Machine Learning: algorithms, Code Lecture 4

Recursion and Decision Trees

Nicky van Foreest

May 21, 2021

Contents

1 Overview

- Last lecture
  - LP solvers, graphs, applied to project scheduling, finding the critical path
  - ridge regression, with direct method ($Ax = b$) and `CVSPY`
  - lasso regression: `CVXPY`
- This lecture
  - Recursion, examples
  - Decision Trees (Trees are recursive)
- Next lecture:
  - Bagging
  - Random Forests Cross Validation implementation
  - Trees with weights.

2 Recursion

Recursion is an important idea in the design of computer algorithms. Here I discuss a few, to help you get in the mood. Then I'll apply the idea of recursion to make decision trees.

The idea of the lecture is to help you think recursively. Once you understand how recursion works, the implementation of the `Tree` class will be much more natural.

The main idea of recursion is do just a tiny bit of work, and pass on the problem to another instance, but with a *smaller* data set. (It's a bit like a manager: do as little as possible, and pass on the problem to somebody else.)

2.1 Fibonacci sequence

Rabbit pairs increase per generation, supposedly, according to Fibonacci's rule:

$$F(n) = F(n-1) + F(n-2), \quad F(0) = F(1) = 1. \tag{1}$$

When using recursion, *always* think about the stopping condition. If you mis a corner case, your code will keep running. Often it will claim more and more memory, and your computer will crash. Stopping conditions should be *complete*: the code below will crash if you call `F(0.5)` (why?), or `F(-1)`. (In good code you should check the input.)

```
def F(n):
    global called
    called += 1
    if n == 0 or n == 1:
        return 1
    return F(n - 1) + F(n - 2)


for n in range(20):
    called = 0
    F(n)
    print(n, called)
```

```
0 1
1 1
2 3
3 5
4 9
5 15
6 25
7 41
8 67
9 109
10 177
11 287
12 465
13 753
14 1219
15 1973
16 3193
17 5167
18 8361
19 13529
```

The number of times we call $F(n)$ in the course of the recursion increases very rapidly. Why is that?

2.2 A bit harder: use *caching* to store information

What if we would *store* intermediate results rather then recompute them? The concept of storing intermediate results is known as *memoization*, or *cashing*, or *hashing*. You should memorize that when doing expensive queries on (very large) data bases you should always try to store the intermediate results.

```
from functools import lru_cache


@lru_cache
def F(n):
    global called
    called += 1
    if n == 0 or n == 1:
        return 1
    return F(n - 1) + F(n - 2)


called = 0
F(19)
print(called)
```

```
20
```

Rather than doing 13529 computations, we now just need 20 calls! If you don't believe how useful memoization is, try your hand at `F(300)` with and without memoization. (Without memoization, you'll be dead before knowing the answer.)

Note that in the code, we first have to check the boundary condtions. Only when we *fell through* all conditions, we can apply the recursion.

2.3 Memoization to solve difference equation (finance applications), boundary conditions.

We have Peter and Paul betting on the outcome of a coin for t rounds. Peter wins a dollar if it lands heads, which happens with probability p; if it lands tails, Paul wins a dollar. Peter starts with i dollars, Paul with $n - i$ dollars. Peter wins when during the course of the game he ends up with all n dollars. When he has not won all n dollars before $t = 0$, or Paul won all n dollars, Peter looses.

2.3.1 What is the probability that Peter wins?

Let $u_{t,i}$ denote the probability that Peter wins if he has i dollars. Clearly, $u_{t,i}$ satisfies the recursion:

$$u_{t,i} = pu_{t-1,i+1} + qu_{t-1,i-1}, \tag{2}$$
$$u_{t,n} = 1, \quad t \geq 0, \tag{3}$$
$$u_{t,0} = 0, \quad t \geq 0, \tag{4}$$
$$u_{0,i} = 0, \quad 0 \leq i < n. \tag{5}$$

```python
from functools import lru_cache

p = 26/50
q = 1 - p
n = 10


@lru_cache()
def u(t, i):
    if i == n:
        return 1
    if i == 0:
        return 0
    if t == 0:
        return 0
    return p * u(t - 1, i + 1) + q * u(t - 1, i - 1)


print(u(100, 5))
```

```
0.5939863359639174
```

Using recursion and memoization keeps the code short and very clean. (At least, I find real neat.)

A small detour: for the mathematically interested. In the following problem we remove the dependence on t.

$$u(i) = pu(i+1) + qu_{i-1}, \tag{6}$$
$$u(n) = 1, \tag{7}$$
$$u(0) = 0. \tag{8}$$

Suddently we end up with a *two-point boundary value problem*. For this specific example it's simple to find an analytic solution, but in general two-point boundary value probles are numerically much harder to solve than differential equations with only initial conditions.

2.3.2 What is the expected duration of the game?

Clearly, the game ends when either of the boundaries 0 or n is hit, or when Peter and Paul run out of rounds. The recursions are simple. Let $v(t, i)$ be the expected duration of the game when there are at most t rounds left and Peter starts with i dollars. Then,

$$v_{t,i} = 1 + pv_{t-1,i+1} + qv_{t-1,i-1}, \tag{9}$$
$$v_{t,n} = v_{t,0} = 0, \quad t \geq 0, \tag{10}$$
$$v_{0,i} = 0, \quad \text{for all } i. \tag{11}$$

```python
from functools import lru_cache

p = 26/50
q = 1 - p
n = 10


@lru_cache()
def v(t, i):
    if i == n or i == 0 or t==0:
        return 0
    return 1 + p * v(t - 1, i + 1) + q * v(t - 1, i - 1)


print(v(100, 5))
```

```
24.528910715863503
```

2.4 Sorting without recursion is the the dumbest way to sort

Arrays can be solved very fast with recursion. But to see how sorting works, let us first see how *not* to solve an array.

```python
def bublle_sort_never_use_this_as_it_is_the_dummest_algorithm_to_sort(L):
    for i in range(len(L)):
        for j in range(i + 1, len(L)):
            if L[i] > L[j]:
                L[i], L[j] = L[j], L[i]  # reverse the elements


L = [3, 4, 1, 8, 10]
bublle_sort_never_use_this_as_it_is_the_dummest_algorithm_to_sort(L)
print(L)
```

```
[1, 3, 4, 8, 10]
```

The complexity of this algorithm is $n(n-1)/2 \sim O(n^2$.

2.5 Quicksort

Sorting becomes much faster when using recursion. In fact, we go from an order $n(n-1)/2 \sim O(n^2)$ algorithm (bubble sort) to an $O(n \log n)$ algorithm (quicksort).

```python
def qsort(xs):
    if len(xs) <= 1:
        return xs
    left, right, pivot = [], [], xs[0]
    for x in xs[1:]:
        if x < pivot:
            left.append(x)
        if x >= pivot:
            right.append(x)
    return qsort(left) + [pivot] + qsort(right)
```

- We assume that the order in the initial array is random. Hence, `xs[0]` has an arbitrary value.
- Once, again, the algorithm does a bit work, and passes on the rest of the work to instances, but with *less* data.
- Note the boundary/stopping condition here: it's the check on the length of the list `len(xs) <=1`.
- *All* sorting algoriths that you use on the contact list on your phone, spotify playlists, databases, use recursive sorting algoritms.
- Don't *use* the above code, as we do not cope with many corner cases. The idea is that you understand how recursion works in combination with sorting.

3 Decision Trees

To build a decision tree, I started reading the code of DSML, but I did not really like the implementation. I found it hard to understand, and (worse perhaps) I found the code somewhat ugly, and particulary 'unpyhonic'.

I tried to find another resource on the internet. Searching on the web on `build from scratch decision tree` led me to this useful site. However, this also was not the way I like to see trees implemented.

1. They do not use classes, which makes the code a bit unorganized. In fact, this is a missed opportunity because for trees have a recursive structure, which can be very nicely built with classes.
2. I want to use numpy to speed up things.
3. There also quite some points that confuse me, like the use of `row` and `index`. I find it hard to memorize that `index` refers to a column of `X`, that a `row` is a row of `X` (why not call it `X[i,:]`?), and that `row[-1]` is the outcome/label `Y`.

So, I decided to build my own tree, but steal what seemed the best of the code of DSML and the above site.

You can find my code on `github` in `tree_simple.py`. With the code below I just want to illustrate how such things are built. For real work, use the classes of `sklearn` such as `DecisionTreeClassifier`. These are considerably more sophisticated but also quite a bit harder to understand (and if you don't believe me, just check the source on github).

3.1 Think a bit!

Before building a decision tree, we need to think a bit on the design. For this, we can use (at least) three questions:

1. What is the purpose of a decision tree?
2. How to use recursion, i.e., what should 'the manager' do, and what data should s/he pass on to a lower level?
3. When to stop, i.e., identify stopping conditions.

3.2 Score functions

I know that I'll need a score function to split the data into subsets. So that is what I am going to build first.

Before building the score, we need the fraction $p(z)$ of elements in a set σ with value z:

$$p_z = \frac{1}{|\sigma|} \sum_{y \in \sigma} 1_{y=z}. \tag{12}$$

I'll use Gini's impurity function to score the misclassificaton of the tree. From the definitions of DSML:

$$G = \frac{1}{2} \left(1 - \sum_z p_z^2 \right). \tag{13}$$

With $p(z)$ it's easy to also build the misclassification and entropy scores.

```
def p(Y, z):
    return (Y == z).sum() / len(Y)


def gini(Y):
    res = np.array([p(Y, z) for z in np.unique(Y)])
    return (1 - res @ res) / 2

def missclassify( Y):
    res = np.array([p(Y, z) for z in np.unique(Y)])
    return 1 - res.max(initial=0)


def entropy(Y):
    res = np.array([p(Y, z) for z in np.unique(Y)])
    return -res @ np.log(res)
```

Below you will see that I'll associate a score function to the `Tree` class.

3.3 The tree class

A tree is a bunch of branches; branches can have sub branches, and so on. But, since a branch behaves like a tree, it suffices to just have one `Tree` class. Left and right branches are then also trees.

You should realize now that we are only considering *binary* classification. So, we split in a *left* and a *right* subtree; there will not be a *middle* tree, or anything else.

We don't want to split a tree when it has less than `min_size` datapoints. We also don't want to split when a tree's depths exceeds `max_depth`.

```
class Tree:
    def __init__(self, depth=0, min_size=1, max_depth=1):
        self.X, self.Y = None, None
        self.depth = depth
        self.left, self.right = None, None
        self.split_col = self.split_row = None
        self.split_value = None
        self.min_size = min_size
        self.max_depth = max_depth
```

The `X` is the feature vector, `Y` the response, `left` and `right` refer to a left and right tree, if present. Finally, I use `split_row` and `split_col` to save the row and column of `X` that are optimal to split, the `split_value=X[split_row, split_col]` is the value to use in the optimal split.

For the interested: I wanted implement the max depth and min size as class attributes `Tree.max_depth` and `Tree.min_size`. This makes sense since all subtrees in one tree must satisfy the same constraints. However, I did not see a clear way to set these variables when instantiating a tree. In the current implementation I can just pass the values to the `__init__` method.

3.4 Convenience methods

The size of a tree is determined by the number of datapoints it contains. For this I can just check on `len(self.Y)`, but I prefer to read `self.size()`; this saves me thinking what about the meaning of `len(self.Y)`. (Later, perhaps even tomorrow, I might be thinking why I am interested in the length of `Y`, while `size` tells me directly what I wanted to do.)

It's more robust to call the `score` method everywhere. This hides the details of the exact score function I want to use. If later I would prefer another score function, I can leave the rest code untouched; I only have the change `gini` to, e.g., `entropy`. The argument `s` is a vector with true/false values. When `s[i] = True`, we include `Y[i]` in the score, otherwise not.

```
def size(self):
    return len(self.Y)

def score(self, s):
    return gini(self.Y[s])
```

Later, when building trees with weights, we'll see that it's convenient to pass a selector `s` to the score method, rather than `self.Y[s]`.

3.5 API

The `fit` and `predict` methods are meant to offer the same API as the classes of `sklearn`. We'll build the `split` and `_predict` methods below.

```python
def fit(self, X, Y):
    self.X, self.Y = X, Y
    self.split()

def predict(self, X):
    return np.array([self._predict(x) for x in X])
```

3.6 Score of a split

We want to split the data `X` along column `j`. Then for a given `value`, we should put all rows `X[row,col] <= value` in the left tree, and the rest in the right tree. The selector `s=X[:,j] <= value` gives all rows that have to go the left tree. The value we need to split is the ith element of X in column j. By inverting the selector, i.e., `~s`, I get data that thas to go the right tree.

The value to use in the split is given by `X[i,j]`.

The score of the split is the score of the left and right nodes, each weigthed according to their size.

```python
def score_of_split(self, i, j):
    s = self.X[:, j] <= self.X[i, j] # split on pivot
    l_score = self.score(s) * len(self.Y[s]) / len(self.Y)
    r_score = self.score(~s) * len(self.Y[~s]) / len(self.Y)
    return l_score + r_score
```

3.7 Optimal split

I have to find the best split of the data. For this, I run over each column in `self.X`. Then for a given column, I find the best value to split the rows. This best value must be an element of `self.X`, so I keep a reference to the best column and best row.

Note that we can take `self.score(Y)` as upper bound on the split. If none of the splits results in a lower score, we shouldn't split.

```python
def find_optimal_split(self):
    n, p = self.X.shape
    best_score = self.score(np.full(len(self.Y), True))
    best_row = None
    best_col = None
    for j in range(p):
        for i in range(n):
            score = self.score_of_split(i, j)
            if score < best_score:
                best_score, best_row, best_col = score, i, j
    self.split_row = best_row
    self.split_col = best_col
    self.split_value = self.X[best_row, best_col]
```

3.8 Do the split

Once I know the optimal split, I just have to implement the split. However, we should not split when:

1. the size of the tree is less or equal to `min_size`;
2. `max_depth` is reached;
3. the best split does not reduce the score;
4. after an optimal split, either the left or right tree is empty.

Otherwise, we can make a good split, then we do it, and fit the left and right tree to the subsets of the data each of them receaves. Since these trees are instances of `Tree`, we can leave the process of fitting entirely to each of the sub trees themselves. This is the beauty, but perhaps also a bit of the magic, of recursion. (Once, again, the idea is to not do much yourself; pass on the work to somebody else, but ensure that there is less work to do for the other person.)

```
def split(self):
    if self.size() <= self.min_size or self.depth >= self.max_depth:
        return
    self.find_optimal_split()
    if self.split_row == None:
        return
    s = self.X[:, self.split_col] <= self.split_value
    if s.all() or (~s).all():
        return
    self.left = Tree(
        depth=self.depth + 1, max_depth=self.max_depth, min_size=self.min_size
    )
    self.left.fit(self.X[s], self.Y[s])
    self.right = Tree(
        depth=self.depth + 1, max_depth=self.max_depth, min_size=self.min_size
    )
    self.right.fit(self.X[~s], self.Y[~s])
```

3.9 Predict

If the search to classify a new feature vector `x` (note that this is normal letter, not a capital) along the trees reaches a terminal tree, then return its majority vote. Otherwise, for that tree I know I have to split along the optimal column `self.split_col` and compare `x[self.split_col]` with the best value `self.split_value`. Depending on the outcome, I can leave the rest of the prediction to the left or right tree.

The method `predict` is a convenience function that can be applied to a matrix X, say, of data, not just one observation.

```
def _predict(self, x):
    if self.terminal():
        return self.majority_vote()
    if x[self.split_col] <= self.split_value:
        return self.left._predict(x)
    else:
        return self.right._predict(x)
```

3.10 Terminate the split

I need to know when a tree is a terminal(end) tree, because when the search has reached a terminal node, the tree should return a classifier of the terminal node. So, when is a tree a terminal tree? Well, if a tree doesn't have a left and/or right tree, this tree is not split, hence must a be terminal node.

```
def terminal(self):
    return self.left == None or self.right == None
```

3.11 Majority vote

Here I use a majority vote to classify. `np.unique` gives all unique values in `self.Y` and also how often they occur. So, `np.argmax(count)` gives the index of the value that occurs most. If I return the value with that index, I get the values that occurs most often.

```
def majority_vote(self):
    values, counts = np.unique(self.Y, return_counts=True)
    return values[np.argmax(counts)]
```

3.12 A test

Here is some test code.

```
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))
```

We'll use the `Tree` class again when we will deal with boosting. The tree class we will use then is more difficult because we have to deal with weighted score functions.

4 Exercises

1. Why is Quicksort an $O(n \log n)$ algorithm? (This helps you a lot in understanding how recursion works!)

2. If $n = 10^6$ (about the number of student ids in the data bases of our university), show that sorting with buttlesort takes about 10^5 as long as with quicksort. Hence, memorize that $O(n^2)/O(n \log n)$ becomes big when n is big. BTW, the performance of many algorithms without memoization is terrible. Use memoization whenever you can.

3. Think hard about how recursoins work classes.

4. Define (and do) some tests for the `Tree` class. Here are some simple (and dumb) things to test:

   (a) Test on `min_size=0`? Should the tree crash? What should happen in that case?

   (b) Test on `max_depth=np.infty`? Then the size of each branch should be at most `min_size` large. Is that indeed true?

   (c) What happens if you pass `X=[]`, i.e., an empty array to the tree?

   (d) What if all values of `X` are the same?

   (e) What if all values of `Y` are the same?

   (f) What if just one value of `Y` is different?

   (g) Can we mix positive and negative values in `Y` and/or `X`? Of course, all should work, but does it actually?

   (h) If you like, add some error checking. For instance, ensure that the shapes of X, Y and w are compatible. How should we deal with divisions by zero problems?

5. Compare the code of DSML and this site to my code.

   (a) What parts of each code do you like? Why?

   (b) Can you see what ideas of the others I copied?

   (c) Do you understand why I changed certain parts?

   (d) Think about how you would build your code based on these three examples of code. And if you're into R, then try to build your own tree class with recursion.