Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Querétaro

Programming Languages
Final Project

Feb - Jun 2021

Generating a Decision Tree

Professor: Benjamín Valdés Aguirre PhD

Sandra Román Rivera
A01702863

Table of contents

Context of the problem

Decision Trees

Decision trees are Machine Learning algorithms that can perform classification, regression, and even multi output tasks. They are really powerful, they can fit complex datasets. In this project, we'll use decision trees as a classification algorithm. This means that given some data, we'll be able to tell the class that the given data belongs to.

As it name indicates, a decision tree is basically a tree structure. This tree has attributes of a dataset as nodes. The node's branches represent the values that node can have. You go down the tree following the branches that best suit the data object you want to classify until you reach a leaf node. That leaf node will contain the predicted class for your data object. Pretty simple, right?

Figure 1

Decision Tree for "weather" dataset

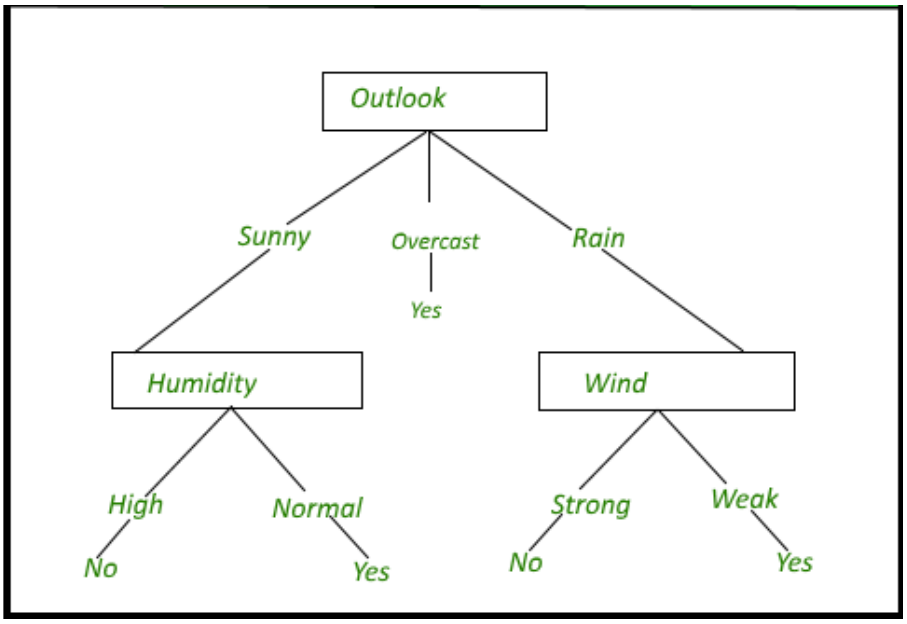

Figure 1 (GeeksforGeeks, 2019) shows what a decision tree actually looks like. This tree was built based on the weather dataset which you can find in the project's repo following this link. This is a really simple dataset, so the generated tree is also very simple. Here we have 3

attributes: outlook, humidity and wind. Each of the attributes can have multiple values. For example: the outlook attribute has sunny, overcast and rain as its values.

If we want to know whether we should play tennis or not, according to this decision tree we should take into consideration 3 things: what the outlook, humidity and wind are like today. If the outlook is overcast, we should definitely play tennis, no more assessments are needed. However, if it's sunny we should evaluate humidity and if it's raining, we should evaluate wind in order to make our final decision.

You may be wondering what the problem is, decision trees seem to be pretty awesome (and they are). Well, the problem is that decision trees requiere a lot of recursive calculations and since good datasets tend to be really big, they need to perform these calculations for huge amounts of data. To get you to understand the magnitude of the problem, I'll walk you through ID3, an algorithm to build decision trees that fit a specific dataset.

Building a decision tree

As mentioned before, we'll be explaining how to build a decision tree using the ID3 algorithm, so that we can better understand the complexity of building decision trees. So let's dive right into it.

In order to get this algorithm we first need to understand what entropy is, how it is related to information gain and why information gain is important.

Let's start with entropy. In this case, entropy is a value we use to know how diverse or uniform is a dataset or part of it. A set's entropy is zero when it contains instances of only one class. Entropy increases as the instances of the datasets are distributed between multiple classes. The formula to calculate entropy of the i^{th} node of a decision tree is given by:

$$Entropy(X) = -\sum_{i=0}^{n-1} P_i log_2(P_i)$$

$$P_i \neq 0$$

Where n is the number of different values an attribute has and P_i is the frequency of the i^{th} value divided by the total number of instances in the evaluated dataset.

Now that we know what entropy is and how to get it, we can move on to information gain. Information gain was first mentioned in Shannon's information theory. He used it to calculate the loss in the beeps of the telegraph. We'll be using it for a whole other thing, but thanks Shannons anyways. In this case I think it's better to first take a look at the formula to calculate it and then explain it. So here it is:

$$Gain(T, X) = Entropy(T) - Entropy(T, X)$$

Where Entropy(T) is the entropy of the system and Entropy(T,X) is the entropy given a certain attribute. What that means is that we need to calculate a subentropy. This is done by selecting an attribute X and performing the same entropy calculation, but now once per attribute's value. Maybe you can now start to notice how big the number of calculations can get.

Going back to information gain, we can now see that it is the difference between the system's entropy and one of its subentropies (a system has as many subentropies as attributes). This value indicates how much knowledge we get by using a certain attribute. The more knowledge we gain, the better.

Knowing this, we want to know which attribute gives us the most knowledge. For this, we calculate the gain for every attribute in the system. The number of calculations continues to grow. Once we have al the gains, we choose the greatest one and use that attribute as the root of our tree. All these calculations and we only know the root!

When we decided on the root, it's time to reorder the dataset using the root's attribute and split it into different subdatasets (one for each value of the attribute). We repeat the whole process for each of the subdatasets we got, ignoring the attribute we've already used. The attributes we get will now be the next level of the tree. We continue to recursively do this whole process until one of three things happen:

1. We end up with a system entropy equal to zero.
2. We reach the maximum level specified by the user.
3. We run out of attributes.

Hopefully by now you are aware of the humongous amount of computations needed in order to build a decision tree. Of course it varies depending on the size of your dataset. But, as mentioned before, good datasets tend to be huge.

The documentation on Scikit Learn's decision tree indicates that the total cost of building a decision tree is $O(n_{features}n^2_{samples}log(n_{samples}))$. In this report I'm referring to features as attributes and to samples as instances. That's a lot! Thankfully, many of these operations can be performed in parallel. That and the computer power we can get now a days is how we can build decision trees in a reasonable amount of time. I'll get into this in the next section.

Solution

As mentioned before, building a decision tree requires a lot of calculation. Luckily, some of these can be performed in parallel. The ideal way to parallelize computations is using a GPU and, therefore, CUDA. I didn't choose to implement this algorithm in CUDA because I don't have a GPU and using Google Collab would slow down my workflow significantly. However, if you want to get the most out of parallel computing to build your trees, go ahead and use CUDA.

Taking into account that I have the limitation of working on CPUs and not GPUs, I decided to tackle the problem using concurrent programming instead of parallel programming. I used Java to take advantage of it's ease when it comes to managing threads. Also, I implemented several different ways of multi-threading the code. I'll get into each of them in detail in a second, and I will show the comparisons between all of them in the "Results" section.

One thing all the different approaches I took when multi-threading the building of the decision trees have in common is the use of a thread pool. This allows the code to reuse the created threads, instead of creating new ones every time. This will help us decrease the overhead caused by thread creation. Furthermore, in all of them, I use concurrency only when calculating subentropies (for the main dataset and all of the resulting subdatasets). Knowing this, let's take a look at the different approaches I took.

Thread per attribute

This was the first approach I took and the one I consider to be the simplest. In this approach, every time we need to calculate the subentropies, we create a thread for each attribute and let them all calculate each of its values' relative entropies. The entropy and gain for the attribute are set within the thread.

For the pool I used an ExecutorService object initialized with a CachedThreadPool which creates new threads as needed and will reuse available threads. I chose this one because this way I don't need to fix the maximum number of threads.

What I like about this approach is that we're not likely to fall into race conditions. This is because every thread modifies a different attribute and its values. Threads don't interfere with each other.

What I don't like about this approach is that we need to shutdown the pool in order to wait for the threads to finish. This means that every time we recursively calculate subentropies, we need to initialize a new thread pool. In this way, the thread pool is not helping that much, since all the available threads created for the previous execution won't be available on the next one. This causes a lot of overhead.

Thread per value

This approach is really similar to the previous one. The only difference is that, instead of having a threads calculating each of the attribute's subentropy, we have a threads calculating each of the relative subentropies for each value of each attribute. This results in a significantly larger number of threads.

To explain this a little bit better, know that we need two nested for-loops: one that loops through the attributes, and inside that one, another one that loops through that attribute's values. The thread per attribute approach creates the threads in the first for loop and the second one is done inside each thread. The thread per value approach creates the threads inside the nested loop.

What I like about this approach is that the Runnable object's code is simpler and, therefore, easier to debug. It also takes advantage of the thread pool since we'll need many threads and hopefully we'll be able to reuse many.

I was trying to think of other pros of this approach, but I actually don't like it that much. I don't think creating this many threads is that good since we can't run that many at the same time. This will make the code run almost sequentially, because there will be a lot of threads waiting for a few threads to finish, creating a bottle neck. Also, since we used the same ExecutorService as before, we need to shutdown the pool too and initializing it again the next time. So we get this overhead as well. Furthermore, in order to avoid race conditions, we need to calculate each of the attributes' entropies and gains once all threads are done executing.

Fork Join

Fork Join seems like a more suitable approach. It is designed for work that can recursively be split into smaller pieces. It uses a work-stealing algorithm, which makes it faster. The idea of work-stealing algorithms is for free threads to help busy threads finish their work. I wish humans followed a work-stealing algorithm. This way someone would be helping me finish all my projects, but well…

Getting back to it, I implemented two versions that use Fork Join. One of them executes a RecursiveAction and the other one a RecursiveTask. They're both kind of similar, but I'll get into more detail to showcase their differences. For now, just know that the need for pool shutdown is gone, we use a ForkJoinPool for this approach.

Recursive Action

This approach is the one that makes the most sense to me. We start by calling an action per attribute. Inside the action we get all the values of that attribute in an array and we recursively split the array in half until its size is lesser or equal to a threshold (which I defined to be equal to 3). Once we are done dividing the action into subactions, we calculate the relative entropy of each of the 3 or less values we ended up with.

That was the fork. The join in this case does nothing. You may be wondering why the join does nothing. Well, I'm glad you asked. The thing that would make sense for us to do in the join is add up all the relative entropies to calculate the attribute's subentropy and gain. Unfortunately, we cannot do that because we would need to be modifying the same attribute from multiple threads. This could lead to inconsistencies due to race conditions. So, running the program with the same inputs could give different outputs (and that's not what we want).

In this case, we're not taking advantage of the join, but we do take advantage of the fork and the work-stealing algorithm. When all values are done calculating their subentropies we can calculate the attributes' entropies and gains. For this, we ask the program to wait for the pool. Fortunately, as mentioned before, we don't need to shutdown the pool this time, so we'll be able to reuse it later.

Recursive Task

A recursive task works almost the same way as a recursive action, the only difference is that a recursive task returns a value and a recursive action doesn't. In this case, instead of just setting the value's subdataset and subentropy, we also start to add the subentropies in every subtask. Therefore, when the recursive task is done, we already have the total entropy of the attribute, we just need to calculate its gain and store them both for future reference.

In this case, we can calculate the attributes' entropies in the join, because we are able to return the calculated value instead of having to save it to the attribute directly. This is why we're not falling into any race conditions.

This may seem like a better approach than a recursive action, since we're calculating the entropy in the task and just need to set it when it's done. The problem here is that we need to return a value and store it, so a new task cannot be invoked before the previous one is done. This will calculate entropies faster, but won't let us calculate multiple entropies at the same time. It will execute a task, wait for it to faster calculate the entropy, we'll set it in the attribute and then execute the next one.

Results

By now, we know what the approaches taken were and we kind of understand some of their pros and cons. Let's now compare their execution times to see which is actually faster for different types of datasets.

The way the results were measured was pretty simple. We counted the number of milliseconds it took for each of the recursive calls to calculate all subentropies and added them together. In the end, we're left with the total time the program spent calculating the needed subentropies. The time is given in milliseconds.

Each approach was tested 10 times using three different datasets: weather, iris and titanic. You can follow the links to see what each of them looks like. We don't actually care about what the datasets represent, we just care about their sizes. Let's now analyze this:

1. Weather: this is the smallest and simplest one of the three. It has only 14 instances and 4 attributes. Each attribute has 2 to 3 possible values. The size of the dataset can be represented as 14 x 4, resulting in a total size of 56.
2. Iris: this dataset has 150 instances and 4 attributes as well. Besides the fact that it contains way more instances than the weather one, its attributes have way more different values. This is because the values are not categories, they're decimal numbers. I didn't change this and grouped them into categories because I want to know how having tons of different values affect the process. The size of this dataset is 600.
3. Titanic: this last one is the one with the most instances (1309) and attributes (6). However, I did use one hot encoding so that the age attribute didn't have as many different possible values. The number possible values for the different attributes goes from 2 to 8. This dataset has a 7,854 size.

To sum up, each dataset is around 10 times bigger than the previous one and has different instances/attributes proportion. Now let's take a look at the results.

Sequential

Table 1 shows the results gotten from executing different datasets multiple times calculating subentropies sequentially. Average time for the weather dataset is 4.8 ms, 10.4 ms for the iris dataset and 48 ms for the titanic one. As we can see, the average time increases exponentially as the dataset gets bigger. Times don't vary that much within each dataset.

Table 1

Time spent calculating subentropies in ms for different datasets using sequential approach

	Weather	Iris	Titanic
1	5	11	47
2	5	10	48
3	5	11	47
4	5	10	48

	Weather	Iris	Titanic
5	5	11	49
6	4	9	48
7	5	10	49
8	5	10	49
9	4	11	49
10	5	11	46
Average	**4.8**	**10.4**	**48**

Thread per attribute

Table 2 shows the results gotten from executing different datasets multiple times creating a thread per attribute to calculate their subentropies. Average time for the weather dataset is 4.8 ms, the same time as the sequential program. The iris dataset has an average of 8.8 ms. Here we can see an improvement of 1.6 ms. Finally, the titanic dataset took an average of 68.2 ms to calculate subentropies. This represents a 42% increase in time compared to the sequential approach.

Table 2

Time spent calculating subentropies in ms for different datasets using thread/attribute approach

	Weather	Iris	Titanic
1	4	9	69
2	3	8	73
3	5	9	66
4	6	10	68
5	5	9	65
6	4	9	70
7	6	9	68
8	5	8	70
9	5	9	71

	Weather	Iris	Titanic
10	5	8	62
Average	**4.8**	**8.8**	**68.2**

While performance was better than sequential for the iris dataset, this approach is overall slower than or equal to the sequential one.

Thread per value

Table 3 shows the results gotten from executing different datasets multiple times creating a thread per attributes' value to make the subentropy calculations. Average time for the weather dataset is 5.8 ms, 1 ms slower than sequential. The iris dataset has an average of 11.6 ms, also around 1 ms (1.2 ms) slower than sequential. The third dataset (titanic) took an average of 89.1 ms to calculate subentropies. This represents a 85% increase in time compared to the sequential approach.

Table 3

Time spent calculating subentropies in ms for different datasets using thread/value approach

	Weather	Iris	Titanic
1	5	14	88
2	5	12	84
3	5	13	93
4	7	11	97
5	6	10	88
6	6	11	92
7	6	11	86
8	5	12	89
9	6	11	85
10	7	11	89
Average	**5.8**	**11.6**	**89.1**

This approach is much slower than sequential.

Fork Join

Recursive Action

Moving on to ForkJoin approaches, table 4 shows the results gotten from executing different datasets multiple times creating a RecursiveAction per attribute to make subentropy calculations. Average time for the weather dataset is 4.6 ms, 0.2 ms (4%) faster than sequential. The iris dataset has an average of 8.4 ms. This is 2 ms (19%) faster than sequential. The third dataset (titanic) took an average of 42.8 ms to calculate subentropies. This represents a 11% decrease in time compared to the sequential approach, it was 5.2 ms faster.

Table 4

Time spent calculating subentropies in ms for different datasets using recursive action approach

	Weather	Iris	Titanic
1	4	8	45
2	5	9	45
3	5	9	45
4	5	8	45
5	5	8	39
6	4	8	46
7	5	8	44
8	4	9	43
9	5	9	38
10	4	8	38
Average	**4.6**	**8.4**	**42.8**

The use of ForkJoin with RecursiveAction achieved a 4 to 19 percent faster calculation of subentropies than the sequential version.

Recursive Task

In the approach that invokes a recursive task per attribute to calculate subentropies, results were as shown in table 5. Average time for the weather dataset is 4.2 ms, 0.6 ms (12.5%) faster than sequential and 9% faster than the previous ForkJoin approach. The iris dataset has an average of 26.6 ms. This is 16.2 ms slower than sequential. The third dataset (titanic) took an average of 55.5 ms to calculate subentropies. This represents a 16% increase in time compared to the sequential approach, it was 7.5 ms slower.

Table 5

Time spent calculating subentropies in ms for different datasets using recursive task approach

	Weather	Iris	Titanic
1	3	26	55
2	5	29	55
3	4	27	58
4	5	27	55
5	4	26	55
6	4	25	52
7	3	28	51
8	4	27	58
9	5	25	59
10	5	26	57
Average	**4.2**	**26.6**	**55.5**

The use of ForkJoin with RecursiveTask achieved a 12.5 percent faster calculation of subentropies for the weather dataset than the sequential version. However, it is overall slower than the sequential version.

Conclusions

As we can see, most of the concurrent implementations of subentropies calculations (an essential part when building decision trees) are slower than the sequential approach. This is

due to the limit in the number of threads we can concurrently run and the overhead caused by thread creation (and, in some cases, thread pool creation).

The only approach that proved to be faster than its sequential version was the use of recursive actions when using ForkJoin. This doesn't mean that this approach doesn't have any overhead. It does have overhead, but the reduction of processing time was enough to counteract and even defeat the lack of overhead on the sequential approach.

Another approach I would use, but only with really small datasets, is the ForkJoin that uses recursive tasks. This also demonstrated a reasonable increase of speed for the weather dataset. However, as mentioned before, real life datasets are almost never this small, so let's stick to the other ForkJoin version.

When it comes to the normally threaded approaches, I wouldn't recommend creating a thread per value approach, since it proved to be much slower in all tested datasets. If we scaled this, the results would be terrible. The awfulness of this approach falls back into the bottle neck created due to not being able to execute that many threads at a time and the overhead caused by thread and thread pool creation.

On the other hand, creating a regular thread per attribute is recommended for datasets with little attributes and many many different attributes (like the iris dataset). This proved itself to be slightly faster than the sequential version. This is due to the fact that not many threads will be needed and each thread will do lots of work, so the overhead of its creation won't be that significant.

Of course there are many other ways to concurrently calculate subentropies that must be, like the use of recursive actions, faster than a sequential approach. I just decided to implement this simple ones that are already built within Java so that I could compare and contrast many of them instead of just one or two implementations that are more complex.

To sum up, I would advice to use a parallel approach to this problem, using CUDA if possible. This is one of those problems that can actually be run in parallel since every attribute can calculate its subentropy in an independent way, that's why there is almost no need for thread synchronization in this approaches. As mentioned in the context of the problem, I didn't use CUDA because I don't own a GPU and using Google Collab would slow down my work flow

considerably, but I'm certain that this project can have incredible results if programmed in parallel.

How to run it

There are two main ways in which you can run the project: downloading the whole project or downloading just the zip that contains the jar (which I recommend).

The zip with the jar file can be found here. Just download the zip from there and extract it wherever you want. Open the terminal in the "jar" folder and run the following commands:

```
$ chmod +x ConcurrentTrees-1.0-SNAPSHOT.jar
$ java -jar ConcurrentTrees-1.0-SNAPSHOT.jar
```

The other way is to download or clone the repository (which can be found here). Once you have it in your computer, you can import it as a maven project to you favorite Java IDE (ex. Netbeans, Eclipse, IntelliJ) and run it as you would normally do.

One of those two ways should do the trick. You should now be able to enter a dataset's name (either weather, iris or titanic), the max depth you want your tree to have and select the approach you want to use. The built decision tree will be shown and, after it, the time your computer spent calculating subentropies.

References

GeeksforGeeks. (2019, 17 abril). *Decision Tree.* https://www.geeksforgeeks.org/decision-tree/

Géron, A. (2017). *Hands-On Machine Learning with Scikit-Learn, Keras, and Tensorflow: Concepts, Tools, and Techniques to Build Intelligent Systems* (1st ed.). O'Reilly Media. http://index-of.es/Varios-2/Hands%20on%20Machine%20Learning%20with%20Scikit%20Learn%20and%20Tensorflow.pdf

Oracle. (s. f.). *Concurrency.* The Java Tutorials. Recuperado 9 de mayo de 2021, de https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Oracle. (2020, 24 junio). *Executors.* Java Platform SE 7. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html