagging
  - Random forests
  - Cross validation
  - Trees with weights
- Next lecture:
  - Boosting
  - AdaBoost

2 Bagging

I found this site very useful to understand how bagging works, and I copied a bunch of ideas from that site. However, I also have some strong objections against how the code is organized there. Let me first show you my code, and then discuss my objections.

You can find `bag.py` in the `code` directory on `github`.

2.1 Implementation

I can use the `Tree` class I derived earlier.

```
import numpy as np

from tree_simple import Tree

rng = np.random.default_rng(3)
```

Note that I don't use `from tree import *`. As a matter of principle, you should never import everything from other modules. See the exercises below to understand why.

There seems to be a small change with respect to how to call the random number generator in `numpy`. I found this change in numpy's documentation when I was searching how to randomly select n elements of a set with replacement.

```
class Bag:
    def __init__(self, min_size, max_depth, n_trees):
        self.n_trees = n_trees
        self.min_size = min_size
        self.max_depth = max_depth
        self.trees = []
```

A bag consists of `n_trees` trees, each of which uses a bootstrap of the data to build the tree. The next method of `Bag` implements this.

```
def fit(self, X, Y):
    self.X, self.Y = X, Y
    n, p = self.X.shape
    for _ in range(self.n_trees):
        bootstrap = rng.integers(low=0, high=n, size=n)
        X, Y = self.X[bootstrap], self.Y[bootstrap]
        tree = Tree(min_size=self.min_size, max_depth=self.max_depth)
        tree.fit(X, Y)
        self.trees.append(tree)
```

The last step is to predict the label of a new measurement `x`, or set of measurements `X`. We use again a majority vote, just as in the case of a single tree. You should check in the implementation of the `Tree` how to find the most occurring label. The idea below follows the same logic. Finally, `predict` is just a convenience function that calls the `majority_vote` method; if we want to use bags for regression, then we can still use `predict`, but then use a another internal method to predict the regression.

Here is a subtle point, I use in the `predict` method that $Y_i \in \{-1, 1\}$. Think about why this works. (If you would actually use such tricks, then ensure that you test the inputs. If the outcomes Y of the training set don't satisfy this condition, you will get an answer, but it can easily be wrong. Why? Hint, if $Y_i \in \{0, 1\}$, can you ever get a negative majority vote?)

```python
def majority_vote(self, X):
    predictions = np.array([t.predict(X) for t in self.trees])
    return np.sign(predictions.sum(axis=0))


def predict(self, X):
    return self.majority_vote(X)
```

I write a capital `X` to indicate that we can pass a bunch of observations to the `predict` method, not just one observation.

2.2 A test

```python
def test():
    X = np.array(
        [
            [2.771244718, 1.784783929],
            [1.728571309, 1.169761413],
            [3.678319846, 2.81281357],
            [3.961043357, 2.61995032],
            [2.999208922, 2.209014212],
            [7.497545867, 3.162953546],
            [9.00220326, 3.339047188],
            [7.444542326, 0.476683375],
            [10.12493903, 3.234550982],
            [6.642287351, 3.319983761],
        ]
    )
    Y = np.array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1])

    bag = Bag(min_size=1, max_depth=2, n_trees=5)
    bag.fit(X, Y)
    print(bag.predict(X))

    tests = [[3, 10], [4, -5], [6, 1], [7, 2], [8, 5]]
    for x in tests:
        x = np.array(x).reshape(1, 2)
        print(f"Predicted class of {x}: {bag.predict(x)}")
```

2.3 Discussion of other code

The code at this site, from which I learned quite a bit, has some shortcomings.

- They use one function to *make* bags and *test* bags at the same time. This is weird: such conceptual ideas should be split.
- Some functions do too much. For instance, the `gini_index` function computes the gini index for each group and then computes the overall score too. It's better to split this. Compute a score for a group in one function, and compute the total score in another function. Like this, if you want to use another score function for a group, you just have make a change in one place, rather than two. In general, the more dependencies among different pieces of code, the buggier it becomes. It's *way* better to build one function for each separate task.

- There are strange names: `sample_size` evokes (for me at least) an `int`, not a `float`. But the author uses this variable as a fraction. Why not call it `sample_frac` then?

3 Random forests

I found this site useful for random forests. However, I made my own implementation.

You can find `random_forest.py` in the `code` directory on `github`.

3.1 Implementation

The imports for a random forest are mostly the same as for the bag. As you'll see in a minute, I can make a random forest by sub-classing a bag. (Classes are so elegant, once you get the hang of it.)

```
import numpy as np
from scipy.stats import bernoulli

from tree_simple import Tree
from bag import Bag

rng = np.random.default_rng(3)
```

To get a random forest, I can subclass a `Bag`, and overwrite the method to make trees. I have to fix the number of features that each tree should use. Hence, I have to overwrite the `__init__` method, since there is the extra argument `n_features`; the rest I can pass on to `Bag.__init__`.

```
class RandomForest(Bag):
    def __init__(self, min_size, max_depth, n_trees, n_features):
        super().__init__(min_size, max_depth, n_trees)
        self.n_features = n_features
```

Making the trees for a random forest is nearly the same as making trees for a bag. We have to bootstrap a number of samples `X` of `self.X`. Then, from the p features (columns of `X`), we have to choose `n_features` at random without replacements. Then we build a tree for the 'thinned' observations. And that is all there is to it. The rest of the methods we inherit right away from `Bag`; no sweat here.

```
def fit(self, X, Y):
    self.X, self.Y = X, Y
    n, p = self.X.shape
    for _ in range(self.n_trees):
        bootstrap = rng.integers(low=0, high=n, size=n)
        X, Y = self.X[bootstrap], self.Y[bootstrap]
        features = rng.choice(range(p), size=self.n_features, replace=False)
        tree = Tree(min_size=self.min_size, max_depth=self.max_depth)
        tree.fit(X[:, features], Y)
        self.trees.append(tree)
```

3.2 A test

```python
def test():
    X = np.array(
        [
            [2.771244718, 1.784783929],
            [1.728571309, 1.169761413],
            [3.678319846, 2.81281357],
            [3.961043357, 2.61995032],
            [2.999208922, 2.209014212],
            [7.497545867, 3.162953546],
            [9.00220326, 3.339047188],
            [7.444542326, 0.476683375],
            [10.12493903, 3.234550982],
            [6.642287351, 3.319983761],
        ]
    )
    Y = np.array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1])

    RandomForest.max_depth = 2
    RandomForest.min_size = 1
    rf = RandomForest(max_depth=2, min_size=1, n_trees=51, n_features=1)
    rf.fit(X, Y)

    tests = [[1, 10], [4, -5], [6, 1], [7, 2], [8, 5]]
    for x in tests:
        x = np.array(x).reshape(1, 2)
        print(f"Predicted class of {x}: {rf.predict(x)}")
```

4 Cross validation (CV)

Part of a model's complexity is determined by (a) hyper parameter(s), such as the highest degree of the polynomials used to fit the data, or the number of trees in a bag, or the number of features in a random forest.

We like to 'know' the best/most robust hyper parameter(s) while balancing bias (errors due do selecting a too simple model) and variance (errors due to a too sensitive/complex model, over fitting). With CV we can find the values for the hyper parameter(s) that perform 'best'. Once we have the best values for hyper parameter(s), we use the *entire* data set to fit (e.g., find β).

BTW: At first I was a bit confused about this procedure, i.e., we did CV, but then what? I missed the step that we use CV to find the best hyper parameters (or to see whether our fitting model makes any sense at all). Only after some thinking, I realized that once we have the hyper parameters, we can use the entire data for fitting.

```
import numpy as np
import pandas as pd
import ols

df = pd.read_csv("data_by_year_o.csv")

X = df[["tempo"]].values
Y = df["popularity"].values

xx = X  # keep for plotting
X = np.c_[np.ones(len(Y)), X]  # put 1 in front for beta_0

n, p = X.shape

K = 100

loss = []
B = int(n / K)  # fold (batch) size
for k in range(K):
    s = np.full(len(Y), True)  # train set
    s[k * B : (k + 1) * B] = False  # take out the test set

    X_train, Y_train = X[s], Y[s]
    X_test, Y_test = X[~s], Y[~s]

    ol = ols.OLS(X_train, Y_train)
    ol.solve()
    loss.append(ol.loss(X_test, Y_test, ol.beta))

print(sum(loss) / n)
```

In this code I just use OLS to illustrate how to do CV with python (and make the code of DSML a bit more pythonic.). For OLS there are no hyper parameters, so there is not much to vary here. If you were to apply this to bagging for instance, then you should select a range for the number of trees you want to include in the bag. Then do the CV for each of the number of trees. Recall, the number of trees to include in the bag is the hyper parameter. We use CV to figure out which value of the hyper parameter works best.

Compare DSML, Figure 2.11 to see how to embed all this.

5 Tree with weights

5.1 Levels of understanding

1. I first built the tree in generic code with the formulas of DSML, but without the weights.
2. Then I wanted to use AdaBoost with my own tree, rather than the one of `sklearn`.
3. AdaBoost involves trees with weights.
4. Making a tree with weights required much more understanding than I initially thought
5. Finally, I realized that we are dealing with a binary tree, so $Y_i \in \{-1, 1\}$, and there can be just left and right sub trees. With this, I could make the code a bit simpler.

Here is my tree with weights. The code is on github in the file `tree_with_weights.py`.

5.2 Score functions

As it turns out, the *description* of the AdaBoost algorithm in DSML does not completely match the python *implementation* of AdaBoost in DSML. The subtlety is that the description in DSML uses a zero-one training loss function to compute the impurity, i.e., similar to the misclassification impurity, while the `DecisionTreeClassifier` of `sklearn` uses the Gini impurity. To see how to align this, I had to rethink the entire procedure. It's an interesting challenge to try to do this yourself. Spoiler alert: Below is my explanation, so in case you want to derive things yourself, stop reading.

To see how to generalize the computation of score functions with weights, it seems best to follow the steps of DSML, and then see how to include weights.

Given n data points, suppose we want to find a separator s^* such that most points with label -1 are assigned to the left node L, and the points with label 1 to the right node R. For this we first need to compute the misclassification score of an arbitrary separator s. Let s split a set A into a left set L, which outcomes -1, and a right set R, with outcomes 1, then the score of s is given by

$$S(s) = \frac{1}{n}\sum_{i\in L} 1_{Y_i\neq -1} + \frac{1}{n}\sum_{i\in R} 1_{Y_i\neq 1} \tag{1}$$

$$= \frac{|L|}{n}\frac{1}{|L|}\sum_{i\in L} 1_{Y_i\neq -1} + \frac{|R|}{n}\frac{1}{|R|}\sum_{i\in R} 1_{Y_i\neq 1}. \tag{2}$$

Here we can interpret

$$m(L) = \frac{1}{|L|}\sum_{i\in L} 1_{Y_i\neq -1}, \qquad\qquad m(R) = \frac{1}{|R|}\sum_{i\in R} 1_{Y_i\neq 1},$$

as the missclassification scores of the left and right set. Let us rewrite this formula for S into a more general form, so that we can see how to include weights.

Define, for some set A of observations, the proportion of observations with label $z \in \{-1, 1\}$ as

$$p_z(A) = \frac{1}{|A|}\sum_{i\in A} 1_{Y_i=z}. \tag{3}$$

Suppose that label 1 occurs most in set A, then the misclassification $m(A)$ of this set satisfies

$$m(A) = \frac{1}{|A|}\sum_{i\in A} 1_{Y_i\neq 1} = 1 - \frac{1}{|A|}\sum_{i\in A} 1_{Y_i=1} = 1 - p_1(A) = 1 - \max\{p_{-1}(A), p_1(A)\}. \tag{4}$$

Observe that since the RHS does not depend on the label z, the misclassification $m(A)$ actually does not depend on our choice of label 1. It's a set property, hence only depends on A.

With this notion of misclassification impurity, we can write the score of the selector s as

$$S(s) = \frac{|L|}{n}\frac{1}{|L|}\sum_{i\in L} 1_{Y_i\neq -1} + \frac{|R|}{n}\frac{1}{|R|}\sum_{i\in R} 1_{Y_i\neq 1} \tag{5}$$

$$= \frac{|L|}{n}m(L) + \frac{|R|}{n}m(R). \tag{6}$$

Now realize that we can replace $m(L)$ and $m(R)$ by other impurity measures, for instance the Gini measure

$$G(A) = \frac{1}{2}\left(1 - p_{-1}(A)^2 - p_1(A)^2\right). \tag{7}$$

In that case, the score of s can be written as

$$S(s) = \frac{|L|}{n}G(L) + \frac{|R|}{n}G(R). \tag{8}$$

The next step is to include weights in the score function. Suppose we give weight w_i to assigning point i to the correct class. Then we generalize the proportion $p_z(A)$ to the *weighted proportion*

$$p_z(A, w) = \frac{\sum_{i\in A} w_i \, 1_{Y_i=z}}{\sum_{i\in A} w_i}. \tag{9}$$

By analogy, the weighted misclassification impurity for a set A becomes

$$m(A, w) = 1 - \max\{p_{-1}(A, w), p_1(A, w)\}, \tag{10}$$

and the Gini score becomes

$$G(A, w) = \frac{1}{2}\left(1 - p_{-1}(A, w)^2 - p_1(A, w)^2\right). \tag{11}$$

The weights $|L|/n$ and $|R|/n$ in $S(s)$ now should be replaced by the new weights $\sum_{i\in L} w_i / \sum_i w_i$ and $\sum_{i\in R} w_i / \sum_i w_i$.

With this, the *weighted score* of the separator s that splits the set of data points into subsets L and R becomes

$$S(s) = \frac{\sum_{i\in L} w_i}{\sum_i w_i} m(L, w) + \frac{\sum_{i\in R} w_i}{\sum_i w_i} m(R, w). \tag{12}$$

We can simplify this trivially if $\sum_{i=1}^n w_i = 1$. Finally, we can replace the impurity $m(\cdot, w)$ by $G(\cdot, w)$ if we like.

Recall, our goal was to find the separator s^* the minimizes $S(s)$, but we already built this in our earlier tree implementation, so we don't have to deal with this here.

5.3 Implementation

I am going to subclass from my earlier `Tree` class, the tree without weights. I do this on purpose so that you can understand the similarities and differences. In a 'more professional' implementation, I would just use the tree with weights, and then give the weight default values to cover the tree with uniform weights (i.e., the standard tree).

```
1 import numpy as np
2 from sklearn.tree import DecisionTreeClassifier
3 from sklearn.datasets import make_blobs
4
5 from tree_simple import Tree as SimpleTree
```

As explained above we need a probability with weights. Note that when there are no outcomes Y_i equal to $z \in \{-1, 1\}$, we should return 0.

```
def p(Y, w, z):
    res = w[Y == z].sum()
    if res == 0:
        return 0
    return res / w.sum()
```

For the score I use that there just two possible observations: -1 and 1. Thus, the implementation is a bit less generic than the one in `tree_simple.py`. However, since we are splitting in a left and right tree, there is no need to have more options. In fact, in good code, we should check that $Y_i \in \{-1, 1\}$ for all i, and we trigger an error if this is not the case.

```
def missclassify(Y, w):
    return 1 - max(p(Y, w, -1), p(Y, w, 1))


def gini(Y, w):
    return 1 - p(Y, w, -1) ** 2 - p(Y, w, 1) ** 2
```

Here is the tree with weights, as a subclass of the simple tree.

```
class Tree(SimpleTree):
    def score(self, s):
        # return missclassify(self.Y[s], self.w[s])
        return gini(self.Y[s], self.w[s])

    def score_of_split(self, i, j):
        s = self.X[:, j] <= self.X[i, j]
        l_score = self.score(s) * self.w[s].sum() / self.w.sum()
        r_score = self.score(~s) * self.w[~s].sum() / self.w.sum()
        return l_score + r_score

    def fit(self, X, Y, weights=None):
        self.test_input(X, Y)
        self.X, self.Y = X, Y
        if weights is None:
            weights = np.ones(len(Y))
        self.w = weights / weights.sum()
        self.split()

    def majority_vote(self):
        if p(self.Y, self.w, -1) >= p(self.Y, self.w, 1):
            return -1
        return 1
```

I'll explain the `test_input_` method below.

The `majority_vote` method requires some attention. When using weights, the prediction with the largest probability should win.

5.4 Tests on input data

Here is some simple code to demonstrate whether the input is OK or not. In real code this can be quite extensive. You should know that testing the input is often a good idea: suppose for

some crazy reason that the input is wrong, but your program still gives some result... That is a clear recipe for disaster later.

```
def test_input(self, X, Y):
    n, p = X.shape
    if len(Y) != n:
        print("Tree: len Y is not the same as the number of rows of X")
        exit(1)
    if not set(np.unique(Y)).issubset([-1, 1]):
        print("Tree: The observations Y are not all equal to -1 or 1.")
        exit(1)
    return True
```

Don't take this example too seriously. There are libraries to help you organize how to test input. Once, again, use good ideas of others on how to organize such standard tasks.

5.5 Tests

Here is how to test our check on the inputs of the tree.

```
def test_inputs():
    tree = Tree()
    X = np.arange(5).reshape(5, 1)
    Y = np.array([1, 0, 0, 1])
    # the test on the dimensions of X and Y should fail
    assert tree.test_input(X, Y) is False

    # The test on the values of Y should fail
    Y = np.array([1, 0, 0, 1, 1])
    assert tree.test_input(X, Y) is False

    # Now the dimensions of X and Y are OK, just as the values of Y.
    Y = 2 * Y - 1
    assert tree.test_input(X, Y) is True
```

Here are some tests for the tree class itself.

```python
def test():
    X = np.array(
        [
            [2.771244718, 1.784783929],
            [1.728571309, 1.169761413],
            [3.678319846, 2.81281357],
            [3.961043357, 2.61995032],
            [2.999208922, 2.209014212],
            [7.497545867, 3.162953546],
            [9.00220326, 3.339047188],
            [7.444542326, 0.476683375],
            [10.12493903, 3.234550982],
            [6.642287351, 3.319983761],
        ]
    )
    Y = np.array([0, 0, 0, 0, 0, 1, 1, 1, 1, 1])
    Y = 2 * Y - 1

    tree = Tree()
    tree.fit(X, Y)
    print(tree.predict(X))
```

```python
def test_2():
    X = np.arange(5).reshape(5, 1)
    Y = np.array([1, 0, 0, 1, 1])
    Y = 2 * Y - 1
    n, p = X.shape
    w = np.ones(n)
    w[0] = 100
    w /= w.sum()

    tree = Tree()
    tree.fit(X, Y, sample_weight=w)
    my_y = tree.predict(X)

    clf = DecisionTreeClassifier(max_depth=1)
    clf.fit(X, Y, sample_weight=w)
    their_y = clf.predict(X)
    print((my_y == their_y).all())
```

```
def test_3():
    np.random.seed(4)
    X, Y = make_blobs(n_samples=13, n_features=3, centers=2, cluster_std=20)
    Y = 2 * Y - 1
    n, p = X.shape
    w = np.random.uniform(size=n)
    w /= w.sum()

    tree = Tree()
    tree.fit(X, Y, sample_weight=w)
    my_y = tree.predict(X)

    clf = DecisionTreeClassifier(max_depth=1)
    clf.fit(X, Y, sample_weight=w)
    their_y = clf.predict(X)
    print((my_y == their_y).all())
```

6 Exercises

6.1 Exercise 1.

Read here to understand why using `import *` is a very bad idea. The fact that DSML uses `import *` does not make it any better. Let me be honest: It annoys me that the standard in DSML about math and code are quite a bit different. The math looks like it should, i.e., impeccable, but the code often does not (here and there ugly, bad formatting, heavy-handed design.). This is just as awkward the other way round, correct code and sily math.

6.2 Exercise 2.

Perhaps you like the explanation of trees of this site.

1. I like that the author uses classes. Of course I like my own `Tree` class better. (There is no reason to have a `Node` class and a `DecisionTree` class.) However, beauty is in eye of the beholder, so study which of the two you like best.

2. I also like that the author builds a `fit` method. So I did the same.

3. I don't like the documentation in between the code. It makes it hard to focus on (and locate) the real code. That is one of the reasons I like the concept of *literate programming* much more. In literate programming, code and documentation are very clearly separated, which adds much to the understanding of the code, and the text.

6.3 Exercise 3.

What's your opinion of the code on this site? There is a point of confusion: does the author call a *random forest* what we call *bagging*? I think so, but I haven't studied the code in detail, so I might be wrong.

6.4 Exercise 4.

If you are interested in preventing dumb errors when inputting data, you can consult bear checker.

6.5 Exercise 5.

This is an easy one: check this video on YouTube on how AI learns how to park a car. It will take a long time before computers will beat us at even the simplest things.