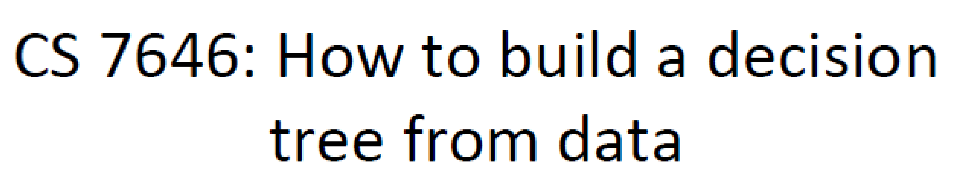
03-Live2 Decision Trees

Compiled by Shipra De, Fall 2016

- Hi everybody. Welcome. Sorry for my delay in getting started. I make some of these videos from my basement and I got down here to discover some of my technology was missing because of a certain child of mine. So anyways I had to find that and plug t it all in. Now I'm all ready to go.
- Ok, I will check periodically on Piazza to see if we've got any questions. One thing I would appreciate is if somebody who is a currently watching could post a question on piazza indicating that you can hear me. I'm going to start blazing forward anyways but it would be good to get that feedback, so thanks. And I'm going to check on Piazza right now. Okay, somebody says they can hear. Thank you.
- Alright I'm going to now step through some PowerPoint slides. Ok let's see what happens when I go full screen. Give me just a moment here. Ok. I put together a couple of PowerPoint slides I'll step you through.



CS 7646: How to build a decision tree from data

- So if you remember the last lecture we talked about if you had a decision tree how you might use it. So in this lecture we're going to talk about, okay, how do we build the decision tree in the first place?

Decision Trees

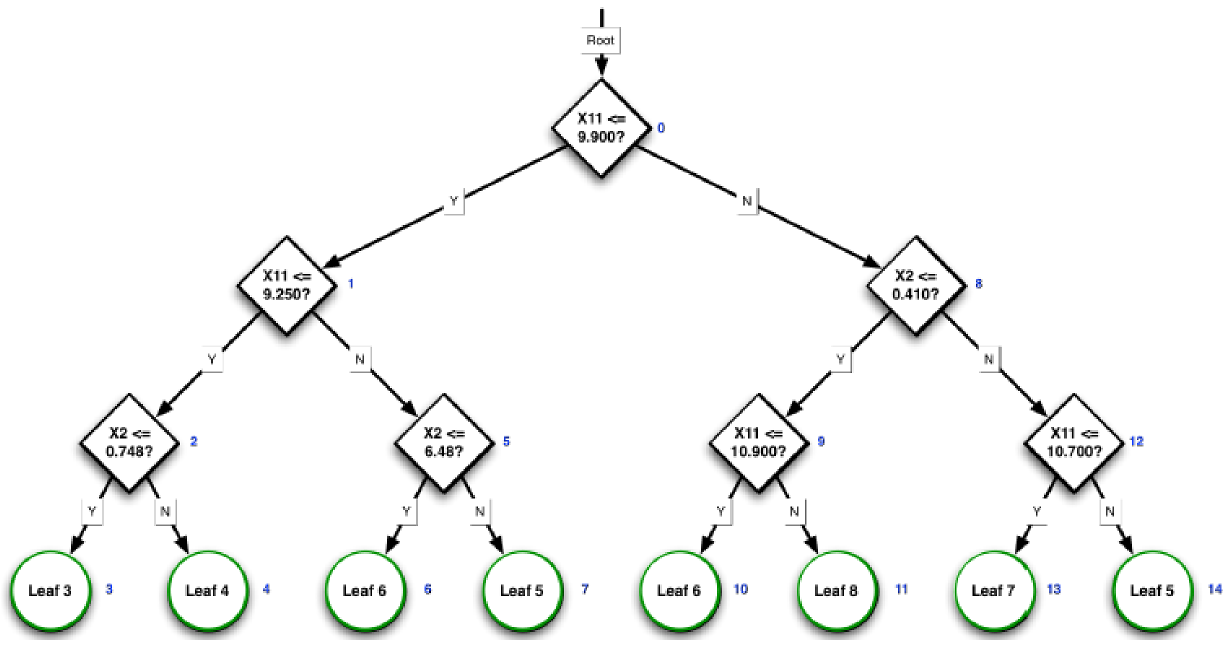
- List of factors or attributes X1… XN
- Labels Y
- Decision nodes
  - Binary decision: Xi <= SplitVal?
- Outgoing edges
- Leaves: Values

- Built from data examples <X1, X2,…, XN, Y>

- So let's pause for a second and think about what are the things that make up a decision tree . And big picture, remember here, we're trying to build a model that from some factors or features or attributes, things we can measure about a system, what is going to happen later.
- And so the factors or attributes are our X's and we might have many of them. We might have up to N of these factors.
- Now associated with each group of factors are a label, Y. So our data consists of the values of the factors and the values of the labels. Think about it like a big matrix where the columns are the X's and then that last Y and the rows are individual samples of data. So each row might represent a day for the price of the stock or each row might represent a day of weather.
- Then using that data we build a decision tree and the decision tree consists of decision nodes. Each node represents a binary decision. Now we can build decision trees that aren't necessarily binary. They might have more than two outgoing edges, but for simplicity (and actually turns out for efficiency) it turns out that building binary trees is the best way to go. So we're only gonna have two potential outputs of each decision node.
- There's then the outgoing edges that lead to additional decision nodes. And there are two kinds of special nodes. There is the root node, which I forgot to mention here, and also leaves. So when you finally reach a leaf at the very end, that's the value that you're trying to predict. The root is the first node of the very top that you start at when you're trying to make a decision.
- And we build these decision trees from data examples and again we have each row in our data is a set of X's and a Y.

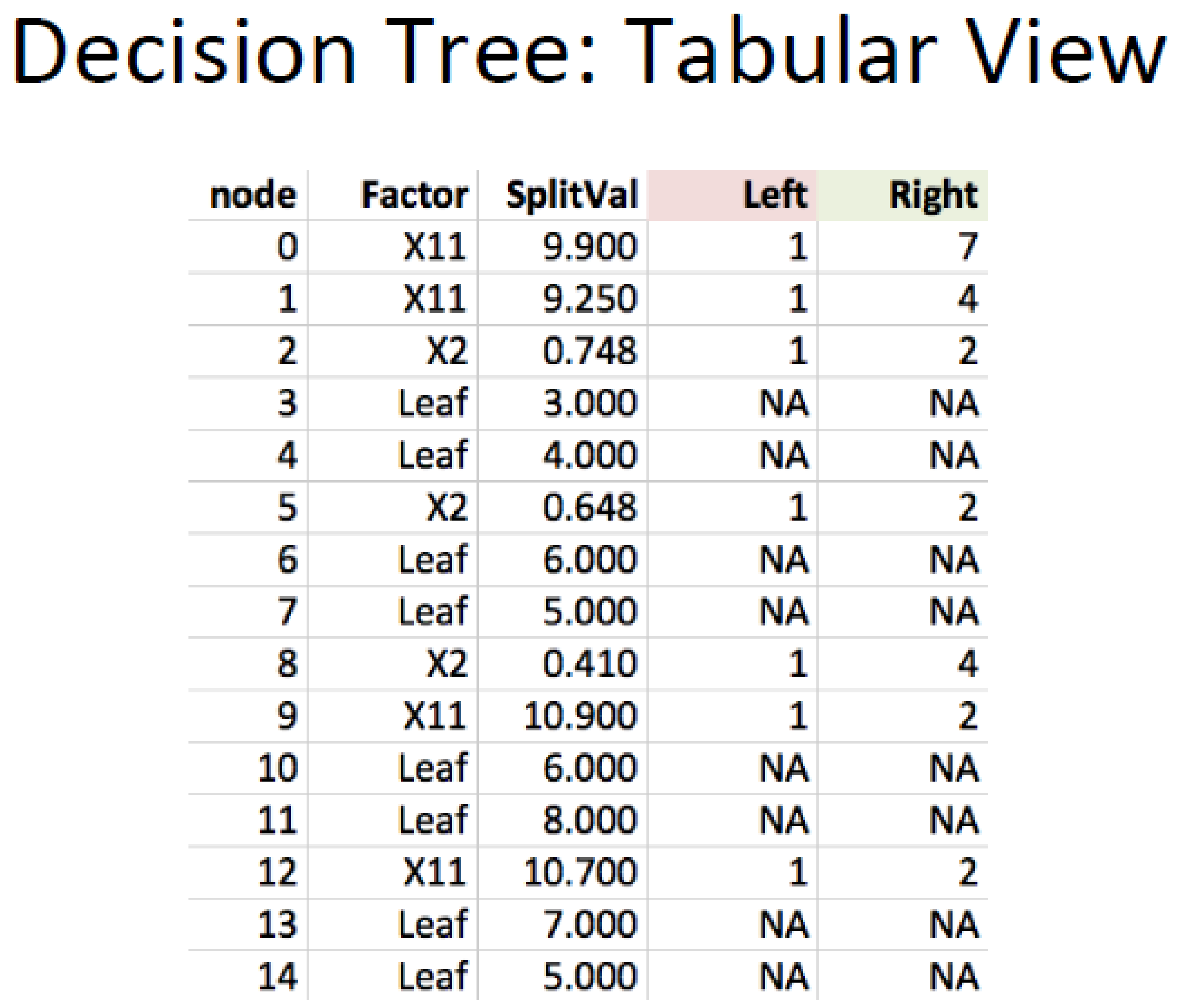
- Now I'm gonna double-check that you are able to see my PowerPoint slide. I don't want to get all the way through this and discover the people aren't seeing the slides. I'm gonna go take a quick check at piazza. People must see the PowerPoint because they're asking if they can have it. And the answer is yes. You'll be able to have it. I'll come back a little bit later to answer the other questions that are popping up. Great, thank you very much for answering.

Decision Tree: Graphical View



- Ok, here's an example of a decision tree. We've identified the root here up at the top. So whenever we want to query our model to see what the value, what the predicted value, is, we start here at the root and each decision node asks a binary question. Is this factor X11 less than or equal to 9.9?
- If it is, yes, we go down this left edge and we come to another question about that same factor X11. And if it's less than or equal to 9.25, we go left. If it's not we go out to the right.
- And here we see a new factor that we're asking questions about, X2. If it's less than or equal to 6.480 we go down here. We arrive at a leaf and the value is six.
- Now couple things I want to point out. This is an example tree. I'm going to actually show you in a few minutes how we built this exact tree, and it's using that wine data that I introduced the other day. So X11, for instance, is a percentage of alcohol.
- I forgot what X2 is, but the thing to note here is this particular tree was only built using two factors. And in some cases the same factor (well in all cases) the same factor appears multiple times through the tree. So for instance, factor 11 appears here and here. So if we go down this branch we end up asking two questions about factor 11. If we go down this branch again we end up asking two questions about factor 11.

- So we can repeat factors. Sometimes not all factors are included in the tree and sometimes some factors appear more than others. It turns out that that's often related to how predictive the individual factors are.



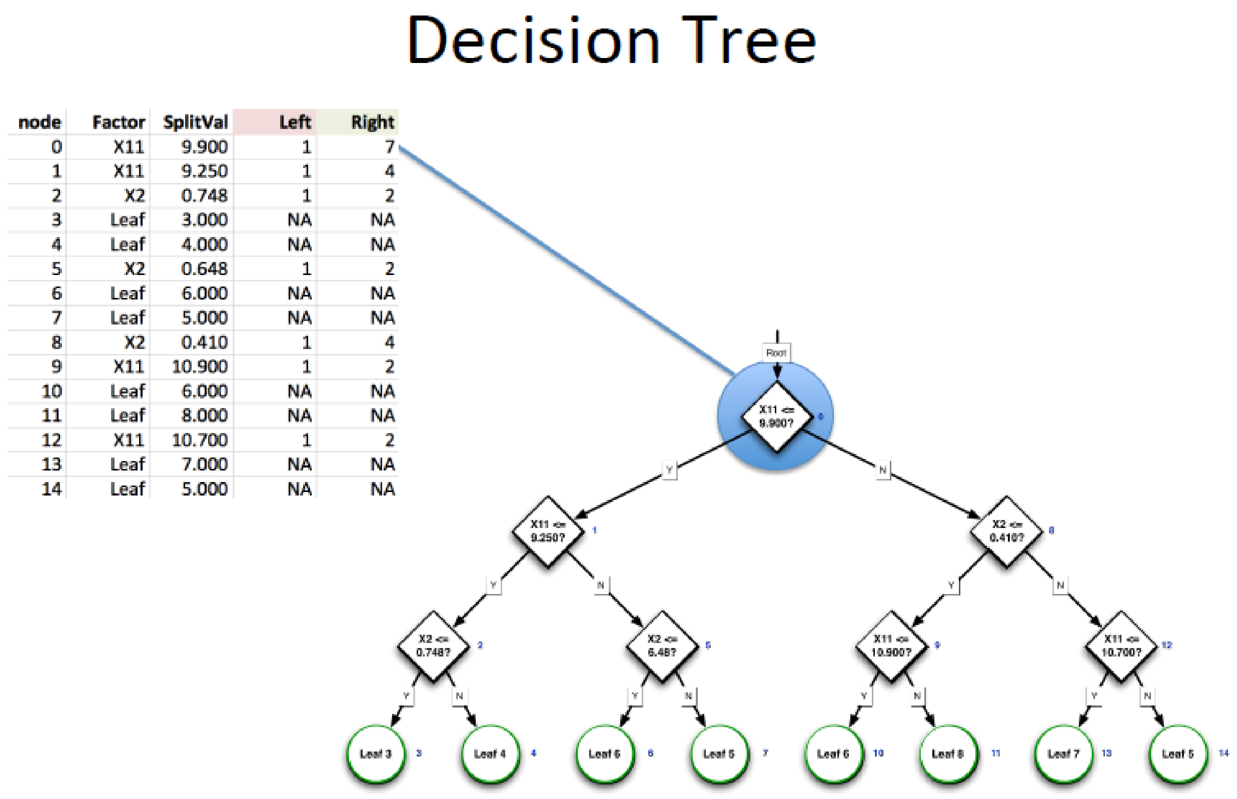
Decision Tree: Tabular View

node	Factor	SplitVal	Left	Right
0	X11	9.900	1	7
1	X11	9.250	1	4
2	X2	0.748	1	2
3	Leaf	3.000	NA	NA
4	Leaf	4.000	NA	NA
5	X2	0.648	1	2
6	Leaf	6.000	NA	NA
7	Leaf	5.000	NA	NA
8	X2	0.410	1	4
9	X11	10.900	1	2
10	Leaf	6.000	NA	NA
11	Leaf	8.000	NA	NA
12	X11	10.700	1	2
13	Leaf	7.000	NA	NA
14	Leaf	5.000	NA	NA

- Now this is one way to view a tree. We can also view it in a different way, in a tabular view. And this data right here represents the same tree but in a tabular view and this is the way I want you to build your decision tree for the next assignment. I want you to use a NumPy array.
- And I want you to consider these four columns.
  - The first column is, for that particular node which factor are we considering.
  - The next column is what is the split value. In other words, which value do we decide to go down the left branch or right branch.
  - And then these two other columns tell you where in my matrix here does the next part of the tree begin. So here we're saying the left tree starts in the next row and the right tree starts at the eighth row.
- Now leaves are special entries. So if we have here in the first column an indication that it's a leaf, the next column split val, instead of it representing a value to split on, it represents the

value of the leaf. And it doesn't really matter what's in the left or right columns here for a leaf because we're not going down further.

- One more thing I want to say. Yes in this example we have sort of text in this column but numpy arrays can only be numbers. This is just for, sort of clarity, and so you know instead of saying X11 here you would put the number 11. And instead of you know putting leaf here, you would put some special number that signifies to you that this is a leaf.

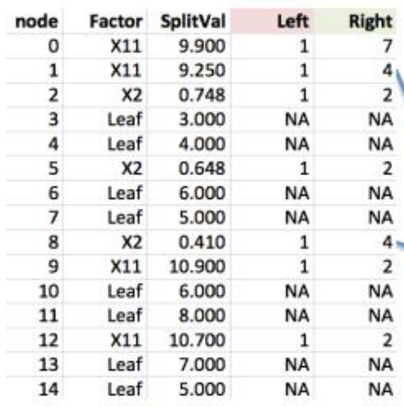


Decision Tree

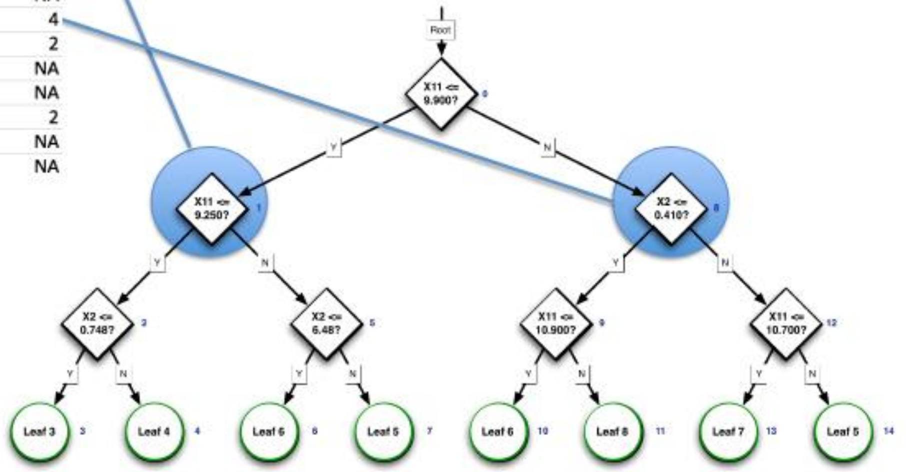
node	Factor	SplitVal	Left	Right
0	X11	9.900	1	7
1	X11	9.250	1	4
2	X2	0.748	1	2
3	Leaf	3.000	NA	NA
4	Leaf	4.000	NA	NA
5	X2	0.648	1	2
6	Leaf	6.000	NA	NA
7	Leaf	5.000	NA	NA
8	X2	0.410	1	4
9	X11	10.900	1	2
10	Leaf	6.000	NA	NA
11	Leaf	8.000	NA	NA
12	X11	10.700	1	2
13	Leaf	7.000	NA	NA
14	Leaf	5.000	NA	NA

- And again these are dual representations. They represent exactly the same information. The root node is always the very first row in this ndarray.

Decision Tree

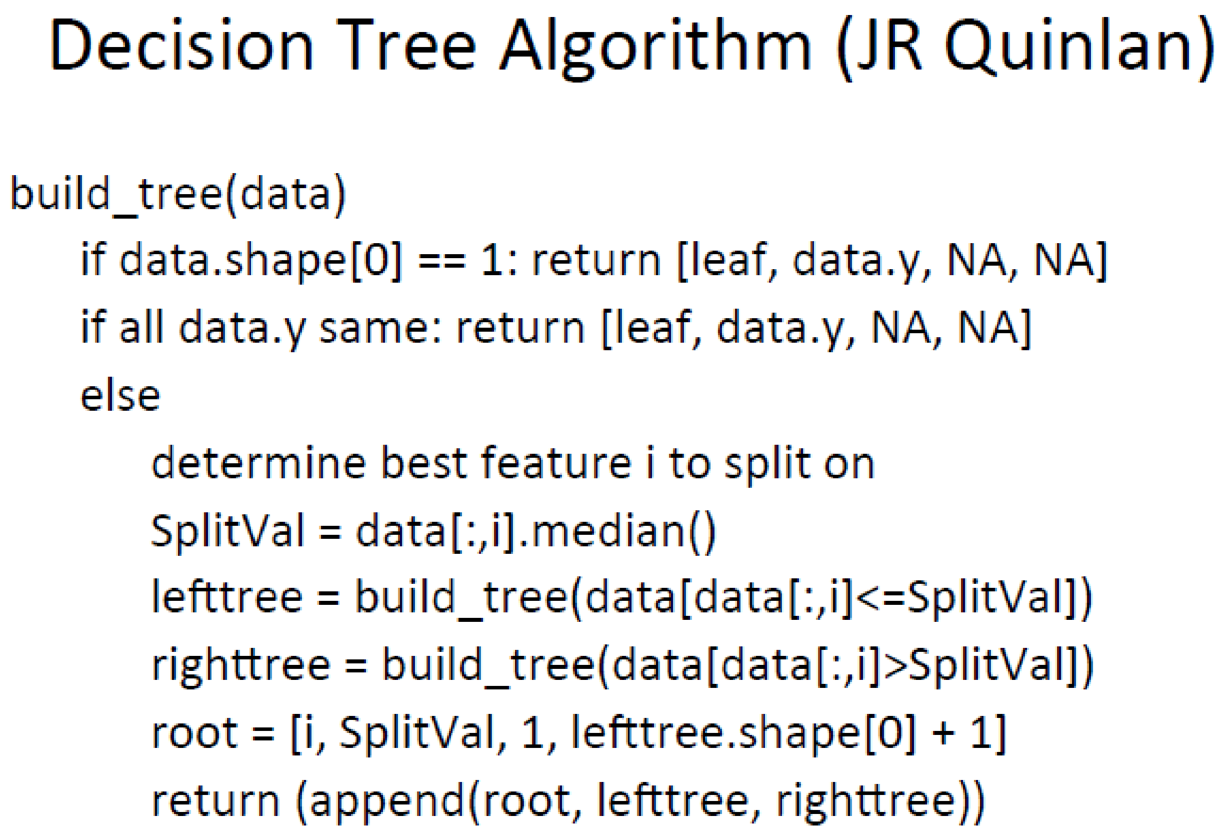


node	Factor	SplitVal	Left	Right
0	X11	9.900	1	7
1	X11	9.250	1	4
2	X2	0.748	1	2
3	Leaf	3.000	NA	NA
4	Leaf	4.000	NA	NA
5	X2	0.648	1	2
6	Leaf	6.000	NA	NA
7	Leaf	5.000	NA	NA
8	X2	0.410	1	4
9	X11	10.900	1	2
10	Leaf	6.000	NA	NA
11	Leaf	8.000	NA	NA
12	X11	10.700	1	2
13	Leaf	7.000	NA	NA
14	Leaf	5.000	NA	NA



- And then you can see here where the left and right subtrees are. So that's how we represent decision trees, or at least my recommendation on how you want to represent for your project.
- Now those who have sort of an object-oriented bent and want to do things in an object-oriented way, you can do that in Python, but let me suggest again, first do it this way. Get it working, and if you want to go forward with the object-oriented approach, that's fine. But get it working like this first.
- But now I'm going to take a quick look back to Piazza see if we have any questions and then I'll go on with the presentation. Ok people see the slides.
- Abida asks a question. Building binary trees is best in efficiency in terms of building the tree, you mean? Why is that? Why would a higher branching number be worse? So I meant actually that if the tree is binary it is likely a little bit easier to process. You know keep in mind each question...uh...so think about efficiency in terms of, you know, a CPU. If our questions are simple comparisons, that's something that modern-day processors can do incredibly fast. So if it's a binary tree they only have to do a single comparison at each node to decide what to do next. Now you can probably dig up some papers that might contradict what I'm saying, so I'm not going to assert too strongly, but I will say they're definitely easier to build. And it's my intuition that they're probably fast as well. But, yeah, sure, maybe a higher branching factor could perform better.
- The video of the slides is too blurry to read sometimes. Okay I understand. I'm going to share the slides and also the spreadsheet that I'm going to use shortly. I'll put that up on the website so you'll be able to see the PowerPoint as well.

- Somebody is enjoying porto. It's Patrick. Hi Patrick. So keep in mind everybody can see my screen right now, but I'm with you Pat. I'm glad you're enjoying it. Oh, and by the way, he's checking out he's doing an OPS test, ground truth, on the upcoming problem.
- In numbering nodes I see you're going by depth instead of breadth. Indeed, it doesn't matter. Actually I'm going to get to that in a little bit and you'll see why they have the numbers that they do. So ask that question again if you don't understand it or if I don't make it clear coming up.
- Ok let me go on with my PowerPoint presentation.



Decision Tree Algorithm (JR Quinlan)

```
build_tree(data)
    if data.shape[0] == 1: return [leaf, data.y, NA, NA]
    if all data.y same: return [leaf, data.y, NA, NA]
    else
        determine best feature i to split on
        SplitVal = data[:,i].median()
        lefttree = build_tree(data[data[:,i]<=SplitVal])
        righttree = build_tree(data[data[:,i]>SplitVal])
        root = [i, SplitVal, 1, lefttree.shape[0] + 1]
        return (append(root, lefttree, righttree))
```

- Ok, here is the algorithm for building a decision tree. This is the algorithm initially proposed by JR Quinlan and there's a link to this paper actually on the project wiki site if you're interested. It's actually a very good the paper in the sense that it's well-written, that it's not filled with jargon, it's easy to understand the motivation for how decision trees are built, and so on.
- So I know most people in OMS are come from a computer science background, so they are probably familiar with recursion. Some of the people taking the course on campus are from other disciplines, and actually hadn't necessarily seen recursion before.
- But this is a recursive algorithm and in other words, the algorithm or the function called build_tree actually calls itself and an important consideration in any recursive algorithm is you have to be sure it's going to terminate because otherwise it will call itself over and over and over again going to essentially infinite depths and never stop.
- So the very first thing that we do is a couple checks to see if we've met a stopping criteria.

- By the way, I want to emphasize that this is not the final totally complete description of how to build a tree. This is kind of an outline and you're probably going to find that you need to add a few more details, ok? But this is this will give you a good start.
- Anyways, the first things we check our stopping criteria. So when this function build_tree gets called with data, which is the data we're going to use to build the tree. And by the way this structure is an ndarray where each column represents a factor and the very last column represents the label or the Y value.
- So we take that data in and if the number of rows, in other words shape of [0] (that's the number of rows), if there's only one row that means we're being given data, essentially one sample of data. So if we are asked to build a tree out of one sample of data, well the answer is it's a leaf and the value of that leaf is the Y value of that data.
- And I'm using the same columns here that we talked about before in our columnar data structure. The next two items that don't really matter. So we create a single row. That is, again, we designate that it's a leaf. The label is Y. And we don't care about the other two items.
- So that's one stopping criteria. Another is what if we get called with some data and all of the labels are the same. Well, obviously, there's no point in creating a more detailed tree from this data because we know what the answer's going to be. We know it's going to be just Y. So again we can create a leaf and return it. So those are our stopping criteria
- Now, if neither one of those conditions are met, then we have real data that we can build a tree with.
- So first step here is determine the best feature to split on. So as an example if we're getting called for the first time to build a tree say using the wine data, this question is essentially, what is that first question that we're going to use to build our root node?
- Now it turns out that each sub tree is formally a tree as well. It has a root, it has leaves, it has directed edges, and so on. So we can imagine each sub tree is a tree by itself. But the very first time we do this we're looking at the root node. I'll talk in a minute about how do we determine that best feature. There's a number of different ways to do it, and it merits its own slide for discussing.
- Ok, now that we know which feature we're going to use, we need to determine what particular value are we going to split on. And the standard way to do this is to look at our data, look at that column i, remember that's the feature we decided to split on, and we look across all of our samples, and we take the median.
- The median is the value that divides the data in half. In other words, half the data is going to be less than the median, half the data is going to be greater than the median. There are special edge cases that you know people probably ask about. You know, what if the same value is repeated 10 times in the middle, you know, so you're going to have a sort of lopsided split of the data. Yeah, that's true. No big deal.
- Okay so we know our split val. Now we need to gather together the data we're going to use to build the left side of our tree and gather together the data that we're going to use to build the right side of a tree, and here's how we do it.

- And this is showing some of the real power of Python. So again, we're going to call our own tree building function here, build_tree, and the left tree is going to be built from the data that is less than or equal to the split val. This expression is called comprehension, and you can specify, "okay, look through my data, find all the elements where this column is less than split val, grab those columns out of data, and we will use that to build our left branch."
- Same thing for the right side except we're using those rows that are greater than split val. So each of these, you know, will call build tree in turn.
- So what will happen is we'll recurse down the left side of the tree, and we'll go down and calculate this left tree and boomp, it will return and we'll have this value, here lefttree, which is an ndarray. Similarly for the right side, we'll have righttree. So now we have two sides of our tree, the left tree and the right tree.
- We still need the root and here's how we compose the root. So first element here is what's the feature we're gonna split on. We determined that's i ,right? The next column is what's the value we're going to split on, that's split val. It turns out that in our formulation here, the left tree always begins at the next row.
- And one thing I want to mention that I didn't mention when I was talking about our data structure is, these left and right columns, it turns out that it's easiest if you think about them as being relative. So in other words, this is saying that the left tree begins in the next row. So whatever row we are in the matrix, we're saying the left tree starts at the very next row.
- Where is the right tree start? Well we have to make room for our left tree. So this is saying okay, we're going to make room for our left tree. And then the next row after that is where our right tree begins. Then we append those all together. We append our root, our left tree, and our right tree. Boom. That's our whole tree. Ta-da! The end.
- Ok. I'm sure there's probably some questions percolating. I'm going to go back to piazza and see if we have any questions.
- Again, Abida with another excellent question. Are we using median to make it as close to being log base 2 height as possible? Yes, that's right. Put another way, we're trying to keep the tree balanced, so if at each level we split the data exactly in half. Then the levels below, we'll split it in half, and so on and as we go on down at each level we'll be splitting the data appropriately so that the branches will remain balanced. Now you can't always do that perfectly, but yeah, that's the general idea.
- Where is the root in the code? When I was giving this presentation yesterday on campus, I defined the root, I think, up here. It turns out that as soon as you determine which feature to use in which split, you already have all the information you need to define the root. The first column is going to be the factor, the second column is going to be the split Val, the next one is always going to be 1, because the left tree always starts to the left . And you don't know until this line, until you do this computation, how large the left tree is. You could determine it by essentially executing this, but you don't know how many rows are in the left tree until you do this because the data size may be a little bit different.
- Ok so that is how to build a tree using Quinlan's approach. So now let's talk a little bit about how do you determine the best feature.

How to determine “best” feature?

Goal: Divide and conquer

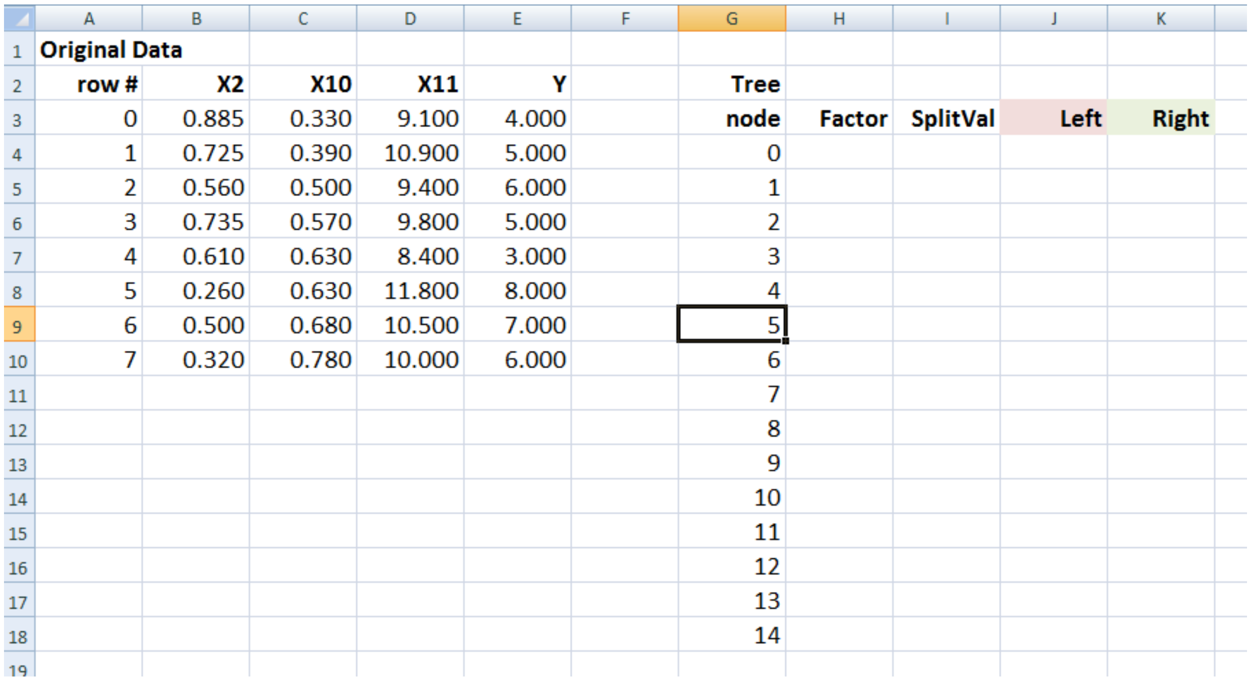
Group data into most similar groups.

Approaches:

- Information gain: Entropy
- Information gain: Correlation
- Information gain: Gini Index

- So in general what we want to do is we want to start with the feature that segregates the data the best.
- So imagine for a moment that we're doing classification. In other words we're trying to decide is it a buy or a cell for a particular stock. We would like to find that for this feature, that root feature that groups, you know that splits it into the most similar groups, and one group should be mostly buys and the other side should be mostly sells. So which factor can help us make that split most effectively?
- Then we go to this next problem of, ok, now I've got a group that has mostly buys, how do I distinguish those few cells from the other buys. And hopefully there's another feature at that level that will help us make that distinction.
- So at each level we want to somehow choose the feature that does this separation most effectively. And one way of thinking about that, or one term folks use to refer to it, is information gain. So which factor provides the most information about our data.
- And there's a number of different ways to evaluate that or estimate it. I believe the original Quinlan paper uses an approach called entropy. Even if they don't use entropy, that still is I think nowadays widely regarded as the most effective method of selection.
- Entropy is just a way to measure... you know, once you've segregated your data into groups, to essentially measure the diversity or randomness of each group. So, as an example, if each side had fifty percent buys and fifty percent sells, then we didn't really gain any information because it still is kind of randomly distributed. However if one side is all buys, and one side is all sells, that was the biggest information gain we could possibly get.
- And there are ways to measure entropy even for non-classification examples, you know where our data, instead of it just being labeled a buy or a sell is a regression sort of problem.

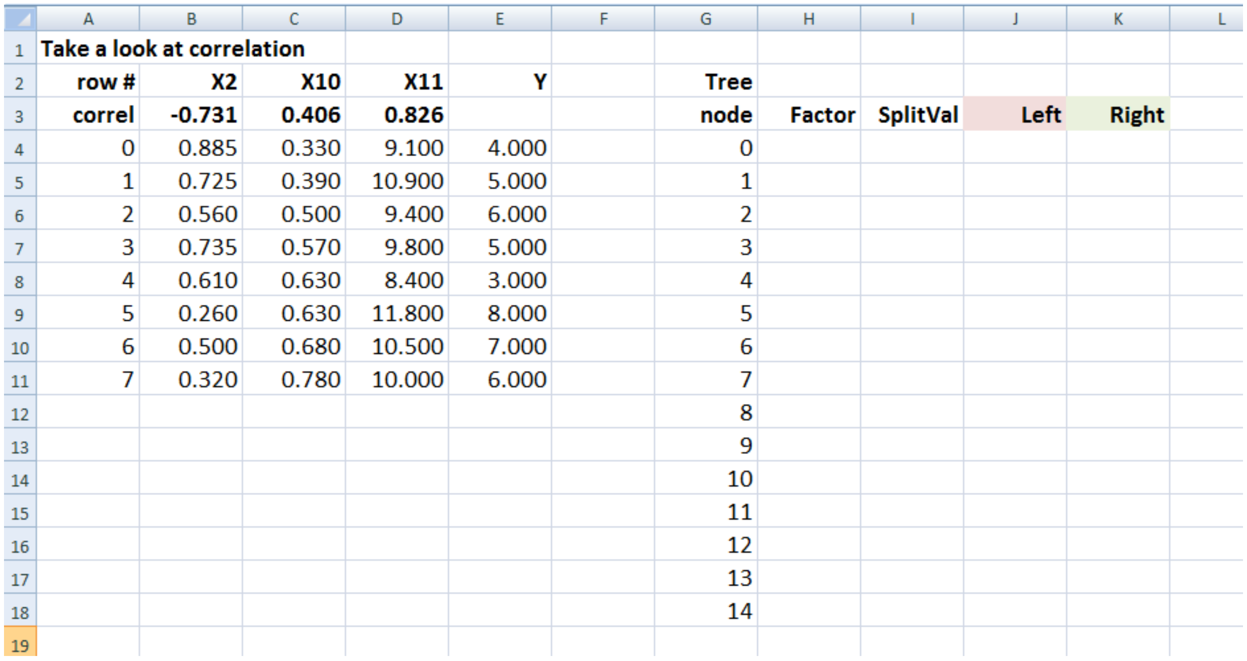
- Now, don't fear. I'm not gonna make you calculate entropy. For this project we're going to use correlation. And it turns out that correlation, actually, for regression problems is very closely related or correlated <ha-ha> with entropy. So what we can do is we can look at each factor in our data and see how well correlated it is with our labels. And the factor that is most strongly correlated is going to help us make that split most effectively.
- There's yet another method called the Gini index that is a essentially another measure of diversity that people use as well.
- Anyways, to summarize, I think entropy is sort of the de facto standard. Works very well. If you were to download and use the decision tree library in SciPy, you would find that they use entropy. I think you can choose between entropy and gini index.
- I want to emphasize, though, for this project you're not allowed to use the SciPy library. Sorry, you have to write your own.
- Okay.   I'm going to now give you an example of doing this with real data. But let me check and see if we got any questions before I do that.
- In the 'build_tree' algorithm, how does the append work? So for ndarrays, so look up the documentation for numpy and in particular, ndarrays and you'll see what append does. Basic idea is it creates a new array out of these three arrays. Puts root at the top, next left tree, finally right tree.   You may you may need to provide an additional argument essentially telling it along which dimension should these arrays be appended together. The default I believe is axis 0, which is rows so you probably don't have to do it in this case. Anyways that's what append does. It makes a new ndarray out of your component ndarrays.



	A	B	C	D	E	F	G	H	I	J	K
1	**Original Data**										
2	**row #**	**X2**	**X10**	**X11**	**Y**		**Tree**				
3	0	0.885	0.330	9.100	4.000		**node**	**Factor**	**SplitVal**	**Left**	**Right**
4	1	0.725	0.390	10.900	5.000		0				
5	2	0.560	0.500	9.400	6.000		1				
6	3	0.735	0.570	9.800	5.000		2				
7	4	0.610	0.630	8.400	3.000		3				
8	5	0.260	0.630	11.800	8.000		4				
9	6	0.500	0.680	10.500	7.000		5				
10	7	0.320	0.780	10.000	6.000		6				
11							7				
12							8				
13							9				
14							10				
15							11				
16							12				
17							13				
18							14				
19											

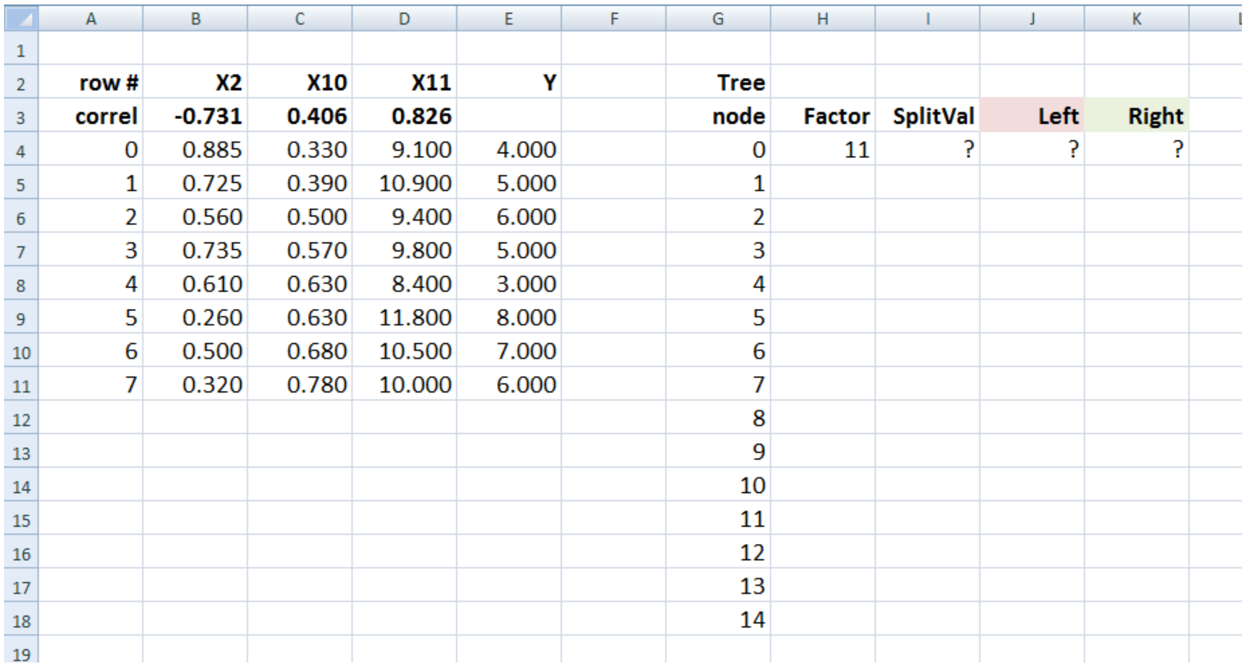
- Ok let me now give you a concrete example. Ok, this is also the example, by the way, from the graphical tree that I showed you. This is the same data in the same tree. And that data, by the way, I started with just eight samples of data because, you know, obviously I'm not gonna sit here and do a 1600 node tree. So I randomly selected eight rows out of the actual wine data and then I picked, arbitrarily, three columns or three factors to build this tree out of.

- But keep in mind for the real data you've got 11 factors. So the data that you're going to be building your trees with, you're gonna have potentially 11 factors.
- And anyways a factor 11, by the way, is alcohol content and I don't recall what factor 2 and 10 are. But anyhow, each row here represents one sample of, you know, a person scored a glass of wine. In this case they scored it 4 out of 10 and the quantitative measures for that glass of wine were these three values. You know, this represents another glass of wine and potentially a different person who tasted it and their scoring.
- Ok, so our first question here. Ok we're going to build a tree. And by the way over here I've got space set aside where we're going to fill in. This is going to be our ndarray that represents our tree.
- Our first question here is which factor should we split on. What's going to be our root node. And another way of putting that is what is the most information-providing factor. So one way to do that is to use correlation .



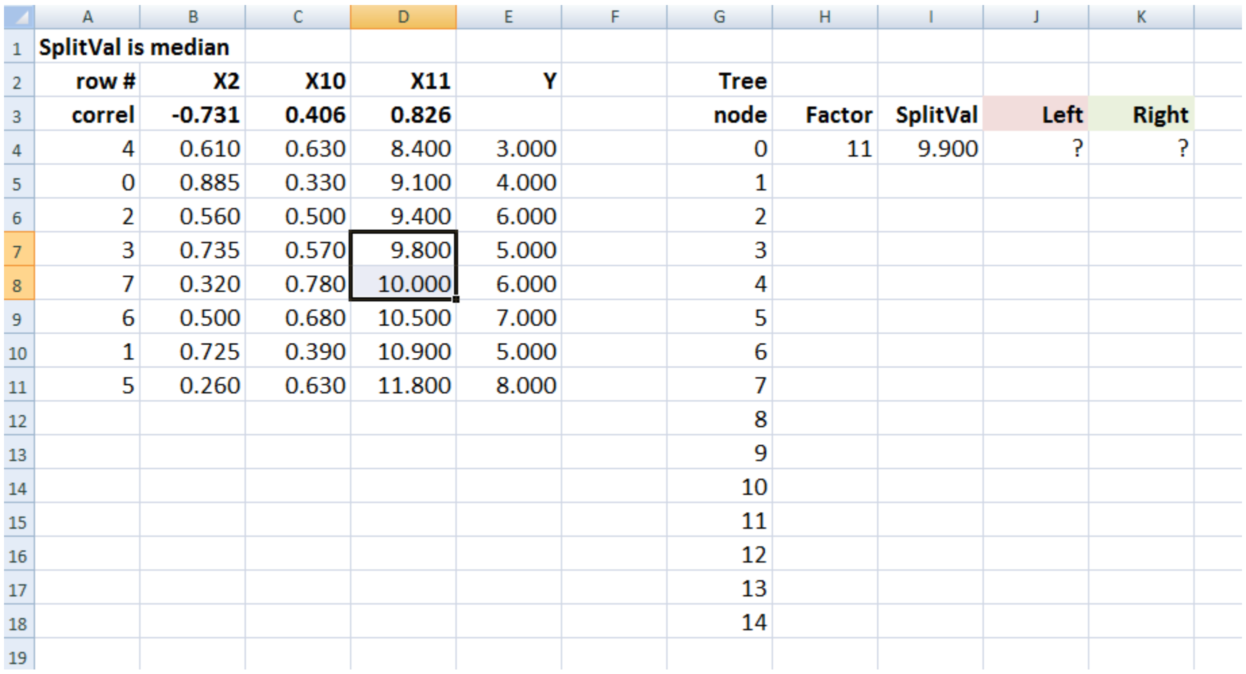
	A	B	C	D	E	F	G	H	I	J	K	L
1	**Take a look at correlation**											
2	**row #**	**X2**	**X10**	**X11**	**Y**		**Tree**					
3	**correl**	**-0.731**	**0.406**	**0.826**			**node**	**Factor**	**SplitVal**	**Left**	**Right**	
4	0	0.885	0.330	9.100	4.000		0					
5	1	0.725	0.390	10.900	5.000		1					
6	2	0.560	0.500	9.400	6.000		2					
7	3	0.735	0.570	9.800	5.000		3					
8	4	0.610	0.630	8.400	3.000		4					
9	5	0.260	0.630	11.800	8.000		5					
10	6	0.500	0.680	10.500	7.000		6					
11	7	0.320	0.780	10.000	6.000		7					
12							8					
13							9					
14							10					
15							11					
16							12					
17							13					
18							14					
19												

- Along the top here, each of these values represents the correlation of this column of data with Y, with our labels. And I just use the Excel correlation function here. So correl short for correlation. Correlation of this column with that column.
- Now you notice that this one is negative. It still is valuable information. All that means is, as this factor increases, Y decreases, and vice versa, but it still is providing us information about Y. Now if you look at these three, we see that X11, factor 11, has the highest correlation. So we're going to choose that to make our root node, to do that first split.
- By the way, when I share this spreadsheet with you, all these tabs along the bottom, you can think of sort of like different pages in a PowerPoint presentation.



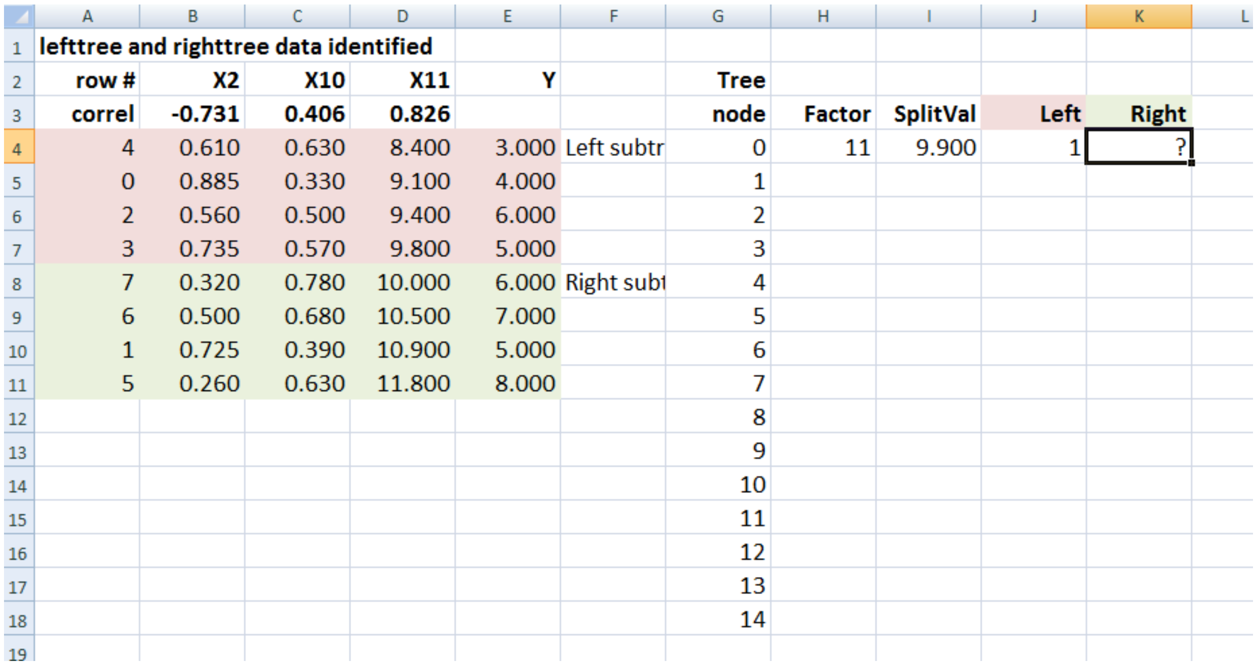
	A	B	C	D	E	F	G	H	I	J	K
1											
2	row #	X2	X10	X11	Y		Tree				
3	correl	-0.731	0.406	0.826			node	Factor	SplitVal	Left	Right
4	0	0.885	0.330	9.100	4.000		0	11	?	?	?
5	1	0.725	0.390	10.900	5.000		1				
6	2	0.560	0.500	9.400	6.000		2				
7	3	0.735	0.570	9.800	5.000		3				
8	4	0.610	0.630	8.400	3.000		4				
9	5	0.260	0.630	11.800	8.000		5				
10	6	0.500	0.680	10.500	7.000		6				
11	7	0.320	0.780	10.000	6.000		7				
12							8				
13							9				
14							10				
15							11				
16							12				
17							13				
18							14				
19											

- Ok so we started to fill in our data structure here representing our tree. So we're going to use factor 11. Next question is what should the split val be? Remember we talked about using median. So the easiest way to compute the median is, first we sort our data.



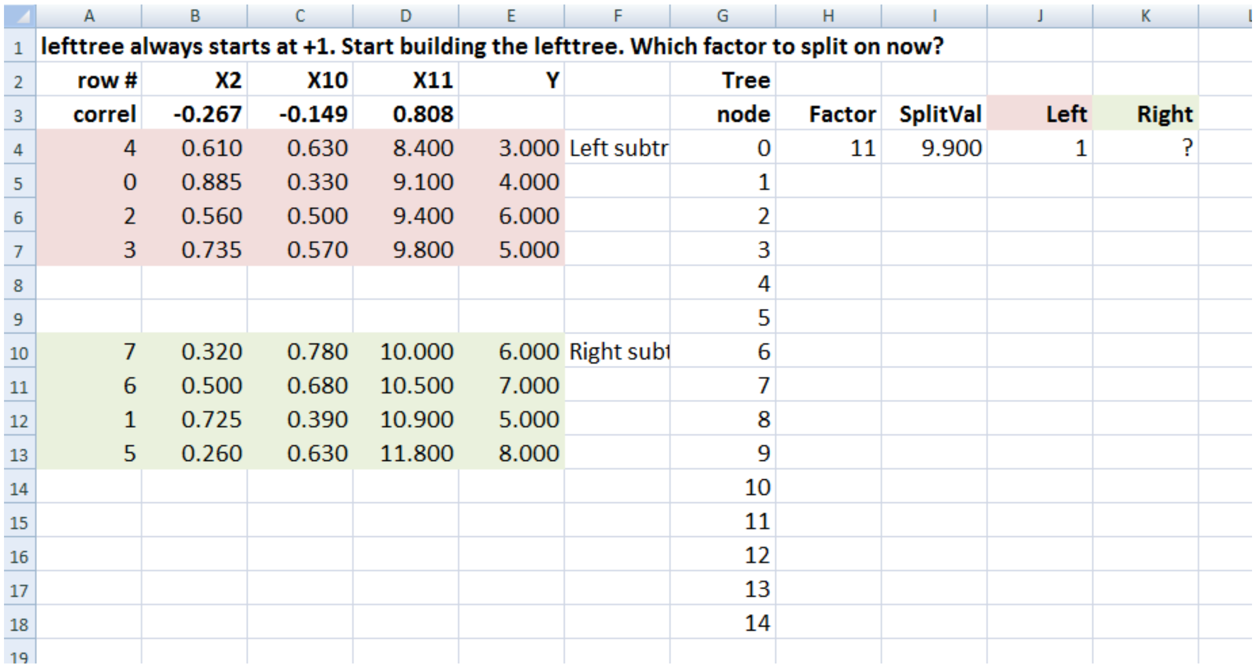
	A	B	C	D	E	F	G	H	I	J	K
1	SplitVal is median										
2	row #	X2	X10	X11	Y		Tree				
3	correl	-0.731	0.406	0.826			node	Factor	SplitVal	Left	Right
4	4	0.610	0.630	8.400	3.000		0	11	9.900	?	?
5	0	0.885	0.330	9.100	4.000		1				
6	2	0.560	0.500	9.400	6.000		2				
7	3	0.735	0.570	9.800	5.000		3				
8	7	0.320	0.780	10.000	6.000		4				
9	6	0.500	0.680	10.500	7.000		5				
10	1	0.725	0.390	10.900	5.000		6				
11	5	0.260	0.630	11.800	8.000		7				
12							8				
13							9				
14							10				
15							11				
16							12				
17							13				
18							14				
19											

- So I've sorted now all of these rows according to this column. It turns out that you don't explicitly have to do this in your program because you can do that comprehension method I mentioned a moment ago. But anyways, if you sort, it becomes clearly evident what the median value is. That you just want the particular value that will split the data in half. So, you know, we go from 9.8 to 10.0. So our split value is a 9.9.



	A	B	C	D	E	F	G	H	I	J	K	L
1	**lefttree and righttree data identified**											
2	**row #**	**X2**	**X10**	**X11**	**Y**		**Tree**					
3	**correl**	**-0.731**	**0.406**	**0.826**			**node**	**Factor**	**SplitVal**	**Left**	**Right**	
4	4	0.610	0.630	8.400	3.000	Left subtr	0	11	9.900	1	?	
5	0	0.885	0.330	9.100	4.000		1					
6	2	0.560	0.500	9.400	6.000		2					
7	3	0.735	0.570	9.800	5.000		3					
8	7	0.320	0.780	10.000	6.000	Right subt	4					
9	6	0.500	0.680	10.500	7.000		5					
10	1	0.725	0.390	10.900	5.000		6					
11	5	0.260	0.630	11.800	8.000		7					
12							8					
13							9					
14							10					
15							11					
16							12					
17							13					
18							14					
19												

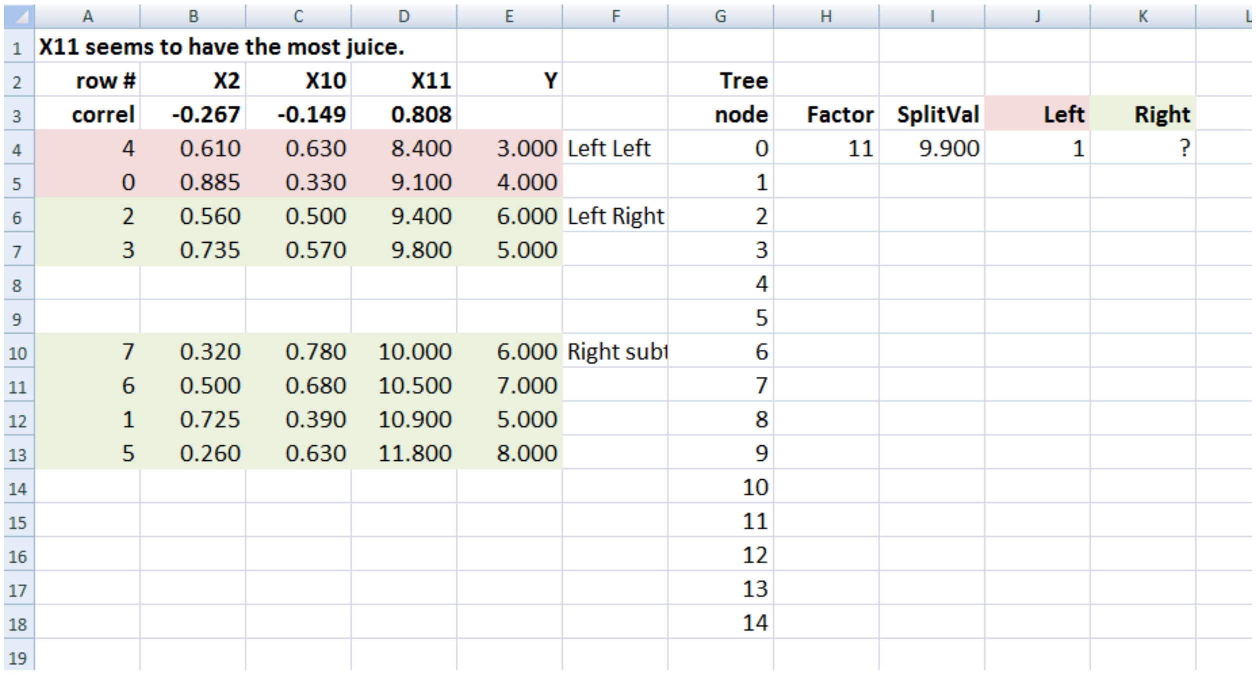
- Alright so I'm indicating now the data that's going to make up our left subtree here in red and the right subtree there in green. Like I said, we already know where our left subtree is going to start. It's from the very next row. So it's going to fill up this part of our data structure.



	A	B	C	D	E	F	G	H	I	J	K	L
1	**lefttree always starts at +1. Start building the lefttree. Which factor to split on now?**											
2	**row #**	**X2**	**X10**	**X11**	**Y**		**Tree**					
3	**correl**	**-0.267**	**-0.149**	**0.808**			**node**	**Factor**	**SplitVal**	**Left**	**Right**	
4	4	0.610	0.630	8.400	3.000	Left subtr	0	11	9.900	1	?	
5	0	0.885	0.330	9.100	4.000		1					
6	2	0.560	0.500	9.400	6.000		2					
7	3	0.735	0.570	9.800	5.000		3					
8							4					
9							5					
10	7	0.320	0.780	10.000	6.000	Right subt	6					
11	6	0.500	0.680	10.500	7.000		7					
12	1	0.725	0.390	10.900	5.000		8					
13	5	0.260	0.630	11.800	8.000		9					
14							10					
15							11					
16							12					
17							13					
18							14					
19												

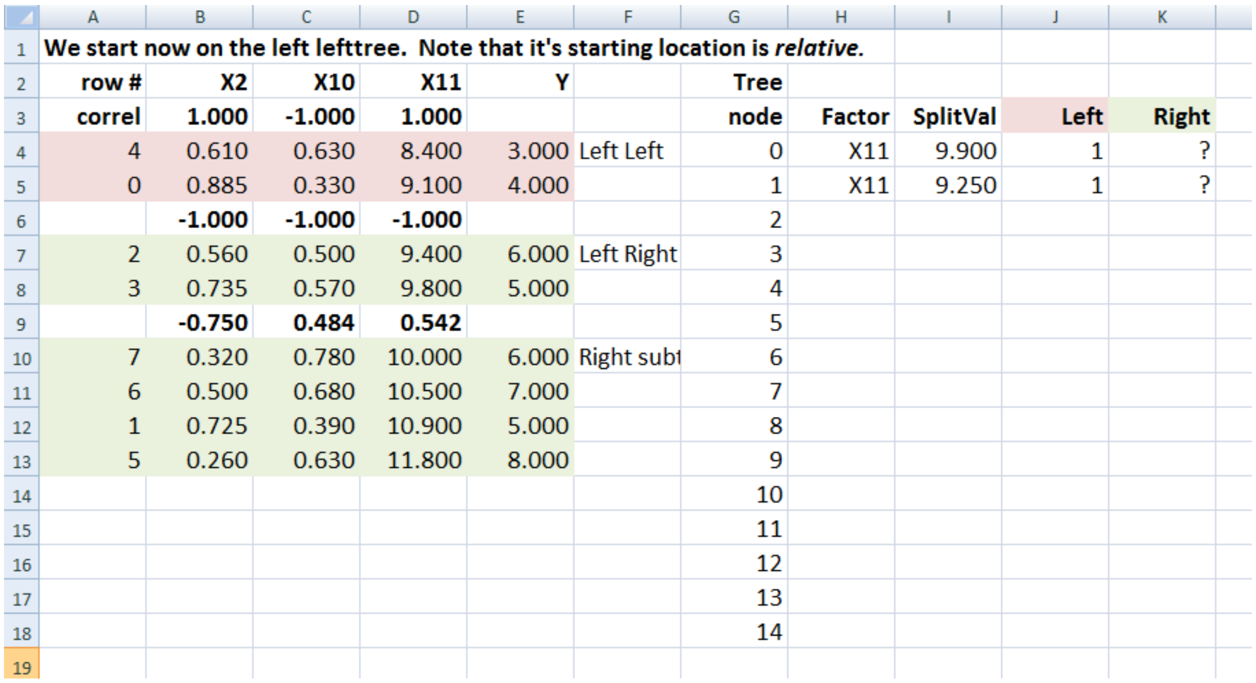
- I'm going to move the right subtree down a little bit so we have some room to think up here. Now we're calculating the correlation with this subtree. So for instance, this now represents the correlation of the data in our left subtree with Y and so on. So we're not looking at the whole dataset anymore, we're just looking at the left subtree.

- So we've recursed down into computing the left subtree. Again, interestingly enough, this same factor X11 has the strongest correlation. So we're going to use that again to split on. And it turns out it's already sorted according to that factor so it's going to be between these two values. So it'll be 9.25 will be our split val because that's the median here.



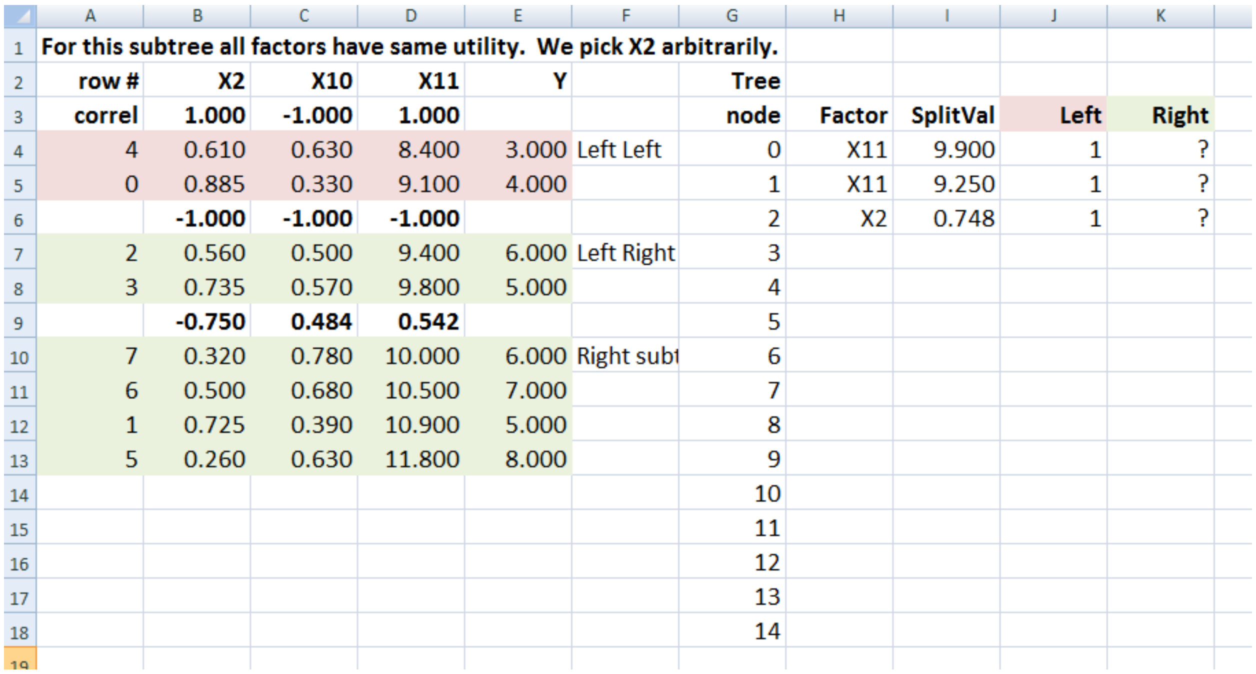
	A	B	C	D	E	F	G	H	I	J	K
1	**X11 seems to have the most juice.**										
2	**row #**	**X2**	**X10**	**X11**	**Y**		**Tree**				
3	**correl**	**-0.267**	**-0.149**	**0.808**			**node**	**Factor**	**SplitVal**	**Left**	**Right**
4	4	0.610	0.630	8.400	3.000	Left Left	0	11	9.900	1	?
5	0	0.885	0.330	9.100	4.000		1				
6	2	0.560	0.500	9.400	6.000	Left Right	2				
7	3	0.735	0.570	9.800	5.000		3				
8							4				
9							5				
10	7	0.320	0.780	10.000	6.000	Right subt	6				
11	6	0.500	0.680	10.500	7.000		7				
12	1	0.725	0.390	10.900	5.000		8				
13	5	0.260	0.630	11.800	8.000		9				
14							10				
15							11				
16							12				
17							13				
18							14				
19											

- So now we've split the data in two now. We're going to compute a left, left subtree and a left, right subtree and again we need to calculate in each case what is the correlation of each column with our labels Y.



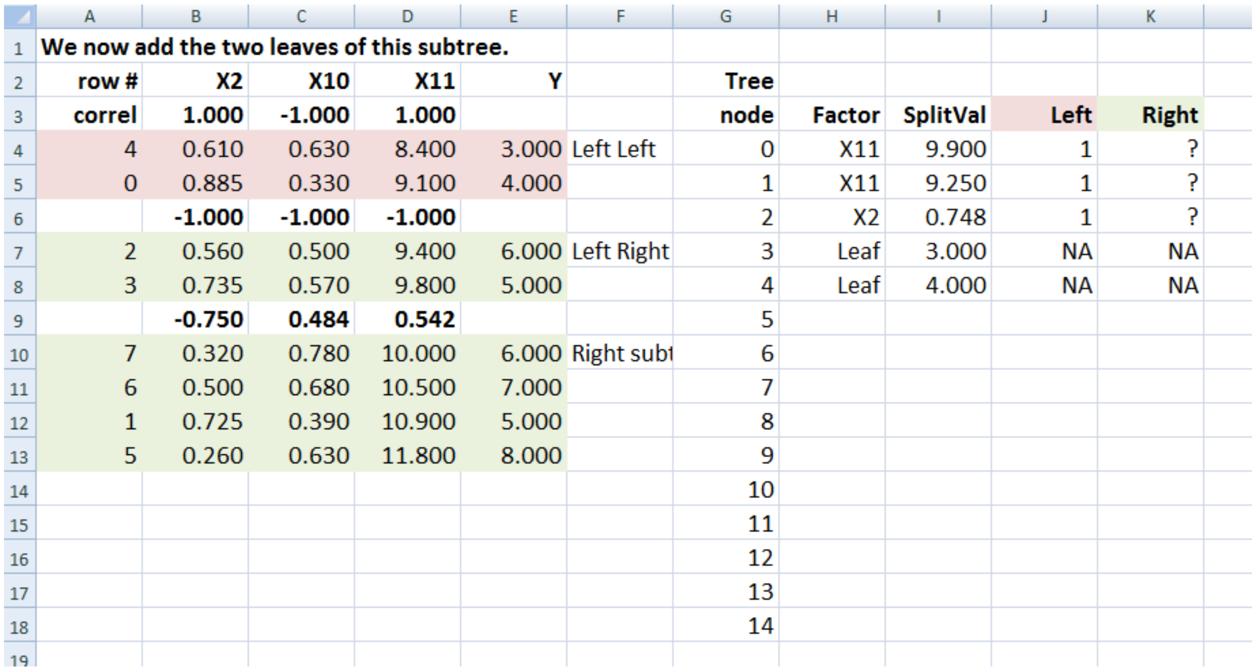
	A	B	C	D	E	F	G	H	I	J	K
1	**We start now on the left lefttree. Note that it's starting location is *relative*.**										
2	**row #**	**X2**	**X10**	**X11**	**Y**		**Tree**				
3	**correl**	**1.000**	**-1.000**	**1.000**			**node**	**Factor**	**SplitVal**	**Left**	**Right**
4	4	0.610	0.630	8.400	3.000	Left Left	0	X11	9.900	1	?
5	0	0.885	0.330	9.100	4.000		1	X11	9.250	1	?
6		**-1.000**	**-1.000**	**-1.000**			2				
7	2	0.560	0.500	9.400	6.000	Left Right	3				
8	3	0.735	0.570	9.800	5.000		4				
9		**-0.750**	**0.484**	**0.542**			5				
10	7	0.320	0.780	10.000	6.000	Right subt	6				
11	6	0.500	0.680	10.500	7.000		7				
12	1	0.725	0.390	10.900	5.000		8				
13	5	0.260	0.630	11.800	8.000		9				
14							10				
15							11				
16							12				
17							13				
18							14				
19											

- By the way we've added another row here which represents the decision node. We're going to split the data on factor 11 again we're going to use 9.25.
- Now something to observe here is when you get down to the point where you've only got two data elements, it's very likely that you're going to see correlations of 1 and negative one for each factor and there's a question of, ok, in general if they have the same value, how to decide which one to split on?
- There's multiple correct answers there. One is select randomly, another is deterministically choose the first one, or deterministically choose the last one. Whatever. I recommend that you go with a deterministic approach because we're going to have plenty of randomness in this data and in these trees anyways. It's not really necessary to arbitrarily add randomness here. And you'll see in a moment some ways that we inject randomness into this.
- Ok so we've selected our root node. We're now building the left subtree using that root node. So this now represents the overall left subtree. We're now going to look at building the left subtree of the left subtree and I'm going to arbitrarily choose factor two here as our split val.



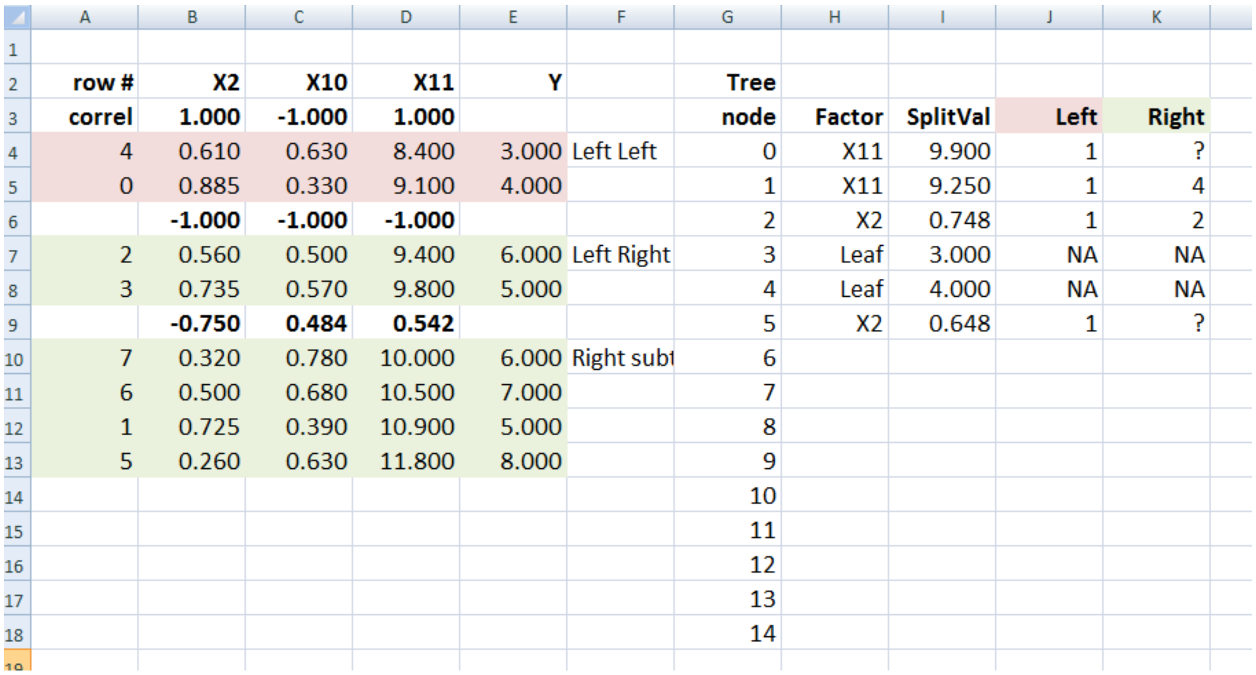
	A	B	C	D	E	F	G	H	I	J	K
1	**For this subtree all factors have same utility. We pick X2 arbitrarily.**										
2	**row #**	**X2**	**X10**	**X11**	**Y**		**Tree**				
3	**correl**	**1.000**	**-1.000**	**1.000**			**node**	**Factor**	**SplitVal**	**Left**	**Right**
4	4	0.610	0.630	8.400	3.000	Left Left	0	X11	9.900	1	?
5	0	0.885	0.330	9.100	4.000		1	X11	9.250	1	?
6		**-1.000**	**-1.000**	**-1.000**			2	X2	0.748	1	?
7	2	0.560	0.500	9.400	6.000	Left Right	3				
8	3	0.735	0.570	9.800	5.000		4				
9		**-0.750**	**0.484**	**0.542**			5				
10	7	0.320	0.780	10.000	6.000	Right subt	6				
11	6	0.500	0.680	10.500	7.000		7				
12	1	0.725	0.390	10.900	5.000		8				
13	5	0.260	0.630	11.800	8.000		9				
14							10				
15							11				
16							12				
17							13				
18							14				

- And the median here turns out to be 7.48. And now we have all the information we need to add our two leaf nodes.



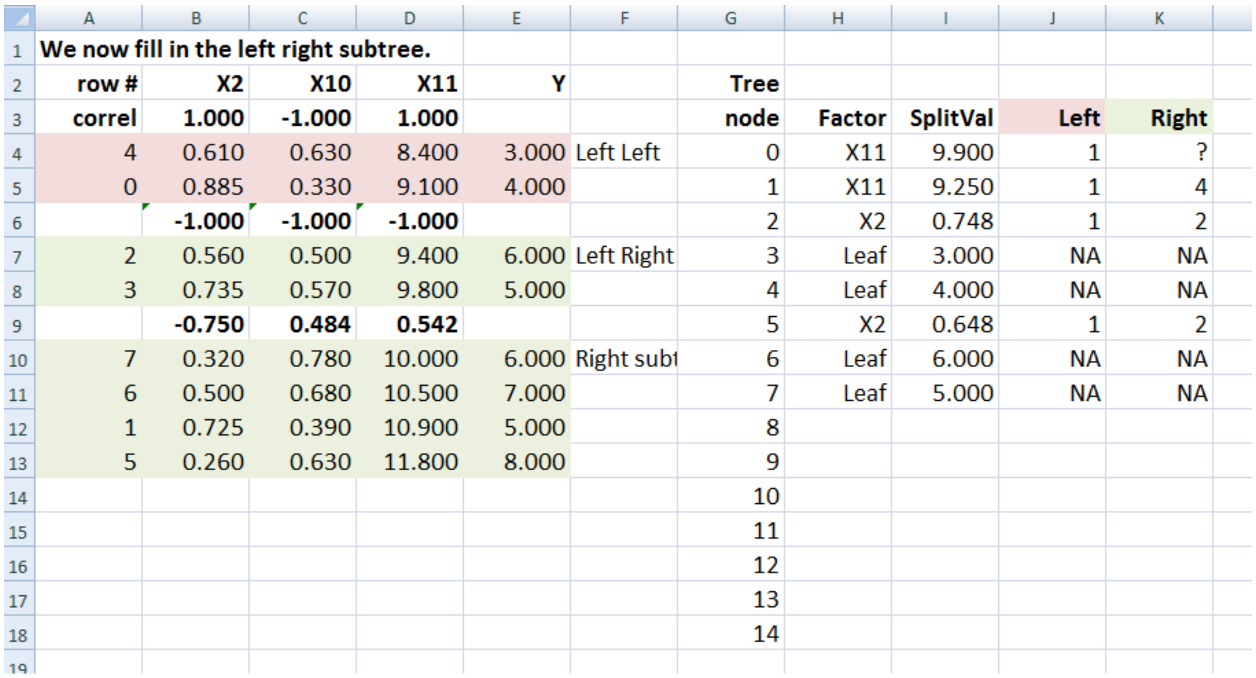
	A	B	C	D	E	F	G	H	I	J	K
1	**We now add the two leaves of this subtree.**										
2	**row #**	**X2**	**X10**	**X11**	**Y**		**Tree**				
3	**correl**	**1.000**	**-1.000**	**1.000**			**node**	**Factor**	**SplitVal**	**Left**	**Right**
4	4	0.610	0.630	8.400	3.000	Left Left	0	X11	9.900	1	?
5	0	0.885	0.330	9.100	4.000		1	X11	9.250	1	?
6		**-1.000**	**-1.000**	**-1.000**			2	X2	0.748	1	?
7	2	0.560	0.500	9.400	6.000	Left Right	3	Leaf	3.000	NA	NA
8	3	0.735	0.570	9.800	5.000		4	Leaf	4.000	NA	NA
9		**-0.750**	**0.484**	**0.542**			5				
10	7	0.320	0.780	10.000	6.000	Right subt	6				
11	6	0.500	0.680	10.500	7.000		7				
12	1	0.725	0.390	10.900	5.000		8				
13	5	0.260	0.630	11.800	8.000		9				
14							10				
15							11				
16							12				
17							13				
18							14				

- So if X2 is less than or equal to 7.48, well that's this leaf and we know the value is three. Boom we fill that in. Otherwise we're going to the right and it's this leaf. Value is four.
- We similarly pop back up now and do the same thing with the right tree. I should say the left-right tree and now we've completed the whole left subtree.



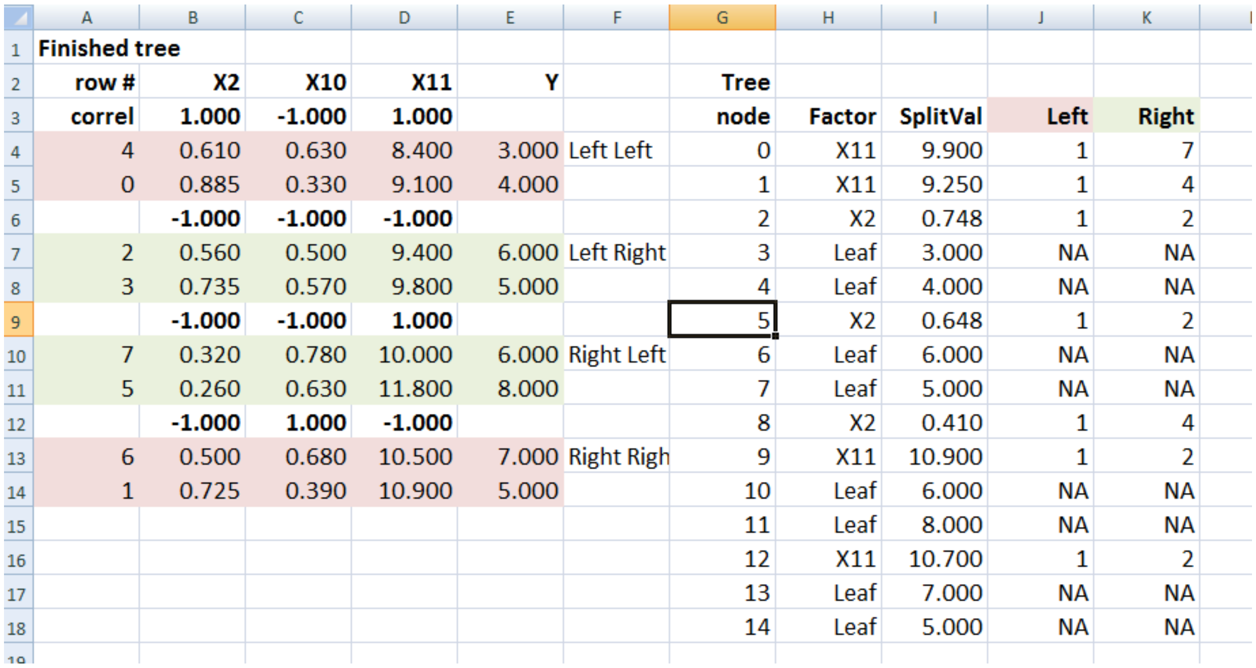
	A	B	C	D	E	F	G	H	I	J	K
1											
2	**row #**	**X2**	**X10**	**X11**	**Y**		**Tree**				
3	**correl**	**1.000**	**-1.000**	**1.000**			**node**	**Factor**	**SplitVal**	**Left**	**Right**
4	4	0.610	0.630	8.400	3.000	Left Left	0	X11	9.900	1	?
5	0	0.885	0.330	9.100	4.000		1	X11	9.250	1	4
6		**-1.000**	**-1.000**	**-1.000**			2	X2	0.748	1	2
7	2	0.560	0.500	9.400	6.000	Left Right	3	Leaf	3.000	NA	NA
8	3	0.735	0.570	9.800	5.000		4	Leaf	4.000	NA	NA
9		**-0.750**	**0.484**	**0.542**			5	X2	0.648	1	?
10	7	0.320	0.780	10.000	6.000	Right subt	6				
11	6	0.500	0.680	10.500	7.000		7				
12	1	0.725	0.390	10.900	5.000		8				
13	5	0.260	0.630	11.800	8.000		9				
14							10				
15							11				
16							12				
17							13				
18							14				

- One thing I want to point out here, the numbers I'm filling in here to point down to essentially the rows at which the next sub trees begin are relative. So this four means for instance that the right subtree begins four rows down so 1, 2, 3, 4. It's starting here and this two for instance means the right leaf is two rows below, 1, 2. So these values are always relative
- The reason for that is it makes bookkeeping a little bit easier when you're traversing the tree, you know, to do a query. It also makes it a little bit easier when you're building the tree.
- You could still make them absolute references, you know, it's up to you. But you have to pass around some more information to keep track of where you are in the tree. Doing it this way you don't ever really have to know where you are in the tree as you're executing the algorithm.



	A	B	C	D	E	F	G	H	I	J	K
1	We now fill in the left right subtree.										
2	row #	X2	X10	X11	Y		Tree				
3	correl	1.000	-1.000	1.000			node	Factor	SplitVal	Left	Right
4	4	0.610	0.630	8.400	3.000	Left Left	0	X11	9.900	1	?
5	0	0.885	0.330	9.100	4.000		1	X11	9.250	1	4
6		-1.000	-1.000	-1.000			2	X2	0.748	1	2
7	2	0.560	0.500	9.400	6.000	Left Right	3	Leaf	3.000	NA	NA
8	3	0.735	0.570	9.800	5.000		4	Leaf	4.000	NA	NA
9		-0.750	0.484	0.542			5	X2	0.648	1	2
10	7	0.320	0.780	10.000	6.000	Right subt	6	Leaf	6.000	NA	NA
11	6	0.500	0.680	10.500	7.000		7	Leaf	5.000	NA	NA
12	1	0.725	0.390	10.900	5.000		8				
13	5	0.260	0.630	11.800	8.000		9				
14							10				
15							11				
16							12				
17							13				
18							14				

- Okay so now we've fleshed out the complete left-hand subtree. Then we start working on the right subtree and we just skip to the end it is a long and boring story to go through the whole thing.



	A	B	C	D	E	F	G	H	I	J	K
1	Finished tree										
2	row #	X2	X10	X11	Y		Tree				
3	correl	1.000	-1.000	1.000			node	Factor	SplitVal	Left	Right
4	4	0.610	0.630	8.400	3.000	Left Left	0	X11	9.900	1	7
5	0	0.885	0.330	9.100	4.000		1	X11	9.250	1	4
6		-1.000	-1.000	-1.000			2	X2	0.748	1	2
7	2	0.560	0.500	9.400	6.000	Left Right	3	Leaf	3.000	NA	NA
8	3	0.735	0.570	9.800	5.000		4	Leaf	4.000	NA	NA
9		-1.000	-1.000	1.000			5	X2	0.648	1	2
10	7	0.320	0.780	10.000	6.000	Right Left	6	Leaf	6.000	NA	NA
11	5	0.260	0.630	11.800	8.000		7	Leaf	5.000	NA	NA
12		-1.000	1.000	-1.000			8	X2	0.410	1	4
13	6	0.500	0.680	10.500	7.000	Right Righ	9	X11	10.900	1	2
14	1	0.725	0.390	10.900	5.000		10	Leaf	6.000	NA	NA
15							11	Leaf	8.000	NA	NA
16							12	X11	10.700	1	2
17							13	Leaf	7.000	NA	NA
18							14	Leaf	5.000	NA	NA

- But here now, finally, is our entire data structure describing the tree. So that's how you build a decision tree from sample data. It's one way. I'm going to give you another way in just a moment.
- But let me now go check and see what questions we've got. Ok, no new questions.

Decision Tree Algorithm (JR Quinlan)

```
build_tree(data)
    if data.shape[0] == 1: return [leaf, data.y, NA, NA]
    if all data.y same: return [leaf, data.y, NA, NA]
    else
        determine best feature i to split on
        SplitVal = data[:,i].median()
        lefttree = build_tree(data[data[:,i]<=SplitVal])
        righttree = build_tree(data[data[:,i]>SplitVal])
        root = [i, SplitVal, 1, lefttree.shape[0] + 1]
        return (append(root, lefttree, righttree))
```

- Let me now show you a different way to build a tree. Before I show you that. Let me ask you to think about something for a while. We'll take a look at this original definition of how to build a tree. Which of these steps in this algorithm do you think are expensive? You know, obviously, some are more expensive than others, but in particular, which one is probably the most expensive? I'll be quiet for a moment while you ponder that. And I'm having a sip of wine myself. Makes the lecture better.
- Okay, certainly one of the most expensive things to do is determining this best feature to split on. Why? Well in the method I presented, if we're using correlation, calculating the correlation you know, for say 1600 rows...Numpy can do it quickly, I admit it, but it still of all the things we might do, is one of the more expensive and we have to do the correlation for each column of data before we can decide what to split on.
- So if there's one way that we could, you know, one thing we can do to speed up our algorithm the most, it would probably be to make that part faster.
- Another one is computing the median. Now in the example I gave, I showed that you should first sort the data and then find the middle values. And that will help you compute the median. It turns out there's faster algorithms now that actually can compute the median in order N time. You don't actually have to do the full sort. So the median is order N but it's a big N, so it can take a while
- But anyways, long story short, if we can figure out a way to do these two steps faster, that would overall help our algorithm be a lot faster, and that's exactly where random trees come in.

Random Tree Algorithm (A Cutler)

```
build_tree(data)
    if data.shape[0] == 1: return [leaf, data.y, NA, NA]
    if all data.y same: return [leaf, data.y, NA, NA]
    else
        determine random feature i to split on
        SplitVal = (data[random,i] + data[random,i]) / 2
        lefttree = build_tree(data[data[:,i]<=SplitVal])
        righttree = build_tree(data[data[:,i]>SplitVal])
        root = [i, SplitVal, 1, lefttree.shape[0] + 1]
        return (append(root, lefttree, righttree))
```

- So we skip to random trees. This is an algorithm by Adele Cutler. She was a PhD student of Brieman who was at UC Berkeley. And he took JR Quinlan's ideas of, you know, creating decision trees, but expanded the idea greatly with random trees and random forests.
- And Ms. Cutler who is now a professor in Utah produced I think one of the best papers describing their methodology. And she's carried it forward in a number of different ways.
- But anyways, the general idea is this. Rather than burn all that time trying to decide what the best feature is to split on, let's choose it randomly. Clearly, picking a random number, you know picking a random number from 1 to 10 is faster than doing the correlation on 10 columns of 1600 pieces of data. So that speeds things up significantly.
- Now you might naturally asked, um yeah it probably does make it faster, but doesn't that screw up your tree? So postpone that question for a second, and I'll answer it in a moment.
- The initial answer is yes, it does impair the quality of the decision tree but there's something we can do to work around that and I'll get to that in a moment.
- But the other thing she does is rather than compute the median and you know having to sort or run the select algorithm, you just choose a random row, two random rows, grab the feature value out of those rows and take their mean. I just divide them by 2 and that's guaranteed to give you a split value that is somewhere not at the edges of the data, is somewhere in the middle. It might be towards an edge but it's definitely not going to be at a very edge.

- So these two random selections here make the tree building algorithm monstrously faster. You still then have to go through the rest of the algorithm like I described before, but these two random selections make it much faster and much easier.
- Okay now back to this earlier question of well doesn't that impair the quality of my tree? The answer is yes, if you have only one tree. But something we're going to get to shortly either later this week or early next week is something called bagging and in general the idea here is it's better to have a learner that is composed of many learners, another term for that is ensemble learners, than just have a single learner.
- So if you have a single tree and you build it carefully using this Quinlan method, yes it can do very good. But if you have a learner that's built of multiple trees even if they're created randomly, its performance can actually supersede that of the best individual tree.
- Again we'll get into more detail on that little bit later but essentially a random forest is a group of trees where each tree is computed somewhat randomly.
- Now I want to mention something about this randomness. There's several different ways that you can cause this forest to be random. One of them is each tree is computed with these random numbers.
- Another is that the data that you construct the tree from could be selected randomly. So you're given a whole set of data from which to create a learner. What you can do is subsample, you know, randomly select not all the data but some subsample of that data, create a tree, resample again, it's with replacement. You pick another group of data, build another tree, and for each tree you build, you grab some random assortment of that data.
- So even if you're building say the Quinlan type tree that isn't random at all, each tree that you build is built from a different selection of the data so it's going to be a bit different. And it turns out that these differences are important. If all of your learners are identical, there's really no benefit to having multiple learners. You want essentially uncorrelated learners or diversity among your learners. So that's why these random approaches end up, when you combine them into an ensemble learner, give you a really valuable system.
- Okay, I'm gonna check one more time for questions and then I want to wrap up with the strengths and weaknesses of decision tree learners. Let me check for questions here real quick. Never mind. Glad to have you all participating by the way at this late evening time.

Strengths and weaknesses of decision tree learners

- Cost of learning:
  - Most: Decision Trees
  - Medium: Linear Regression
  - Least: KNN
- Cost of query:
  - Most: KNN
  - Medium: Decision Trees
  - Least: Linear Regression
- Trees: Don’t have to normalize your data and can easily handle missing data

- Ok, strengths and weaknesses of decision tree learners. So you should look at a few things. Let me actually build this list while I'm talking about it.
- The cost of learning. If you compare, for instance, what's the expense of creating a decision tree compared to say to creating a KNN learner. So we went through this fairly complex algorithm about how to build a decision tree, and as you can see, it can be expensive. KNN, k-nearest neighbors is simple at learning time. You just take the data and plop it into ram and consult it later when you're doing queries. So in terms of cost of learning, decision trees lose to k-nearest neighbor.
- I would say also that linear regression learners, they're somewhere in between. To build a parametric model using linear regression takes a little bit longer than it does to just save things to memory like KNN, but doesn't take quite as long as it does to build a decision tree. Andif you're building a decision forest, you know you gotta multiply that by however many trees you're going to have in your forest.
- Next is cost of query. So among these three algorithms, K nearest neighbor, linear regression, and decision trees, I want you to think for a moment. Which is the fastest, KNN, linear regression, or decision tree? We know in other words we've got our X factors and we want to say "Hey what's the predicted value? What's the predicted Y using this data?"
- I'll pause for secondly, let you consider, then give you the answer. Linear regression is the fastest. All you've got to do if you have a regression model, it's just a set of parameters, you just, you know, X1 times parameter 1, X2 times parameter 2. Add them all together. Boom that's the answer. It's blazingly fast.

- So in many cases linear regression learners don't have such a high quality in terms of you know what's the prediction versus the actual result, but they are blazingly fast to learn and blazingly fast query. KNN is the worst of them all because when you're querying a KNN learner, you have to compute the distance from your query to all of your individual data points, then sort them and find the closest k data points. It's extremely expensive to query a k-nearest neighbor.
- Decision trees, again, are somewhere in between. The beauty of them is because they are binary trees, for say a thousand elements, on average you only have to ask I think whatever log base 2 of a thousand is. I think it's about 10.
- Anyways so if you build a tree with a thousand samples, to query it, you would have to ask at most 10 binary questions and its really fast. So again decision trees are somewhere there in between.
- Now there are some strong benefits of decision trees that I want to mention here. One is you don't have to normalize your data. So I believe in one of the online lectures, we're talking about this problem of suppose one of your factors ranges from zero to a thousand and another factor ranges from 0 to 1. It turns out that if you build a K nearest neighbor type model from that data, the factor that ranges from zero to a thousand ends up overwhelming the other factors and turns out to just be the most important factor even though it may not really be the most important factor.
- So typically with KNN learners you have to do something to normalize your data so that each dimension essentially has the same range. So typically you take the mean of all the data for a factor one and normalize it to that mean and standard deviation.
- You don't have to do that for decision trees. They automatically determine what appropriate thresholds are as they build the tree.
- Just trying to think of the other benefits of decision trees. I'm sure the ones I mentioned the other day will occur to me as soon as we finish the lecture, but I'll add them to the presentation later on.
- But anyways, that concludes my lecture to you on how to build a tree.
- I'll do one more check on Piazza to see if we got any more questions and then I'll wrap it up.
- Pat adds a link to scikit-learn and pandas. I believe he's pointing to something that lets you visualize a tree once you've created it. That's nice. Ah and thank you. Pat has provided log base 2 of several numbers for me. Thanks.
- Ok, getting to this last question. The reason we don't have to normalize the data is each variable is treated the same weight and it's based on correlation or entropy to decide the best split? Yeah, good point. We're essentially looking at correlation to decide which factor to use and then, yeah, we use the median to decide the split value, so there's no additional benefit to normalizing it first. If, in the case of KNN, if you didn't normalize it, like I said, that particular factor that varied over a wide range will become the most important one.
- Ok doing one more check for questions and then I'm going to wrap up. Ok thank you everybody for your attention. I am going to go ahead and close up the presentation. And yes I will post these slides to the wiki. This is kind of fun. Anyways, let me stop that and I'll stop the broadcast. Bye-bye. Have a great evening and see you in TV land.