Decision Trees and Bagging Models [Random Forest , ExtraTrees]

Can we use linear models for non-linear patterns in the data

Linear models are called linear models because we use linear combination of features for our model $f(X_i)$. However contrary to common misconception arising from the name is that linear models are strictly limited to modeling linear patterns. What do we mean by linear patterns ?

Target is said to be linearly related to features when it can be very well approximated in linear proportion to the features , thats what the linear models that we discussed were doing. However what happens if target Y is not linearly proportion to the features .

What if these are the proportionality relations that your target in some data has [say the data has 3 features X_1, X_2, X_3]

$$Y \propto X_1{}^2$$
$$Y \propto log(X_2)$$
$$Y \propto e^{X_3}$$

If this happens to be the case then of course an approximation like $Y \propto w_0 + w_1 X_1 + w_2 X_2 + w_3 X_3$, doesn't work well naturally. Does that mean we can not use the linear model framework that we just studied. You very well can, just create a transformed version of these features , say

$$X_1{}' = X_1{}^2$$
$$X_2{}' = log(X_2)$$
$$X_3{}' = e^{X_3}$$

and then you can use same framework $w_0 + w_1 X_1{}' + w_2 X_2{}' + w_3 X_3{}'$ as the approximation for your target. However in real life things are not that easy. It will rarely happen that you will be able to discover these clean analytical proportionality relationships . Here is an example :
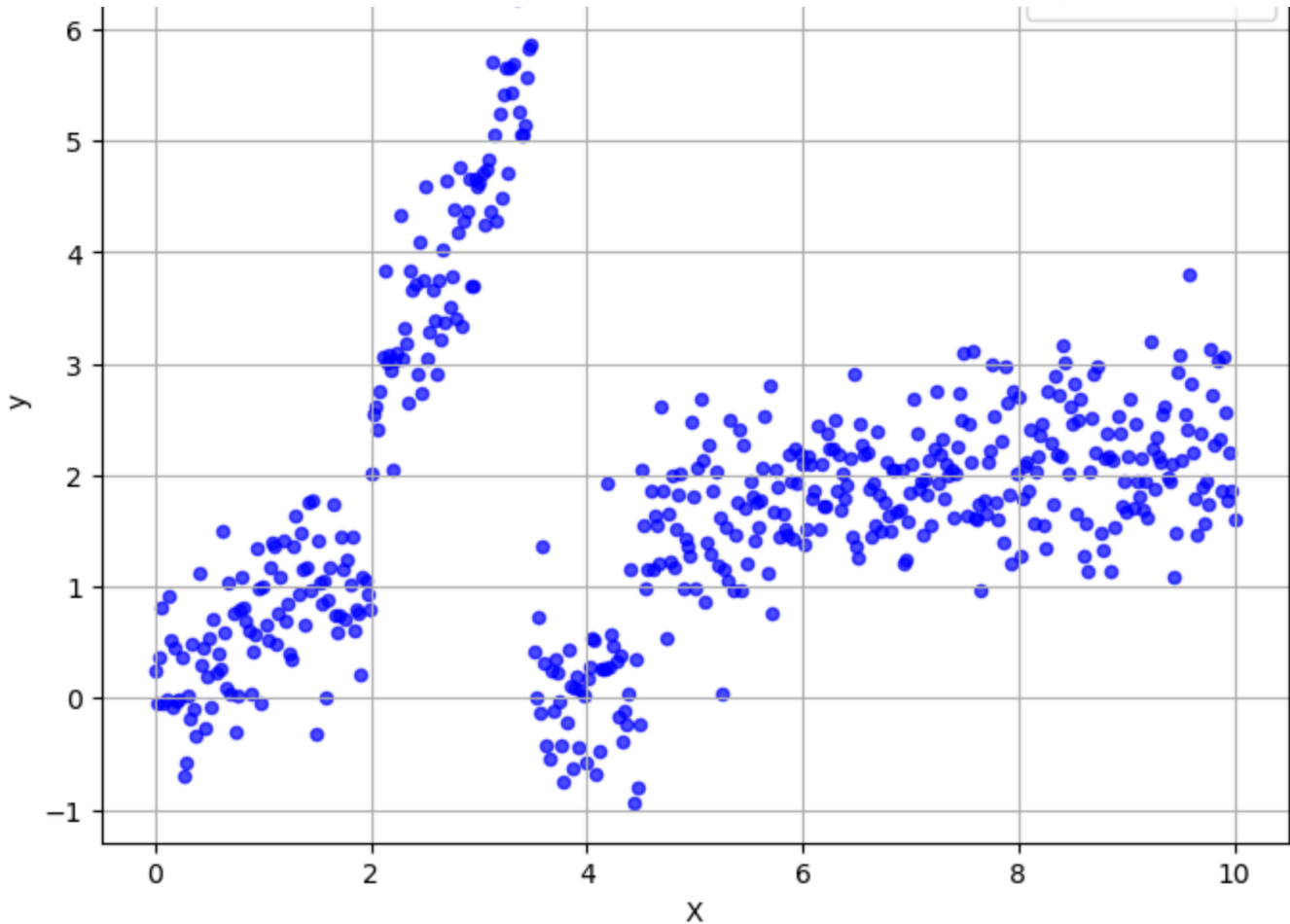
There isn't really a simple analytical function which will account for the proportionality for this one, so in theory; yes you could still potentially use linear models for non-linear relationships but it does not work in practice . We need an algorithm which can model complex non-linear patterns without needing a pre-informed relationship functional to back it up. Decision Trees are one such algorithm which works as the foundation of most of modern ML algorithms for tabular data.

Fundamental Idea behind Decision Trees

The idea arises from thinking about source of non-linearity in the data. Essentially different subsets of the data have different relationships with respect to how target changes with features. This can be modeled by partitioning the data itself into subsets on the basis of feature values and then having different predictions for each subset independently instead of looking for a global relationship . Decision Trees implement this idea by partitioning the data into smaller buckets recursively and using simple average/proportions of the target in each bucket as prediction. This partitioning is driven by some familiar cost functions, some versions of which we have already seen before. Here is a visual look at the idea for a classification problem with two features and and two classes in the target. Notice how the feature space is recursively partitioned with rules based feature values to separate the observations of different classes and eventually with this process we end-up capturing a complex non-linear decision boundary between the two classes.

| iteration 1 | iteration 2 |
| iteration 3 | iteration 4 |
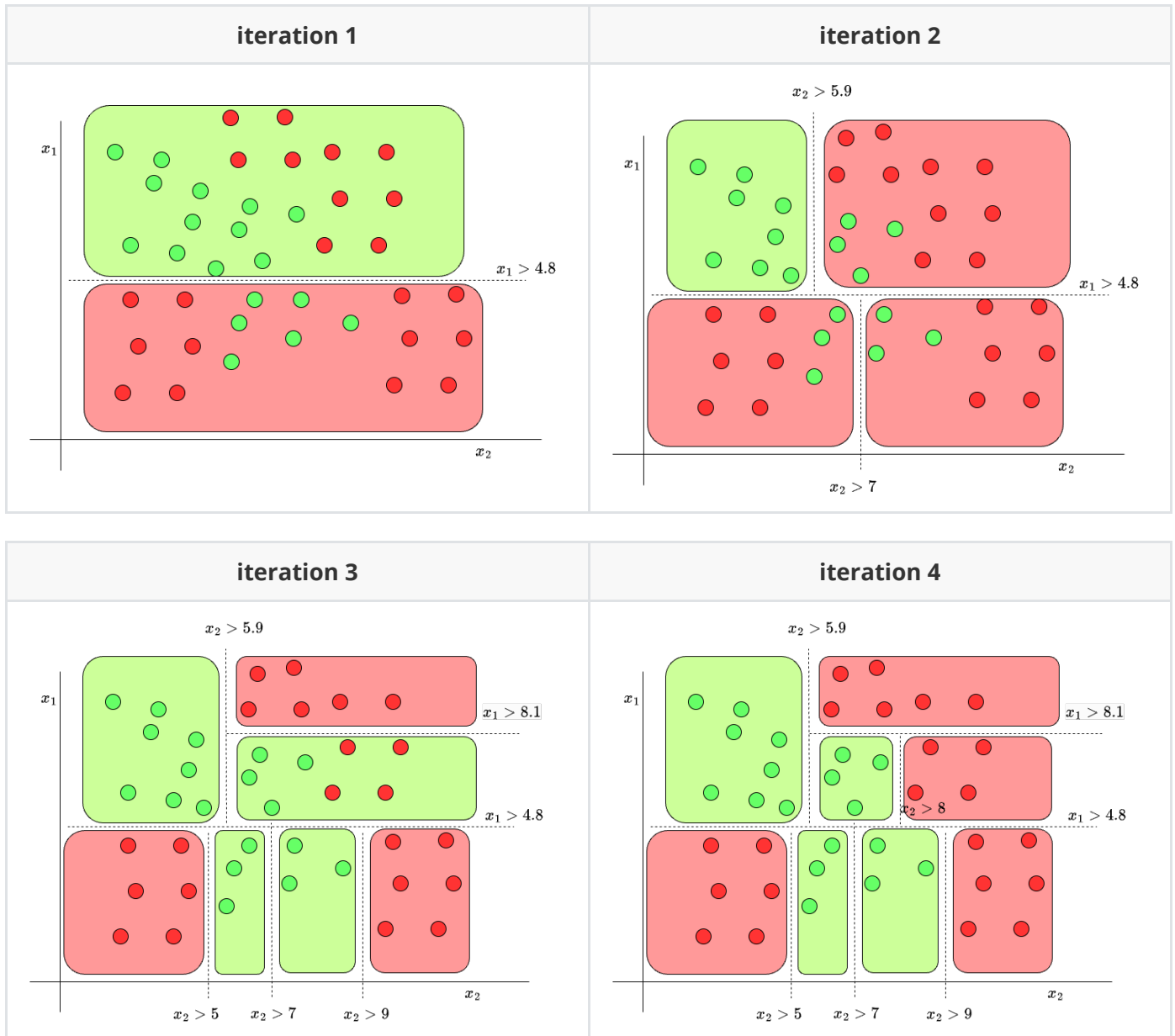
Here is the figure with effective non-linear decision boundary this process leads to
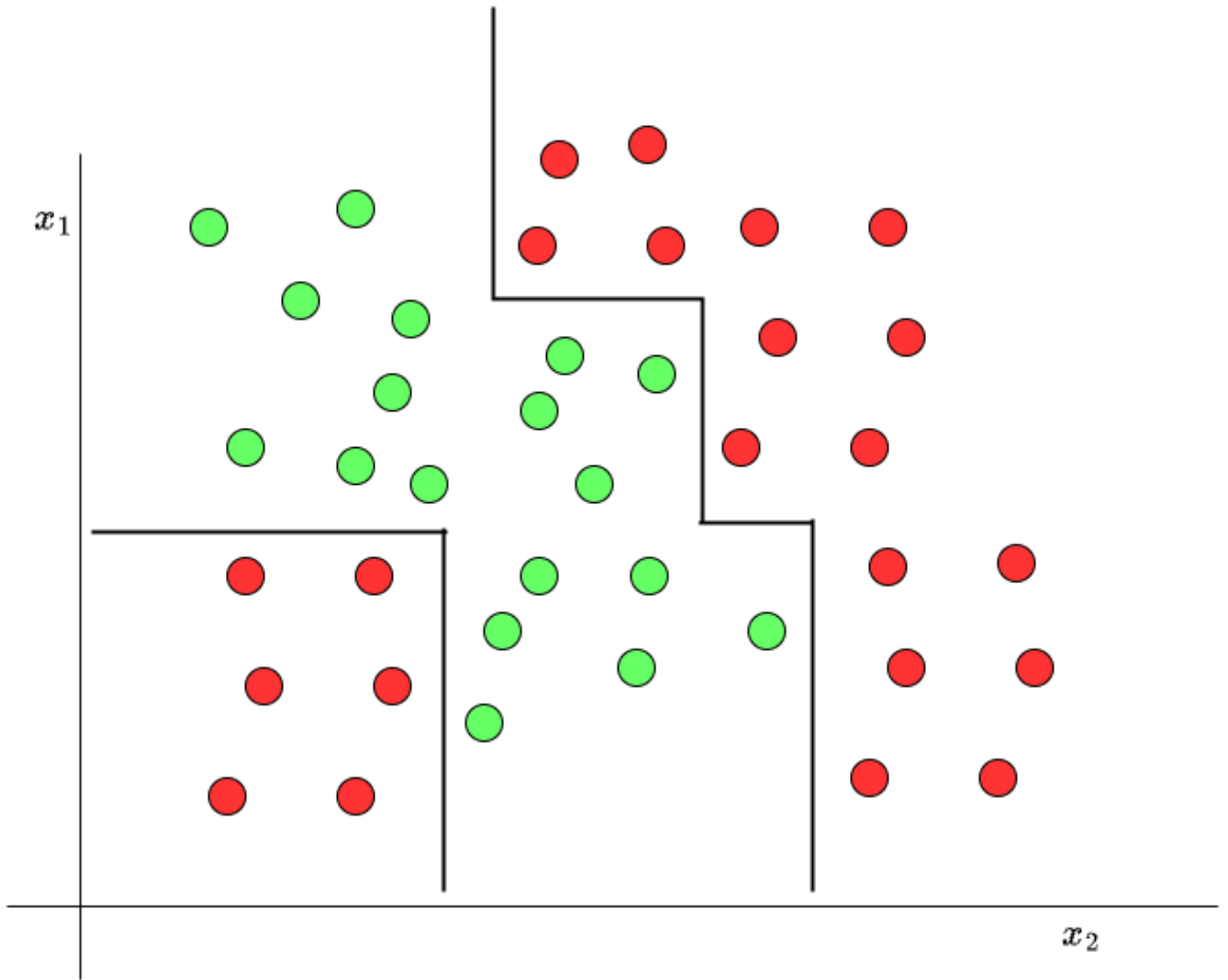
ok, so now that we understand basic idea behind a decision tree that is recursive partitioning of the feature space intuitively; its time to dive in and understand specifics involved in little more detail. Lets begin .

What do decision trees look like ?

The data partition diagrams that we saw above become obsolete way of representing what a decision tree is doing; once we start to have more features in our data, so you will usually find decision tree displayed with more symbolic tree like structure [thats where they draw their name from as well] as shown in the example below .
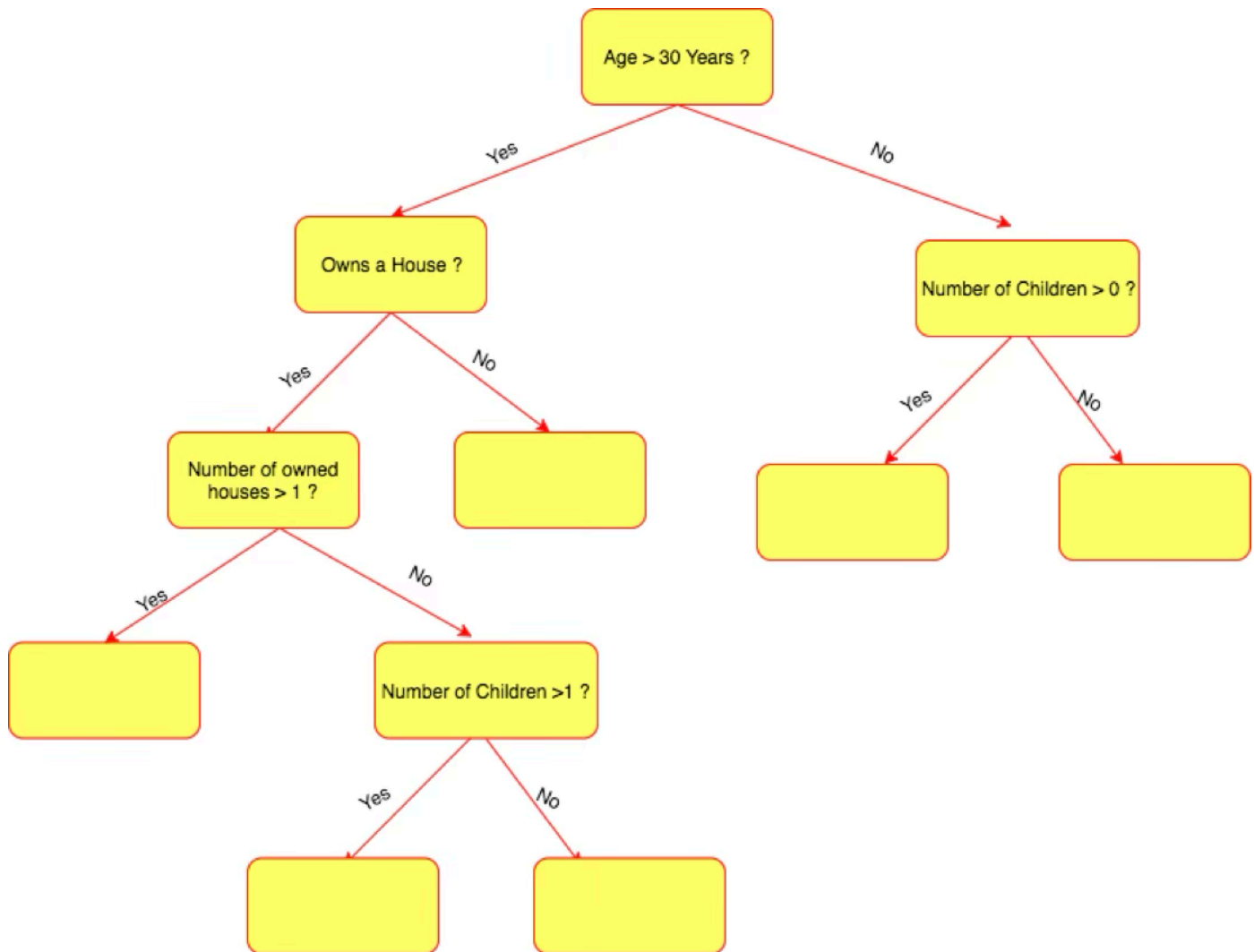
Each box here is known as a `node` . `node` on top is known as `root node` and it represents all the data which we start with . The question that you see in the box or in some diagrams, above it, are known as rules. These rules have binary answers depending on the observation and are used to partition the data in to two parts as you can see in the figure . `node` which is being partitioned is called a `parent node` and the resulting nodes from it are called its `children` . the `child node` which is not being partitioned any further is called a `decision node` or `terminal node` or a `leaf node` or just a `leaf` . Size of a tree can be measured in terms of depth that is levels of partitions that you see, for example the tree in the diagram has depth 4. Another way to measure the size of the tree is to count number of leaf nodes , in the figure , number of leaf nodes are 6.

Its natural ask next is how do we use this tree to make predictions or better, how is this tree constructed . I have broken these curiosities into following questions that we will answer one by one and in the process will make our understanding of decision trees complete .

- How do we make predictions with decision trees ?
- Where do the rules comes from ?
- How do we select rules ?
- When do we stop partitioning the data or when does a node become a leaf node ?

How do we make predictions with decision trees ?

Classification

Someone had given us the rules using which we made the decision tree shown above. Instead, we ask this person to give us the information using which he built the tree i.e. share the data using which he/she could come up with the rules. This data can also be referred to as the training data. We took this training data and passed each observation from this data through the decision tree, resulting in the following tree:
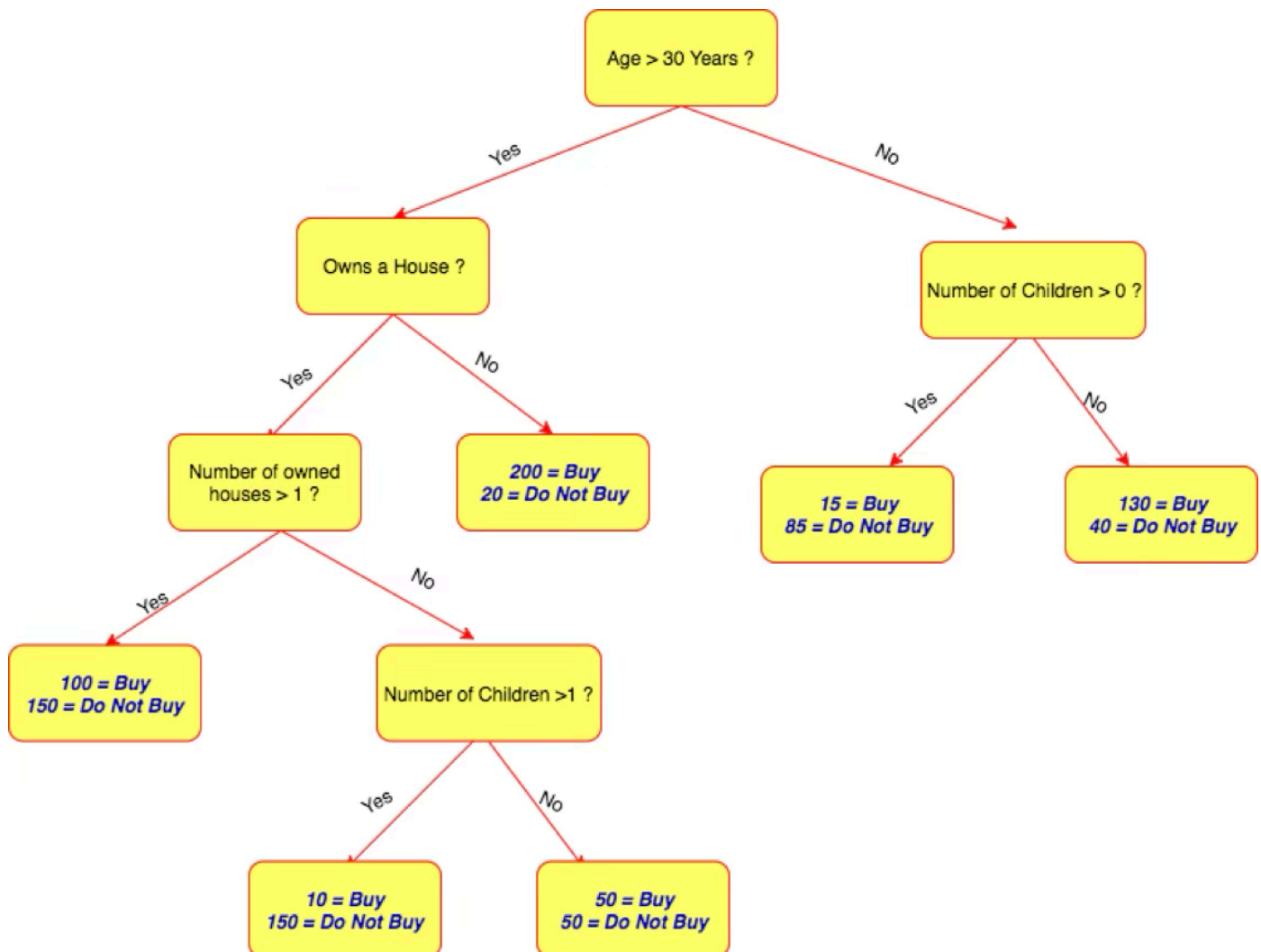
Note the terminal nodes now. We can see that among people with age greater than 30 years and who do not own a house, 200 of them bought the insurance and 20 did not buy. Among people with age less than 30 years and who have children, 15 people bought the insurance and 85 did not buy. Using this information present in the the terminal nodes, we can make prediction for new people for whom we do not know the outcome. For example, lets consider a new person comes whose age is greater than 30 years and does not own a house. Using our training data we saw that most of the people who end up in that node buy the insurance. So our prediction will be that this new person will buy the insurance using a simple majority vote. Instead of using the majority vote, we can also consider the probability of this person buying the insurance i.e. 200/220 = 0.9, which is quite high. We can say that if someone ends up in this terminal node, there is a high chance or probability of this person to buy the insurance.

Now lets consider a person who owns 1 house and has no children, then this person ends in a node where 50% of the people buy the insurance and 50% don't. This probability of a person buying the insurance is as random as a coin flip and hence not desirable.

In short, predictions can be made using either of the following:

1. Probability:

- Given by proportions for the class in the terminal nodes

2. <u>Hard classes</u>:

- Simple majority vote
- Cutoff on the probability score

Regression

When the response variable is continuous, in order to make predictions, we take an average of the response values present in the terminal nodes.

Whether we have a classification decision tree or a regression decision tree, we now know how to make predictions.

But we still don't know how the decision tree was built in the first place. In order to figure this out, we need to answer the next question

Where do the rules comes from ?

Thats one of the easy ones , our data is already numeric post data processing. The ones which have `0` and `1` ; any random split between `0-1` would become the rule.

In almost all modern implementations continuous numeric columns use histogram based process to generate rules . The feature values are first binned, and each bin edge serves as a potential threshold. Binary rules are then formed as "If Feature $X \geq threshold$, go left; otherwise, go right."

How do we select rules ?

For a decision tree to make good predictions , its necessary that resulting nodes from partitions are as homogeneous as possible . Among the many rules available we can select the rules which result in most homogeneous nodes . But we need to quantify this homogeneity . Its different from classification and regression , lets look at classification first .

There are two such popular metrics which are used for classification in `sklearn`,

$$\text{gini index} = 1 - \sum_k p_k^2$$
$$\text{entropy} = -\sum_k p_k \log p_k$$

where p_k is proportion of kth class in the node. Below table gives you example of how these two behave for different composition of data in a node.

# of 0s	# of 1s	Proportion of 0s	Proportion of 1s	Entropy	Gini Index
8	7	0.533	0.467	0.997	0.498
8	6	0.571	0.429	0.985	0.490
8	5	0.615	0.385	0.961	0.473
8	4	0.667	0.333	0.918	0.444
8	3	0.727	0.273	0.845	0.397
8	2	0.800	0.200	0.722	0.320
8	1	0.889	0.111	0.503	0.198
8	0	1.000	0.000	0.000	0.000

You can see here; as the node composition moves towards one class being in the majority; both gini and entropy monotonically decrease; implying, as you can see; lower the metric value , higher the homogeneity in the system. Also note that it doesn't matter which class has higher proportion, all that matters is the homogeneity of the node.

We can use the amount od decrease in the metric as basis to compare 2 rules and then select the one which results in higher decrease. Lets understand that by an example .
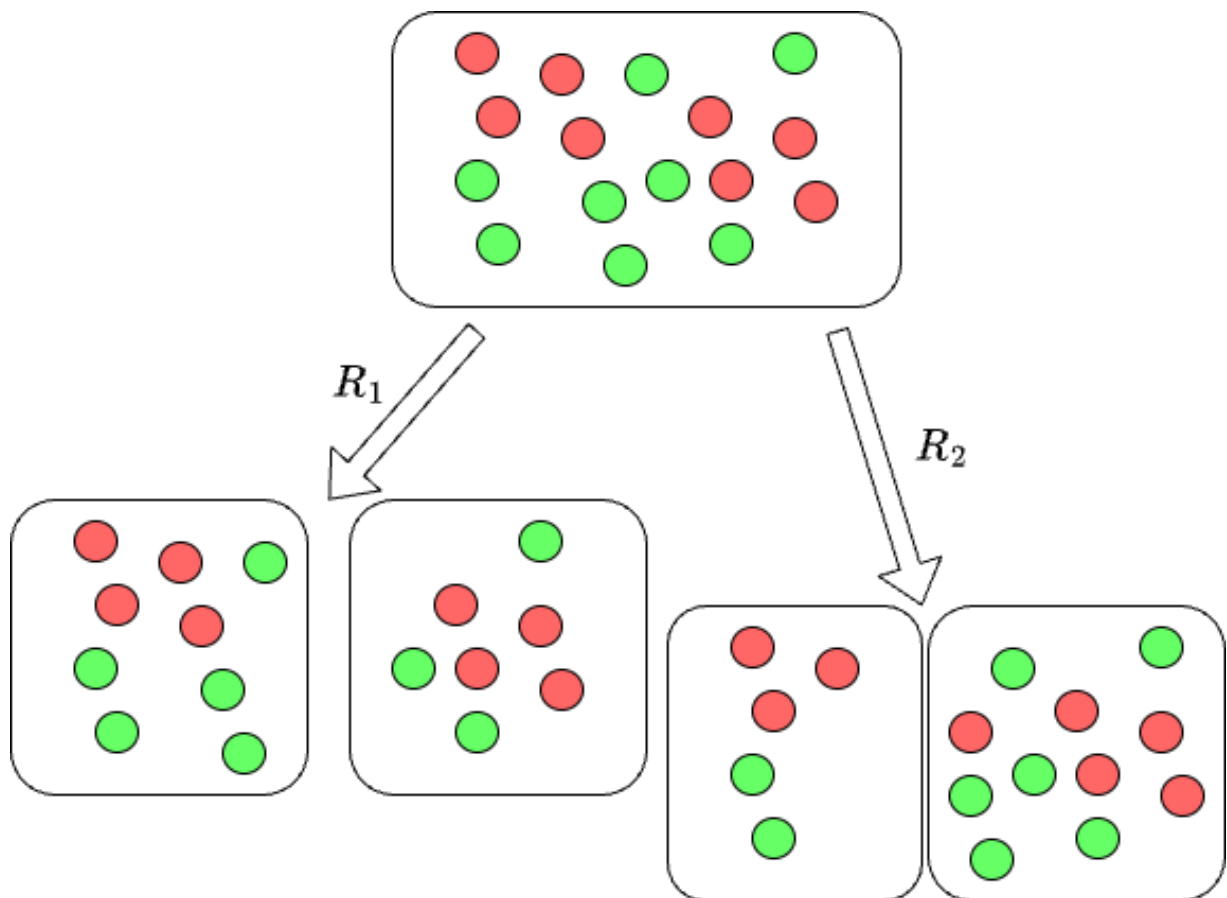
Here gini for the parent node is

$$= 1 - (0.5^2 + 0.5^2)$$
$$= 0.5$$

now homogeneity post R_1 will be measured with weighted sum of gini of the resulting nodes where the weight will be the proportion of the data which has gone to the node from the parent node. So gini for the result coming from R_1 would be :

$$= \frac{9}{16} \times [1 - (\left(\frac{4}{9}\right)^2 + \left(\frac{5}{9}\right)^2)] + \frac{7}{16} \times [1 - (\left(\frac{4}{7}\right)^2 + \left(\frac{3}{7}\right)^2)]$$
$$= \frac{9 \times 40}{16 \times 81} + \frac{7 \times 24}{16 \times 49}$$
$$= \frac{31248}{63504} = 0.492$$

gini for the result coming from R_2 would be :

$$= \frac{5}{16} \times [1 - (\left(\frac{3}{5}\right)^2 + \left(\frac{2}{5}\right)^2)] + \frac{11}{16} \times [1 - (\left(\frac{5}{11}\right)^2 + \left(\frac{6}{11}\right)^2)]$$
$$= \frac{5 \times 12}{16 \times 25} + \frac{11 \times 60}{16 \times 121}$$
$$= \frac{432}{880} = 0.491$$

the comparison here shows that R_2 got more decrease in gini relatively [although very close comparison], hence between the two we will select R_2. This process however can be extrapolated to as many rules as there are .

to make things clear , lets also see an example where target is continuous numeric . metric used will be mean squared error .
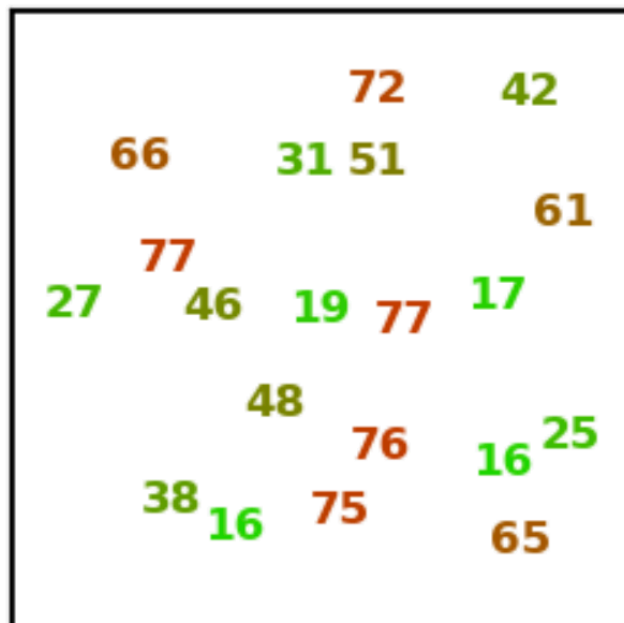
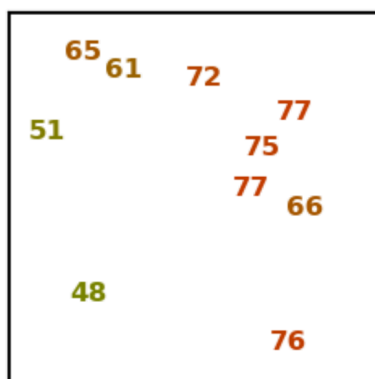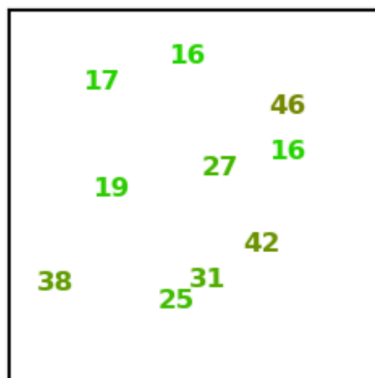mean squared error for this parent node is : 490 . Say R_1 results in split as shown below.
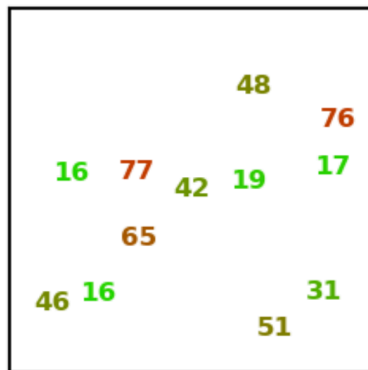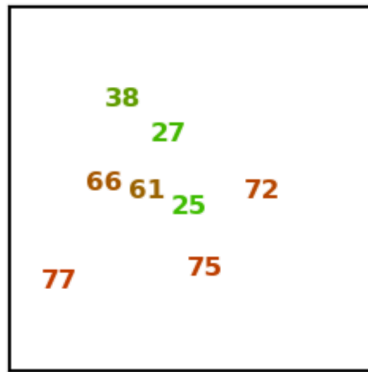
mse for this is

$$= 0.4 \times 413 + 0.6 \times 472$$
$$= 448.4$$

And R_2 results in the split below :

mse for this would be :

$$= 0.5 \times 113 + 0.5 \times 103$$
$$= 108$$

Clearly result from R_2 is more homogeneous , hence among the two R_2 will be chosen and this process can be used for more rules as well.

When do we stop partitioning the data or when does a node become a leaf node

There are some conditions which will force the partitioning to stop naturally :

- Node is left with only one observation

- All the target values in the node are same , there is no point to split any further

- All the feature values are same across observations in the node, there is nothing left to differentiate them

However if any of this leads to decision/leaf nodes having too few observations then our predictions are based on very few data points which is certain recipe for over-fitting . We can control that a little by not actually waiting for partitioning to stop naturally and introduce some constraints to stop the process which ends up in leaf nodes with too few observations . This can be done by following :

- restricting tree size [by controlling `max_depth` or `max_leaf_nodes`] , this essentially stops a tree from growing too big to the point where nodes start to have too few observations

- restricting min number of obs required to split a node [`min_samples_split`]

- disallowing splits which result in child nodes with too few obs [`min_samples_leaf`]

- forcing a minimum decrease in impurity for a split to be allowed [`min_impurity_decrease`]

Random Forests

While decision trees are intuitive and powerful, they suffer from a major drawback: **overfitting**. A single decision tree, if left unchecked, can memorize the training data, leading to poor generalization on new data. Random forests tackle this issue by aggregating multiple decision trees trained with randomness.

Fundamental Idea Behind Random Forests

Instead of relying on a single decision tree, random forests train multiple decision trees and **average their predictions** (for regression) or **take a majority vote** (for classification). The key is that each tree in the forest is trained on a slightly different subset of the data and uses a different subset of features, introducing diversity among trees.

Imagine you are predicting whether it will rain tomorrow. You ask **one** weather expert, and they give you an answer based on their experience. Now, instead of relying on a single expert, you consult **100 different meteorologists**, each of whom has studied different weather patterns from different parts of history. You then aggregate their predictions: If 70 out of 100 say it will rain, you assume a **70% probability of rain**. This is essentially how a random forest works—by taking **the wisdom of a crowd** instead of relying on a single decision tree.

How Are Random Forests Constructed?

Random forests introduce two levels of randomness:

1. **Bootstrap Sampling (Bagging)**
   Instead of training each tree on the full dataset, we create multiple datasets by randomly sampling observations **with replacement**. This means some observations appear multiple times while others may not appear at all in a particular tree. Each decision tree is trained on a different bootstrapped dataset.

   Example: Suppose we have the dataset:
   `D = {A, B, C, D, E, F, G, H, I, J}`
   A bootstrap sample of size 10 might look like:
   `{A, B, C, A, E, F, G, B, I, J}`

   Each observation has a **1/N** chance of being selected in each draw. The probability of **not being selected** after (N) draws is:

   $$\left(1 - \frac{1}{N}\right)^N \approx \frac{1}{e} \approx 0.368$$

   This means that on average, **36.8%** of the observations are left out, and **63.2%** (often rounded to **66%**) of the dataset is included in each bootstrap sample. The unused **36.8%** forms the **out-of-bag (OOB) data**, which can be used for validation.

2. **Feature Randomness (Feature Subsampling)**
   At each split in a tree, instead of considering **all** features, we randomly select a subset of features and find the best split among them. This prevents a few strong features from dominating every tree. It also ensures that individual trees in the forest are uncorrelated and averaging over them actually provides benefit of de-noising .

   Example: Suppose we have features `{x1, x2, x3, x4, x5}` and at a particular node, only `{x1, x3, x4}` are considered for selecting rule for split. This ensures that different trees capture different aspects of the data.

How Do Random Forests Make Predictions?

Once multiple decision trees are trained, predictions are made by aggregating their outputs:

- **Classification**: Each tree predicts a class [with majority voting], and the majority vote on top of that determines the final class or simple proportions on collections of individual tree predictions are used as probability predictions

- **Regression**: Each tree predicts a numeric value, and the final prediction is the **average** of all tree predictions.

Extra Trees (Extremely Randomized Trees)

Extra Trees (short for **Extremely Randomized Trees**) are similar to random forests but introduce **more randomness** in the way they build decision trees. They improve on random forests by further reducing variance at the cost of slightly increased bias.

How Are Extra Trees Different from Random Forests?

While Extra Trees share the **bootstrap sampling** and **feature randomness** of random forests, they make one crucial change:

- **Extra Randomness in Splitting**:
    - In Random Forests, at each split, the best possible threshold is selected from the available features.
    - In Extra Trees, instead of finding the best threshold among all possible values, **a set of random threshold candidates** is chosen for each feature, and then the best threshold among those candidates is selected for splitting.

When to Use Extra Trees vs. Random Forests?

- If the dataset is **noisy**, Extra Trees are preferable since they are less likely to overfit.
- If the dataset has **clear patterns**, Random Forests might perform better since they find optimal splits instead of random ones.
- If **speed** is a concern, Extra Trees train faster than Random Forests.

Although in my experience , it hardly happens in practice [for moderate sized tabular data], that you have to decide on the algorithm to go with before you start building models. Usually you fit multiple models and select the one which performs the best.

Model Inference in Random Forest and Extra Trees

Once a model has been trained using ensemble methods like Random Forest and Extra Trees (ET), the next critical step is **model inference**. This involves interpreting the model's decision-making process using different feature importance techniques. Here, we discuss three main approaches:

1. **Feature Importance from Tree-Based Models**
2. **Permutation Importance**
3. **Partial Dependence Plots (PDP)**

1. Feature Importance from Tree-Based Models

In **Random Forests (RF)** and **Extra Trees (ET)**, feature importance is computed based on how frequently a feature is used in splitting decisions and the corresponding improvement in the objective function (e.g., Gini impurity, entropy, or variance reduction).

The importance of feature j is computed as:

$$I_j = \sum_{t \in T} (w_t \cdot \Delta I_t)$$

where:

- T is the set of all trees in the forest,
- w_t is the number of samples reaching the node where the split occurs,
- ΔI_t is the reduction in impurity (e.g., Gini or variance) due to the split.

Since **scikit-learn normalizes feature importance by default**, the final normalized feature importance values sum up to 1:

$$I_j^{norm} = \frac{I_j}{\sum_k I_k}$$

Issues with Tree-Based Feature Importance

1. **Bias Towards High-Cardinality Features**: Features with many unique values tend to have higher importance scores, even if they are not truly predictive.

2. **Dependence on Correlated Features**: When two features are highly correlated, the importance is arbitrarily split between them, making interpretation difficult.

2. Permutation Importance

Permutation Importance quantifies the contribution of each feature to the model's predictive power by randomly shuffling feature values and measuring the resulting drop in performance.

Given a trained model f and a validation dataset X_{val}, y_{val}, the permutation importance of feature j is computed as follows:

1. Compute the original model performance metric (e.g., accuracy, RMSE, or cross-entropy loss):

$$M_{orig} = M(f(X_{val}), y_{val})$$

   where M is the chosen metric function.

2. Randomly shuffle the values of feature X_j across all samples to create a perturbed dataset $X_{val}^{(j)}$.

3. Compute the new model performance on the perturbed dataset:

$$M_{shuffled} = M(f(X_{val}^{(j)}), y_{val})$$

4. The **Permutation Importance** of feature j is given by:

$$I_j = M_{shuffled} - M_{orig}$$

   A higher value indicates a more important feature.

Advantages of Permutation Importance

1. **Accounts for Feature Redundancy**: Unlike tree-based importance, permutation importance accurately reflects the contribution of correlated features.

2. **Stable Across Runs**: As it evaluates the actual impact of features on predictions, it is more consistent.

3. **Model Agnostic**: Can be used for any modeling algorithm , it does not need the algorithm to be built with decision trees.

Feature	Original Accuracy	Accuracy after Shuffling	Importance Score (Drop)
Feature 1	0.89	0.78	0.11
Feature 2	0.89	0.85	0.04
Feature 3	0.89	0.88	0.01

From this table, **Feature 1** is the most influential, as shuffling its values caused the largest drop in accuracy.

3. Partial Dependence Plot (PDP)

Definition

A **Partial Dependence Plot (PDP)** illustrates how a feature affects the predicted outcome while averaging over all other features.

Mathematically, the partial dependence function of feature X_j is computed as:

$$PD(X_j) = \frac{1}{N} \sum_{i=1}^{N} f(X_j, X_{\neq j}^{(i)})$$

where:

- f is the trained model,
- X_j is the feature of interest,
- $X_{\neq j}^{(i)}$ represents all **other** features for sample i,
- N is the number of samples.

This is something which will be much more intuitive when you look at it visually yourself. Refer to the notebook for the same.

we will stop this discussion here , do refer to the notebook for learning how to build/tune/infer these models and other things in python.