Problem statement

The problem that we will solve is related to the banking sector. A bank has put out a marketing campaign and wants to know how the campaign is working. Given the features of the client and the marketing campaign, we have to predict whether the customer will subscribe to a term deposit or not. This is a classification problem, which we will now try to solve using decision trees. The original dataset has 17 features and around 11000 data points. To get a deeper understanding of the problem, you can read more about the problem here.

For the purpose of understanding decision trees, we will look at a smaller dataset (subset of our dataset) with lesser features and fewer data points.

- `age (continuous)` - age of the bank customer
- `balance (continuous)` - average yearly balance in euros
- `duration (continuous)` - last contact duration, in seconds
- `campaign (discrete)` - number of contacts performed during this campaign and for this client
- `housing_cat (categorical)` - does the customer have a housing loan? (yes(1)/no(0))

Output: y - has the client subscribed a term deposit? (binary: "yes","no")

The smaller dataset will be used to make you understand the different topics related to decision trees. But, we will use the complete dataset in the exercises. The features of the complete dataset will be discussed later.

Throughout learning this concept, you will be doing the following:-

- Understand and load the dataset.
- Perform a basic exploratory analysis of the dataset.
- Know how a decision tree looks like and the various terminology associated with the decision tree.
- Learn in detail, how a decision tree is constructed.
- Learn about overfitting in decision tree and how it can be prevented.

At the end of this concept, you must be able to apply decision tree learning on any dataset and able to interpret it.

With the overall goal in the top of our mind, let us take a look at a subset of dataset for learning and motivate the learning for decision tree.

Features of data

Dataset is loaded with the name `bank_small` and the features are shown in the image below:

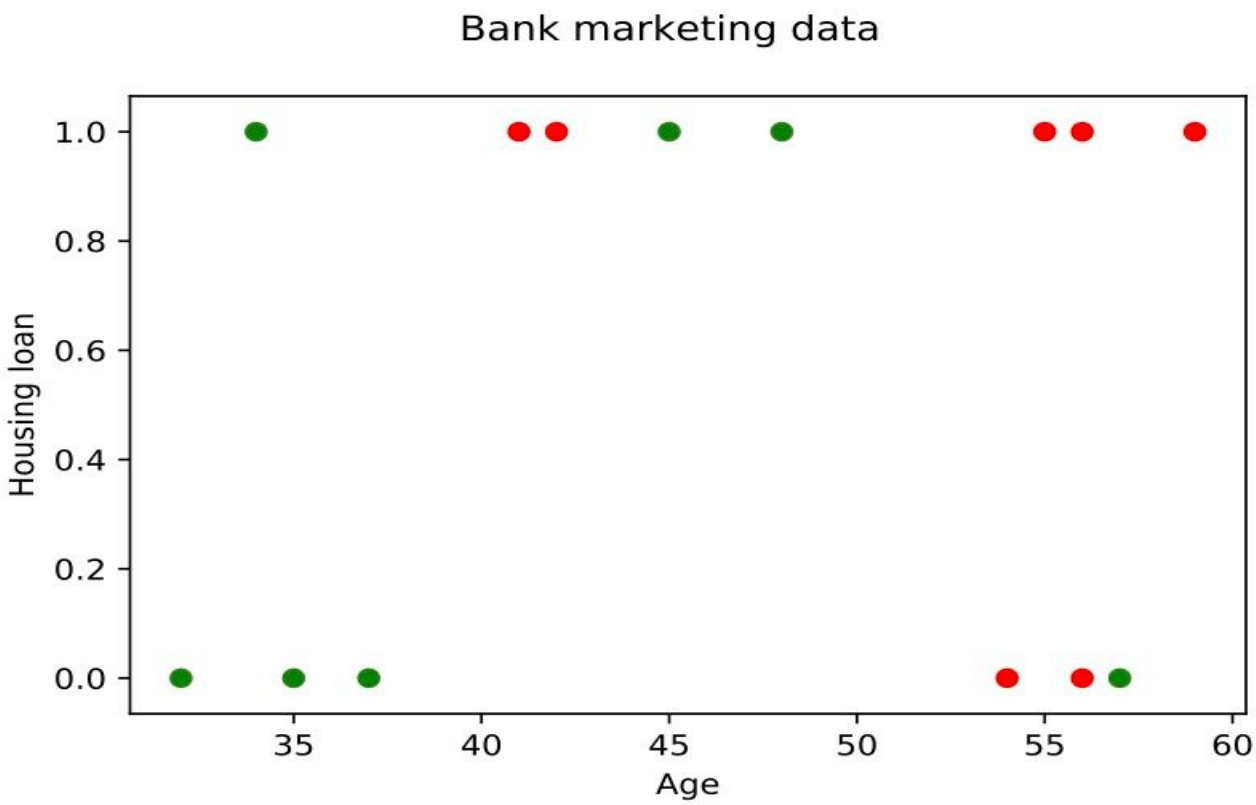
	age	balance	campaign	housing_cat
0	59	2343	1	1
1	56	45	1	0
2	41	1270	1	1
3	55	2476	1	1
4	54	184	2	0
5	42	0	2	1
6	56	830	1	1
7	57	604	1	0
8	45	0	1	1
9	48	238	2	1
10	34	673	1	1
11	37	7944	1	0
12	32	3696	3	0
13	35	347	1	0

Labels of data

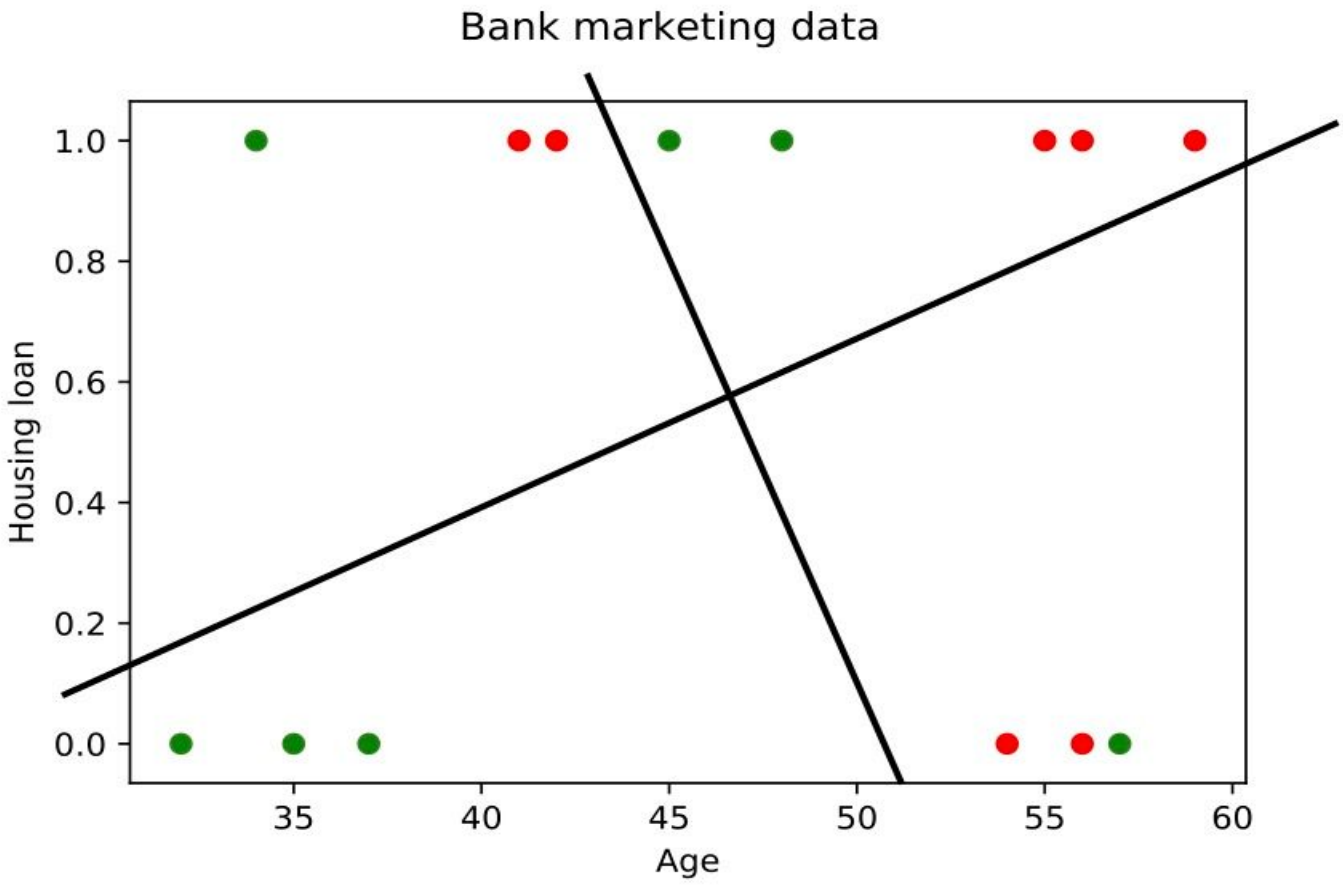
Now we will look at the corresponding labels denoted by `bank_small_labels`- who has subscribed a term deposit with the bank?

	deposit_cat
0	1
1	1
2	1
3	1
4	1
5	1
6	1
7	0

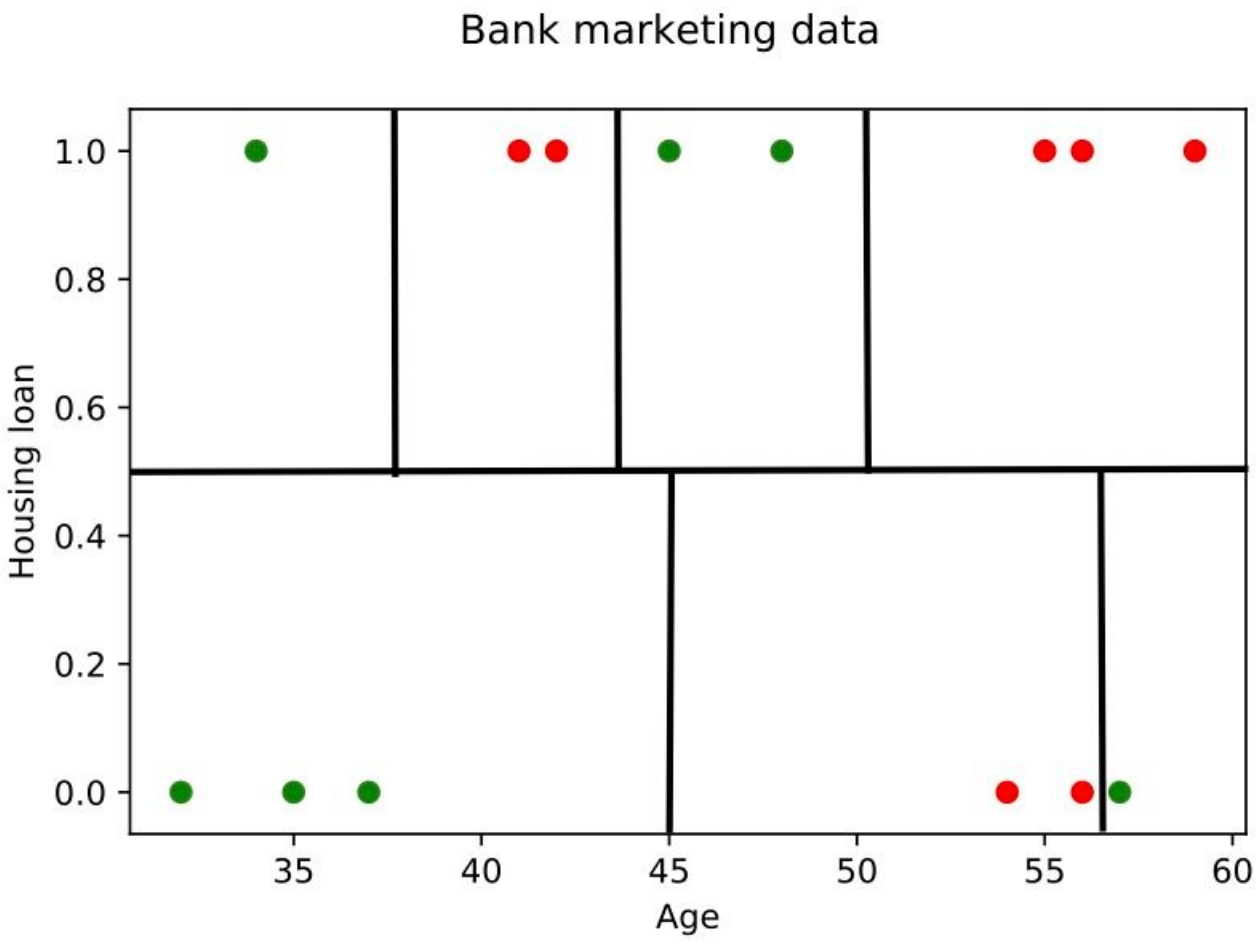
The 1s mean that the first six customers in our bank have opened a term deposit with the bank and the others have not. We will build the decision tree on this data. First let us motivate why decision trees are so powerful.



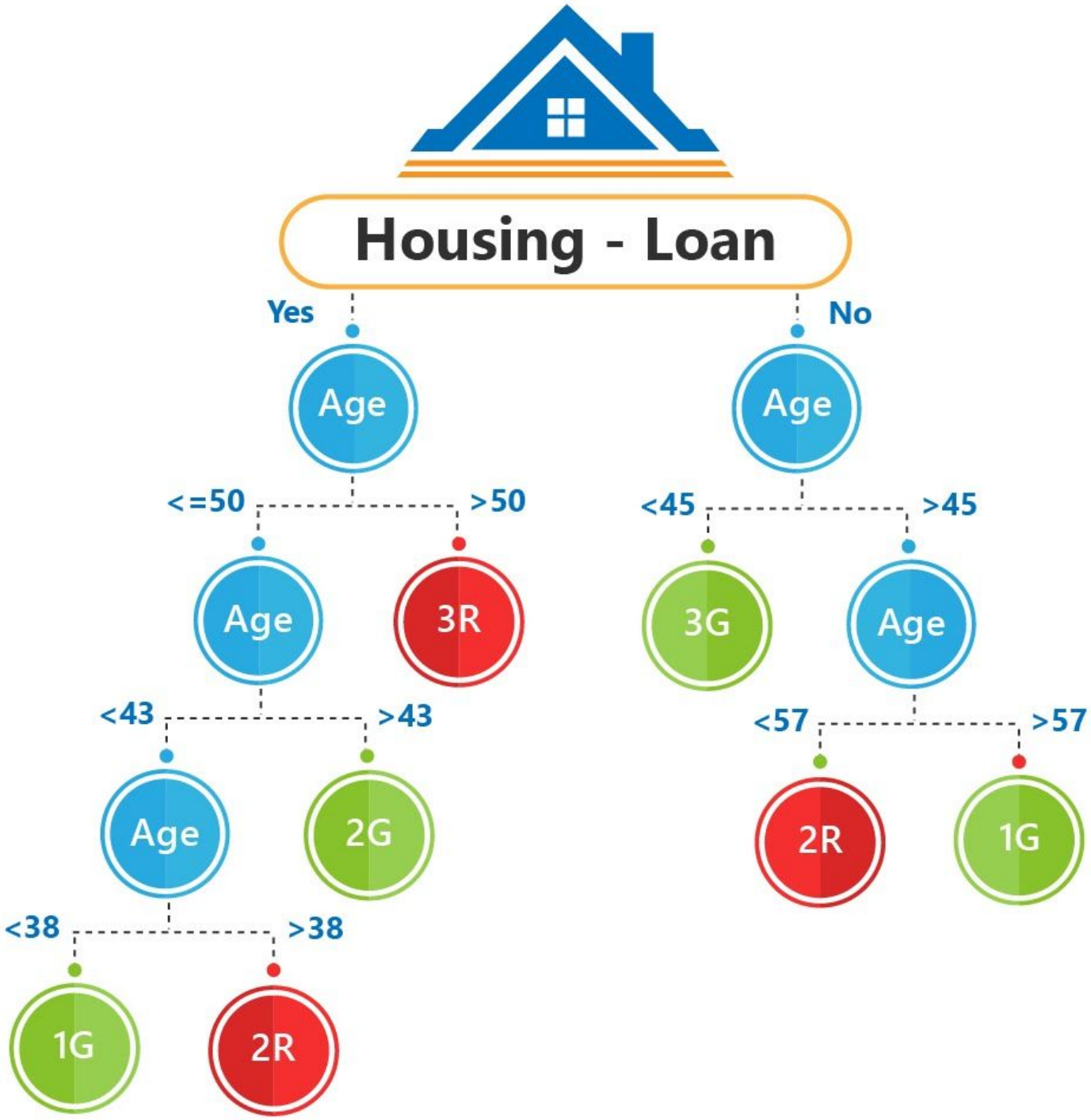
On the x-axis, we have age of the person and on the y-axis, we have the data on whether he has taken a housing loan or not. From the plot, we can see that the data is not linear, and there is no way to achieve zero classification error through a line.



But let us consider another approach. Can we separate the data along the axes and divide into regions so that each region contains only items of one class? Let us make an attempt.



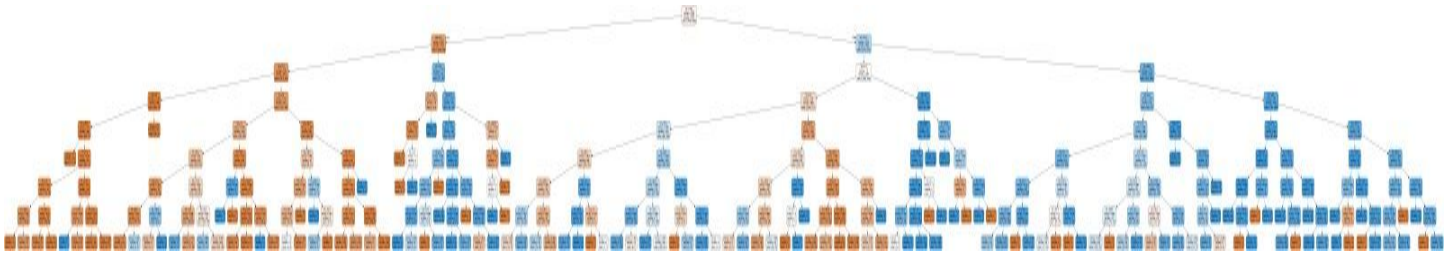
Each of the regions contains members of one class only. So we have achieved zero classification error this way. Let us try to visualize how we had constructed the regions.



The above picture is the visualization of how we cut the axes to divide the data into regions so as to achieve a zero classification error. That's it!! What we have done is - construct a decision tree. Decision trees are powerful non-linear methods.

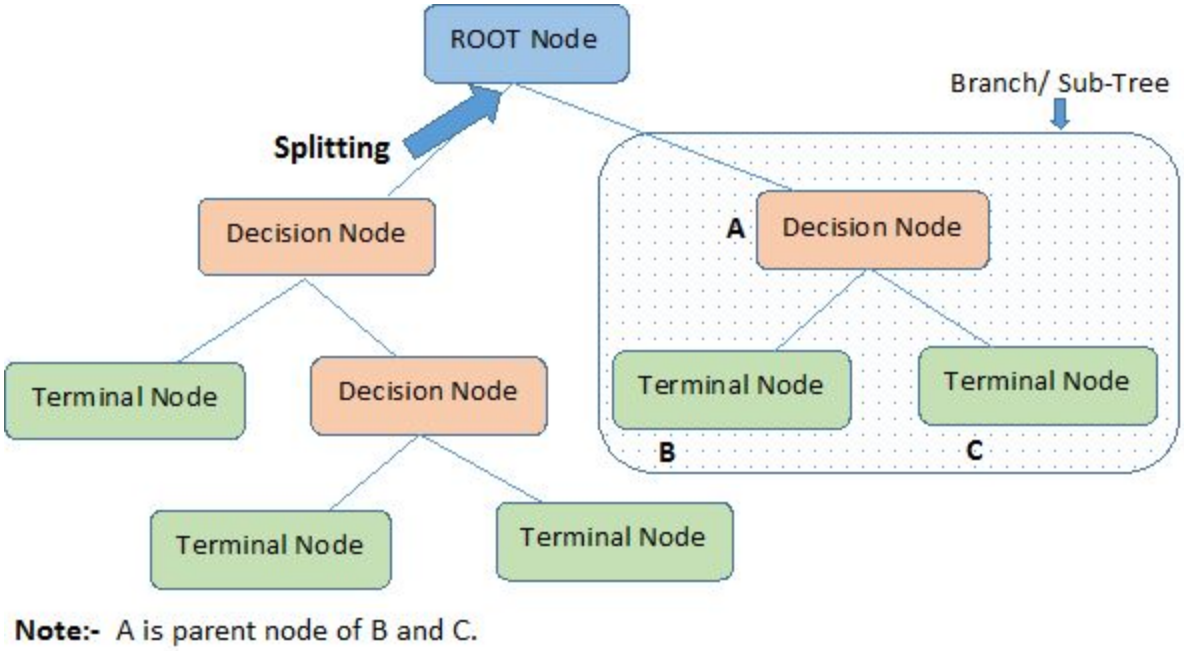
Terminology of a decision tree

In the previous topic, you had a hands-on look at a decision-tree we had constructed by hand. We will now try to understand in detail the various parts of the decision tree. First let us try to look at a real-life decision tree.



The decision tree looks really big right!!! This tree has been built on the entire data from which we had created the toy subset (and we had put some limits due to size constraints!!). We will get our hands dirty working on this data. And by industry standards, this data size is less since it consists of only 11000 instances, while actual industry data runs into millions of instances.

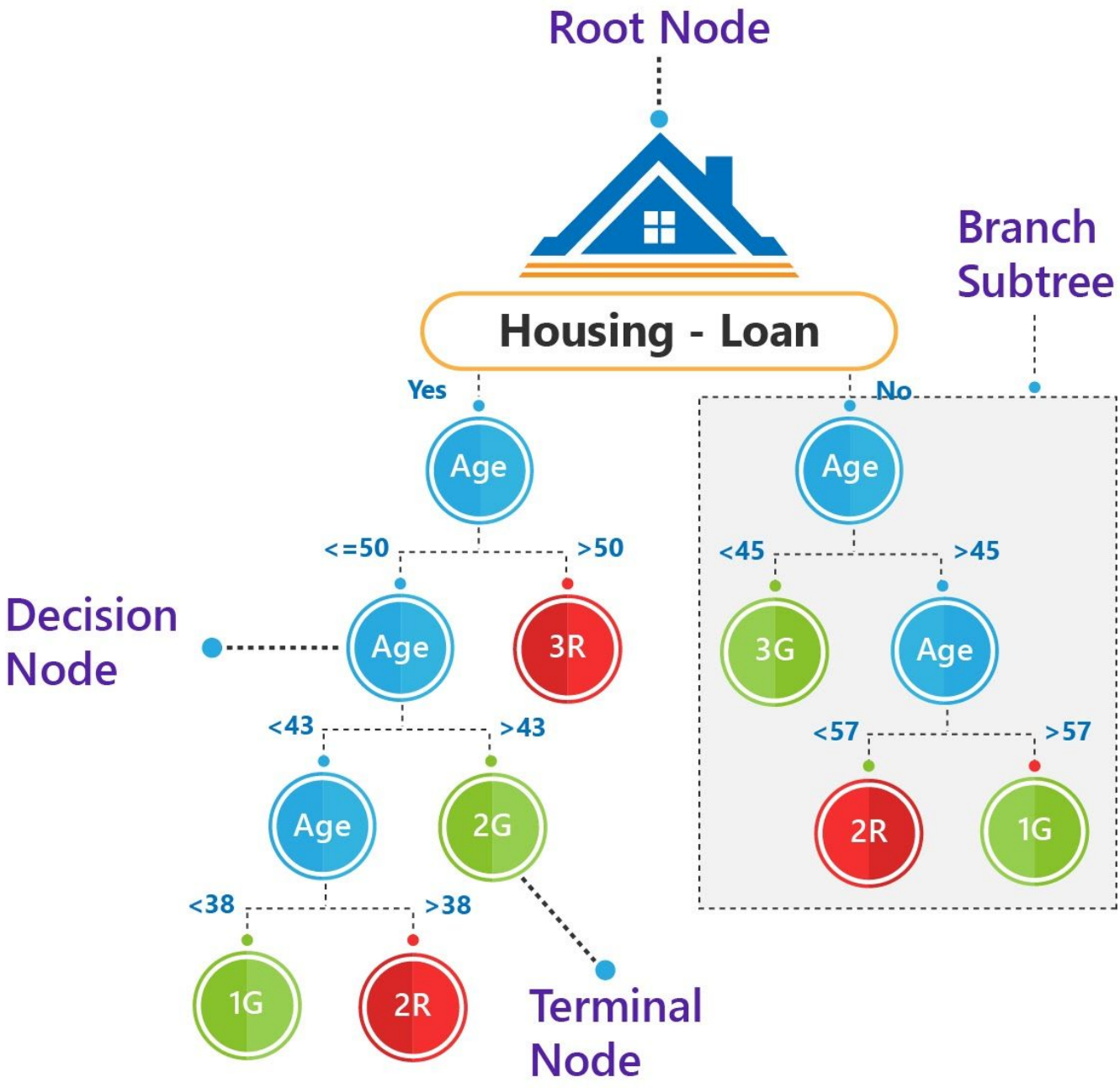
So before that let's try to understand the basic terminology associated with the decision tree.



Decision tree terminology

- Root Node: Entire population or sample, further gets divided into two or more homogeneous sets.
- Parent and Child Node: Node which is divided into sub-nodes is called parent node, whereas sub-nodes are the child of parent node.
- Splitting: Process of dividing a node into two or more sub-nodes.
- Decision Node: A sub-node that splits into further sub-nodes.
- Leaf/Terminal Node: Nodes that do not split.
- Pruning: When we remove sub-nodes of a decision node, this process is called pruning. (Opposite of Splitting)
- Branch/Sub-Tree: Sub-section of entire tree.

Now we shall look at the same decision tree we have built in the previous topic and mark the different nodes.



We will now look at the decision tree on our complete toy data. We will just visualize the data and not bother about the information contained or how the tree is actually constructed.

Deeper understanding of the problem

In the previous topic, you have understood the need for decision tree and its comparison to linear models. In this topic, we will understand the bank marketing data in more detail. We will look at some more features of the dataset. These features are a subset of the original 17 features of the original dataset on which some preprocessing is done. Before we apply machine learning algorithm on any data, this analysis is very essential, and some of this has already been covered in the data pre-processing module. Let us look at the features of the entire data.

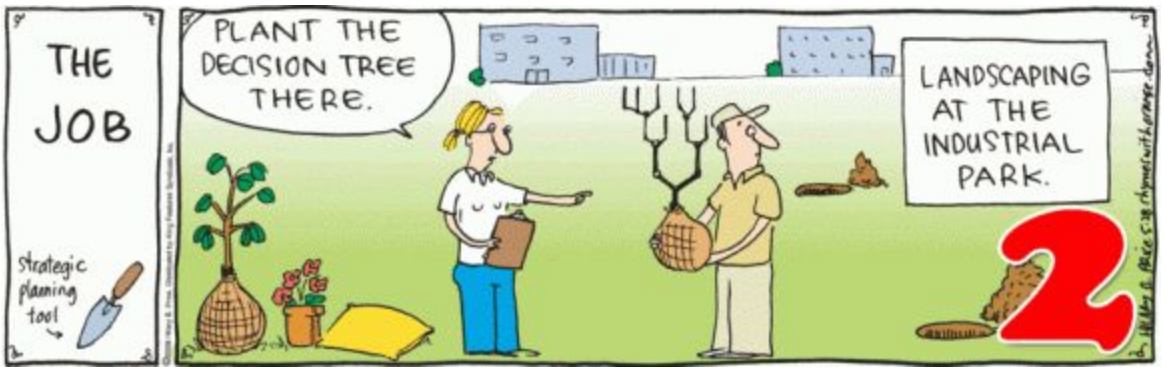
Input variables:

- bank client data:
1. `age` (numeric)
2. `balance`: average yearly balance, in euros (numeric)
3. `housing_cat`: has housing loan? (binary: "yes","no")
- related with the last contact of the current campaign:
1. `duration`: last contact duration, in seconds (numeric)
- other attributes:
1. `campaign`: number of contacts performed during this campaign and for this client (numeric, includes last contact)
2. `pdays`: number of days that passed by after the client was last contacted from a previous campaign (numeric, 10000 means client was not previously contacted)
3. `poutcome_success`: whether the previous marketing campaign was successful(categorical: 1("success"),0("failure or unknown"))
- output variable (desired target):
1. `y` - has the client subscribed a term deposit? (binary: "yes","no")

- Decision Tree is Flow-Chart & Structure in which internal node represents test on an attribute, each branch represents outcome of test and each leaf node represents class label.

- A Decision tree is a support tool that uses a tree-like graph or model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility.

Algorithm for building a decision tree

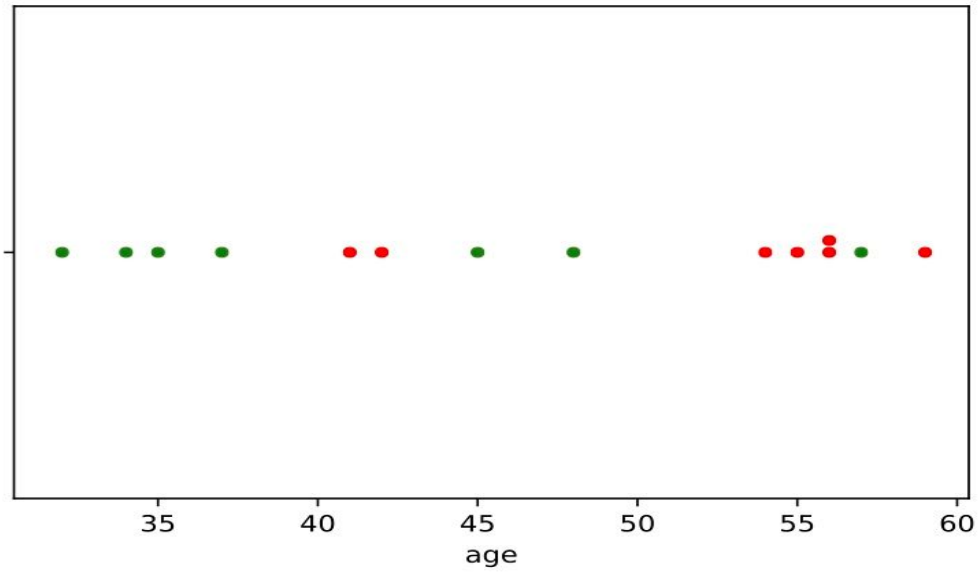


To build a decision tree, let's look at a naive approach and then refine the algorithm step-by-step. Basic Algorithm

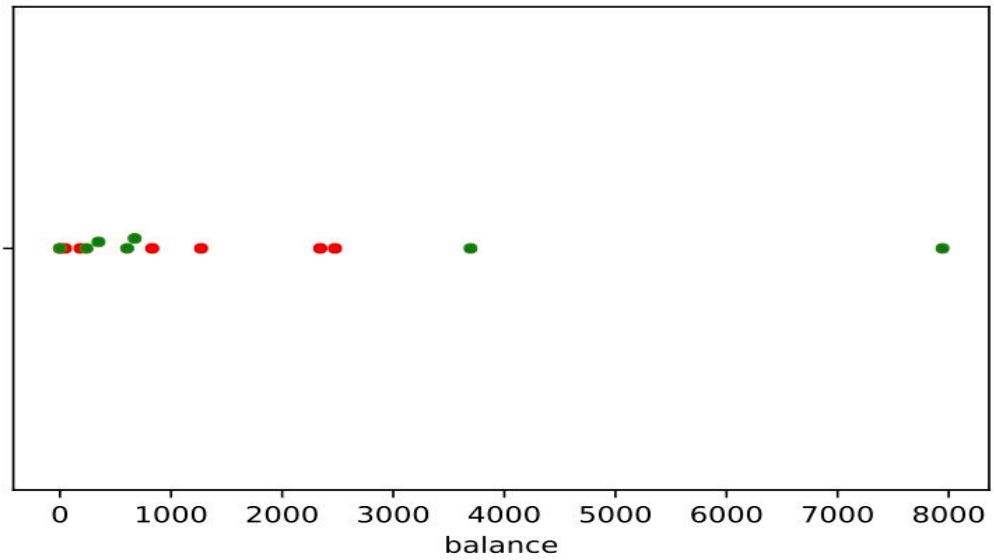
1. Start with an empty tree.
2. Choose a feature to split the tree on.
3. For the split,

- if all examples are positive or negative
    - make the prediction.
- else
    - go to step 2

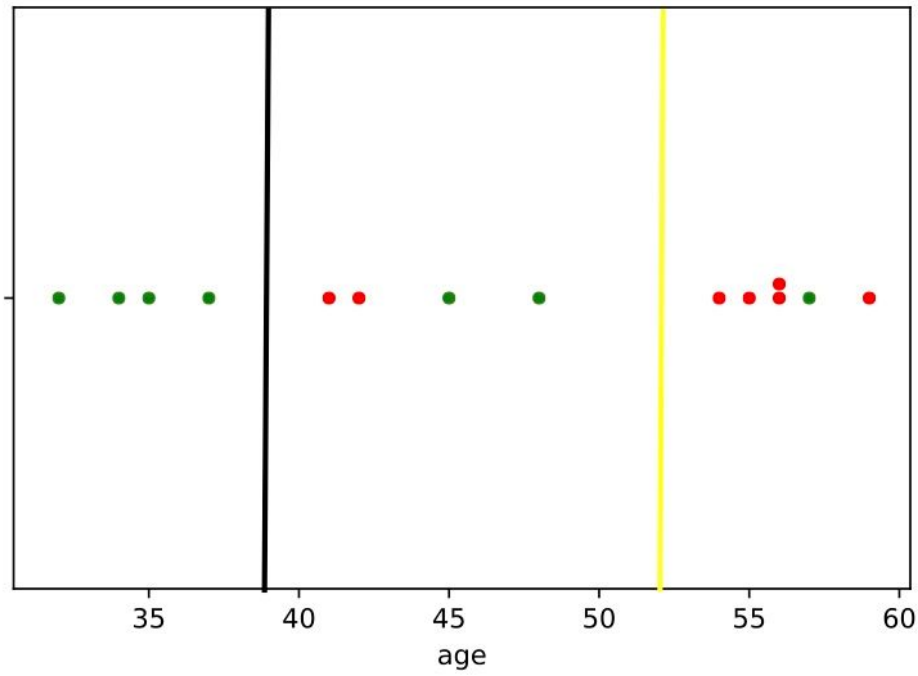
In the earlier topics, we have used visual inspection to decide the feature for splitting. Let us try that approach on our toy data. We will take two attributes - `age` and `balance` and try to decide which feature to use. (There are other features too. But, for now, let us just use these two.) First, let us visualize the classwise distribution on age.



This is a univariate visualization of the data w.r.t parameter `age`. All points which are red correspond to the persons who have a term deposit with the bank. All points which are green correspond to the persons who do not have a term deposit with the bank.



Like earlier, we have the classwise visualization of the data w.r.t the parameter `balance`. Like described earlier, red points correspond to the ones having a term deposit with the bank and green points correspond to the ones not having a term deposit. Intuitively, we would like to take decisions as fast as possible. So we would want the split to ideally bifurcate the data into sets which belong predominantly to one class.

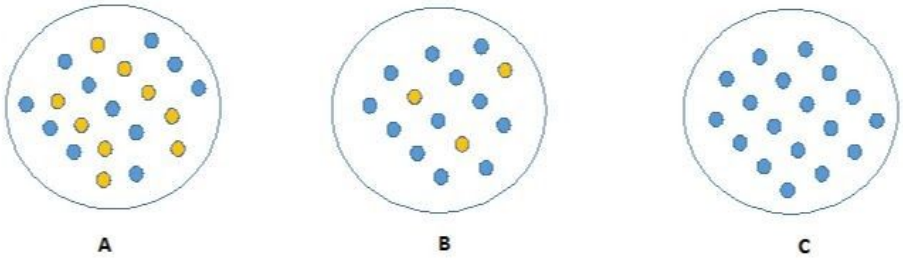


If we divide through age at point ~39 (black line), we divide the data into two sets:

- 4 data points of one class (age < 39)
- 10 data points with 3 of one class and 7 of another class. (age > 39) If we divide through age at point ~52 (yellow line), we divide the data into two sets:
- 8 data points with 2 of one class and 6 of another (age < 52)
- 6 data points with 5 of one class and 1 of another (age > 52) We would like to have all the points in one subset to belong to the same class, so that when any point falls in that bucket, we can take an unambiguous decision regarding its class label. Hence, we choose the first type of division i.e. we split the data at age~39. But this visual inspection might not be possible when you have huge datasets. (It is even difficult for our dataset of 11000 points!)

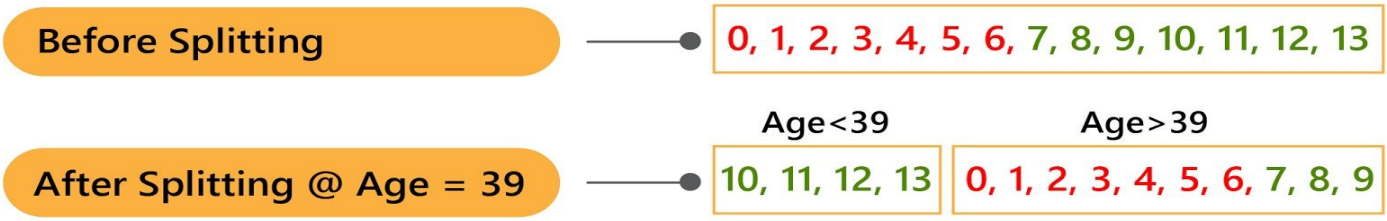
Criterion for splitting a decision tree - Information Gain

How do we decide the criterion for choosing the feature for splitting the decision tree? Let us visually build the intuition...



Consider the amount of information used to describe the pictures. C can be described easily, but B requires more information to describe it and A requires even more information. In other words, we can say that C is pure and A is impure.

Let's go back to the toy bank dataset. Without the root node, we have the collection of item which are impure. Then with the addition of root node, look at the subsets of items w.r.t purity.



After splitting, you can visually see that the overall purity of the subsets is improving. (Especially, the age<39 subset is completely pure.) This notion of purity is captured formally by the concept of entropy.

Entropy is defined as the degree of disorganization in a system. For a completely impure dataset, the entropy is high (~1) and for a completely pure dataset the entropy is low (~0). We will now look at the reduction in entropy / gain in information as we split the node.

The formula for entropy is given by

$$Entropy = -p \log p - q \log q$$

, where p is the probability of choosing the positive class and q is the probability of choosing the negative class.

Before splitting, in the toy dataset, there are 7 positive examples and 7 negative examples (and totally 14 example). The probability of choosing the positive class is $\frac{\text{no of examples in positive class}}{\text{total no of examples}}$. So the entropy is

$$Entropy[7_+, 7_-] = -\frac{7}{14} \log \frac{7}{14} - \frac{7}{14} \log \frac{7}{14} = -0.5 \log 0.5 - 0.5 \log 0.5 = 1$$

Hence, the data is completely impure and the entropy is high. Now let us look at the subsets after splitting.

After splitting at age=39, for the different subsets we will calculate the entropy separately.

Age < 39,

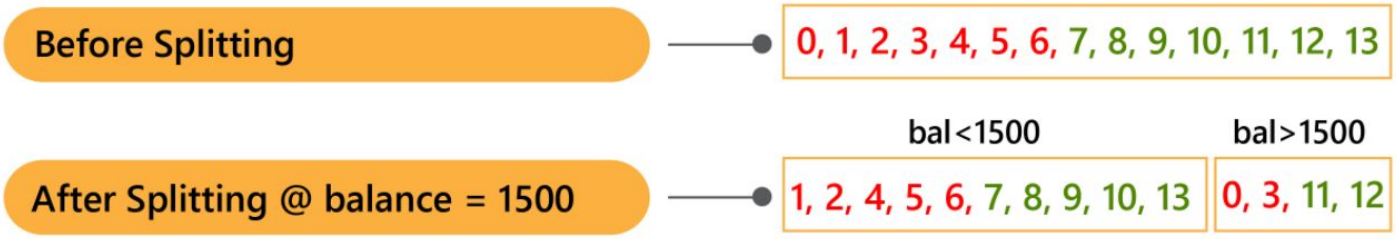
$$Entropy_{\{age<39\}}[0_+, 4_-] = -\frac{0}{4} \log \frac{0}{4} - \frac{4}{4} \log \frac{4}{4} = 0$$

Age > 39,

$$Entropy_{age>39}[7_+, 3_-] = -\frac{7}{10} \log \frac{7}{10} - \frac{3}{10} \log \frac{3}{10} = 0.88$$

So the total entropy is weighted sum of the subsets, where the weights is the ratio of number of elements in subset to the total number of data points in the set.

$$Entropy[age = 39] = \frac{4}{14} Entropy_{age<39} + \frac{10}{14} Entropy_{age>39} = \frac{4}{14} \times 0 + \frac{10}{14} \times 0.88 = 0.63$$



Now let's look at the split at balance=1500. For the different subsets we will have:-

balance < 1500,

$$Entropy_{bal<1500}[5_+, 5_-] = -\frac{5}{10} \log \frac{5}{10} - \frac{5}{10} \log \frac{5}{10} = 1$$

balance > 1500,

$$Entropy_{bal>1500}[2_+, 2_-] = -\frac{2}{4} \log \frac{2}{4} - \frac{2}{4} \log \frac{2}{4} = 1$$

So the total entropy is weighted sum of the subsets, where the weights is the ratio of number of elements in subset to the total number of data points in the set.

$$Entropy[bal = 1500] = \frac{10}{14} Entropy_{bal<1500} + \frac{4}{14} Entropy_{bal>1500} = \frac{10}{14} \times 1 + \frac{4}{14} \times 1 = 1$$

Gain through splitting with age=39,

$$InformationGain(IG) = Entropy[\text{before split}] - Entropy[\text{after split}]$$

Gain for age=39 ,

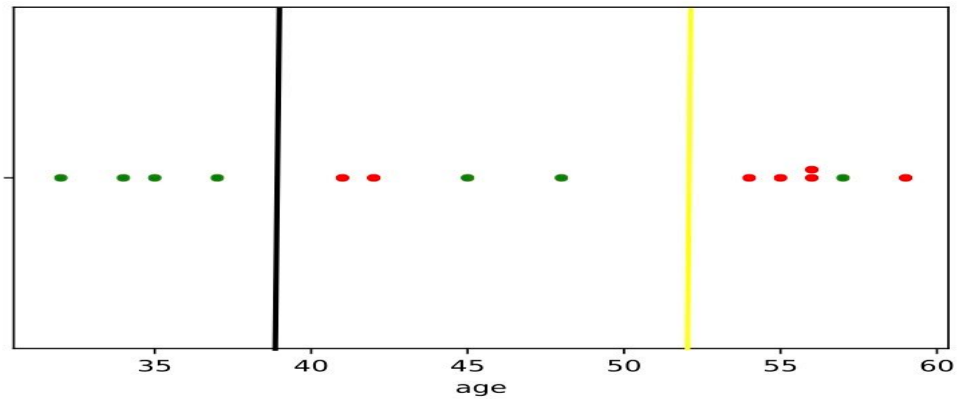
$$IG[age = 39] = Entropy[\text{before split}] - Entropy[\text{split with age=39}] = 1 - 0.63 = 0.37$$

Gain for balance=1500 ,

$$IG[bal = 1500] = Entropy[\text{before split}] - Entropy[\text{split with bal=1500}] = 1 - 1 = 0$$

Since we have more gains through age=39, we choose age=39 as a splitting attribute over balance=1500.

In the same way, let us revisit our splitting criterion on age w.r.t our previous topic. Let us look at the splitting point of age itself (age=39 or age=52).



IG(age=39) = 0.37. Now let's look at IG for age=52. Quickly summarizing our calculations,

Age < 52,

$$Entropy_{age<52}[2_+, 6_-] = -\frac{2}{8}\log\frac{2}{8} - \frac{6}{8}\log\frac{6}{8} = 0.81$$

Age > 52,

$$Entropy_{age>39}[5_+, 1_-] = -\frac{5}{6}\log\frac{5}{6} - \frac{1}{6}\log\frac{1}{6} = 0.65$$

$$Entropy[age = 52] = \frac{8}{14}Entropy_{age<52} + \frac{6}{14}Entropy_{age>52} = \frac{8}{14} \times 0.81 + \frac{6}{14} \times 0.65 = 0.74$$

IG(age=52) = 1 - 0.74 = 0.26

Still, IG(age=39) > IG(age=52), therefore our choice of splitting through visual inspection was good. Now, we also have a formal backing of Information Gain also for choosing the point of split.

So, we have a better information gain with age=39 instead of age=52. But do we check all the possible split points? Do we check the split at age = 10,20,30... and so on? Is there some heuristic to reduce the search space?

Yes. **We only check split-points that lie between the examples of different classes for those can give splits of better purity**. After we split at the root node, we look at the subsets of data independently and again look at all numerical features and possible split-points.

Now we will look at a hands-on implementation of entropy on the toy dataset in python. We define a modified log2 code to handle the log zero error. Then we calculate entropy on the entire dataset.

```
from math import log2

def mod_log2(temp): # modified log2 to prevent zero error

    if temp:

        return log2(temp)

    else:

        return log2(temp+0.00000000000000000000000000001) # just a small value to add!

# entropy of dataset before split

total_label_dist = bank_small_labels['deposit_cat'].value_counts().to_dict()


# total number of points

num_total_data = bank_small_labels.count()[0]


# number of positive class points

pos_class_count = bank_small_labels[bank_small_labels['deposit_cat']==1].count()[0]

# number of negative class points

neg_class_count = bank_small_labels[bank_small_labels['deposit_cat']==0].count()[0]


# calculate the full entropy of the data
```

```
entropy_full_data =
-(pos_class_count/num_total_data)*mod_log2(pos_class_count/num_total_data) + \

-(neg_class_count/num_total_data)*mod_log2(neg_class_count/num_total_data)

print("The entropy of the data before splitting is %.2f.\n" % entropy_full_data)
```

The output is:

```
The entropy of the data before splitting is 1.00.
```

Now let's look at the implementation of the entropy after the split on our toy data. We first split the data based on the splitting point and then get the distribution of class labels. Then we take counts of data in the positive class (with class label 1) and negative class.

```
# take the logical array of the indices for the data split

age_left_split = bank_small['age'] < 39


# taking the logical not operator to get the indices for the right split

age_right_split = ~age_left_split


# getting the class labels of the left split

age_left_split_labels = bank_small_labels[age_left_split]

# number of elements in the left split

num_left_split = age_left_split_labels.count()[0]


# getting the class labels of the right split

age_right_split_labels = bank_small_labels[age_right_split]

# number of elements in the right split

num_right_split = age_right_split_labels.count()[0]


# get the class label distribution

left_split_dist = age_left_split_labels['deposit_cat'].value_counts().to_dict()

right_split_dist = age_right_split_labels['deposit_cat'].value_counts().to_dict()
```

For each of the splits, we individually calculate the entropy and then calculate the total entropy after the split.

```
# entropy of dataset after split - left side

try:

    pos_cc_left_split = left_split_dist[1] # positive class counts

except KeyError:

    pos_cc_left_split = 0


try:

    neg_cc_left_split = left_split_dist[0] # negative class counts

except KeyError:

    neg_cc_left_split = 0




entropy_age_left_split =
-1*((pos_cc_left_split/num_left_split)*mod_log2(pos_cc_left_split/num_left_split)) + \

-1*((neg_cc_left_split/num_left_split)*mod_log2(neg_cc_left_split/num_left_split))


# entropy of dataset after split - right side

try:

    pos_cc_right_split = right_split_dist[1] # positive class counts

except KeyError:

    pos_cc_right_split = 0


try:

    neg_cc_right_split = right_split_dist[0] # negative class counts

except KeyError:

    neg_cc_right_split = 0


entropy_age_right_split =
-1*((pos_cc_right_split/num_right_split)*mod_log2(pos_cc_right_split/num_right_split)) + \

-1*((neg_cc_right_split/num_right_split)*mod_log2(neg_cc_right_split/num_right_split))
```

```
# combining both to get the correct entropy

entropy_after_split_age = (num_left_split/num_total_data)*entropy_age_left_split + \

(num_right_split/num_total_data)*entropy_age_right_split

print("The entropy of the data after splitting with age is %.2f.\n" %
entropy_after_split_age)
```

```
The entropy of the data after splitting with age is 0.63.
```

In a similar way, we calculate the entropy of data after splitting at balance<1500. After that, we calculate the information gain for each attribute.

```
#total_entropy - entropy_after_split_age

print("The information gain after splitting with age = 39 is %.2f.\n" % (entropy_full_data -
entropy_after_split_age))

print("The information gain after splitting with balance is %.2f.\n" % (entropy_full_data -
entropy_after_split_bal))
```

The output is:

```
The information gain after splitting with age = 39 is 0.37.

The information gain after splitting with balance is 0.00.
```

We have looked at information gain/entropy as the splitting criterion. There is another popular splitting criterion for decision tree called gini.

Criterion for splitting a decision tree - Gini

Like entropy, gini is another way of measuring impurity.

A Gini score gives an idea of how good a split is by how mixed the classes are in the two groups created by the split (similar to entropy).

- It works with categorical target variable “Success” or “Failure”
- A Gini score gives an idea of how good a split is by how mixed the classes are in the two groups created by the split.
- It measures how often a randomly chosen element would be incorrectly identified.
- A perfect separation results in a Gini score of 0, whereas the worst case split that results in 50/50 classes in each group results in a Gini score of 1.0 (for a 2 class problem).

- Higher the value of Gini higher the homogeneity

How to calculate gini for a split?

1. Calculate Gini for sub-nodes, using formula sum of square of probability for success and failure

$$(1-(p^2+(1-p)^2)$$

.

2. Calculate Gini for split using weighted Gini score of each node of that split

We will calculate the gini index for the same example as earlier.

First let's define success and failure in our case. Success is having a term deposit with the bank (red colored class). So automatically failure means not having a term deposit.

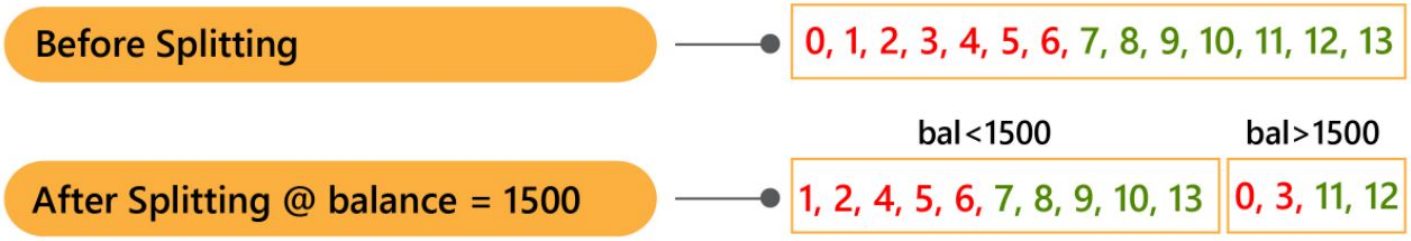
So for age<39, p(success)=0 (none has term deposit). Hence, gini(age < 39) = $1-(p^2+(1-p)^2) = 1-(0+1) = 0$

For age>39,
$p(success) = \frac{\text{number of persons having term deposit}}{\text{number of persons in the subset}} = \frac{7}{10}$. Hence we have

$$gini(age > 39) = 1-(0.7^2+0.3^2) = 0.42$$

$$gini(age = 39) = \frac{4}{14}gini(age < 39) + \frac{10}{14}gini(age > 39) = \frac{4}{14}0 + \frac{10}{14}0.42 = 0.3$$

Now, let's check for split on bal=1500.



So for bal<1500, p(success)=5/10=0.5. Hence, gini(bal<1500) = $1-(0.5^2+(0.5)^2) = 0.5$ Similarly for bal>1500,

$p(success) = \frac{\text{number of persons having term deposit}}{\text{number of persons in the subset}} = \frac{2}{4}$. Hence we have

$$gini(bal > 1500) = 1-(0.5^2+0.5^2) = 0.5$$

$$gini(bal = 1500) = \frac{10}{14}gini(bal < 1500) + \frac{4}{14}gini(bal > 1500) = \frac{10}{14}0.5 + \frac{4}{14}0.5 = 0.5$$

Since gini(age=39) < gini(bal=1500), age=39 is a better split.

The splitting of the data is same as shown in previous topic. Let us look at the calculation of gini index hands on for age=39.

```
try:
    pos_cc_left_split = left_split_dist[1] # positive class counts
except KeyError:
    pos_cc_left_split = 0

try:
    neg_cc_left_split = left_split_dist[0] # negative class counts
except KeyError:
    neg_cc_left_split = 0

gini_age_left_split = 1 - ((pos_cc_left_split/num_left_split)**2 +
(neg_cc_left_split/num_left_split)**2)

try:
    pos_cc_right_split = right_split_dist[1] # positive class counts
except KeyError:
    pos_cc_right_split = 0

try:
    neg_cc_right_split = right_split_dist[0] # negative class counts
except KeyError:
    neg_cc_right_split = 0

gini_age_right_split = 1 - ((pos_cc_right_split/num_right_split)**2 +
(neg_cc_right_split/num_right_split)**2)

gini_age_39 = (num_left_split/num_total_data)*gini_age_left_split + \
(num_right_split/num_total_data)*gini_age_right_split

print("The gini value of data after splitting with age=39 is %.2f.\n" % gini_age_39)
```

The output is:

```
The gini value of data after splitting with age=39 is 0.30.
```

For balance=1500, the gini calculation code is very similar (try it as a homework).

We will look at how to construct a decision tree for categorical values in the next topic. We have looked at the implementation on numerical features. But what if the feature is categorical/nominal? Like in our case, the `housing_cat` feature.

Using nominal values for splitting the decision tree

We have seen till now, how we can split the decision tree for continuous values. Now we will have a look at the split on the nominal values. Let us take a look at our toy data first…

	age	balance	campaign	housing_cat
0	59	2343	1	1
1	56	45	1	0
2	41	1270	1	1
3	55	2476	1	1
4	54	184	2	0
5	42	0	2	1
6	56	830	1	1
7	57	604	1	0
8	45	0	1	1
9	48	238	2	1
10	34	673	1	1
11	37	7944	1	0
12	32	3696	3	0
13	35	347	1	0

We have one categorical/nominal attribute - `housing_cat`. Let us look at how to split at this attribute. Categorical variables are fairly simple to handle. There are limited number of values the attribute can take, and we split the dataset on the basis of that. Now let's split the dataset on the value of `housing_cat` attribute.

```
# getting the split of the logical array
housing_split_yes = bank_small['housing_cat']==1
housing_split_no = ~housing_split_yes

# get the class labels for the data split on housing_cat
housing_split_yes_labels = bank_small_labels[housing_split_yes]
housing_split_no_labels = bank_small_labels[housing_split_no]
```

Once the split is obtained, the rest is same as how we proceeded with continuous values. The difference is - once we use a categorical attribute for split, we cannot use the same variable again down the tree, while for numerical attributes we can reuse it for splitting the data in a subtree (though with a different threshold!).

Q. **In Decision Tree, by comparing the impurity across all possible splits in all possible Predictors, the next split is choosen. Impurity in Decision Tree is generally measured using what?**

Ans: Entropy, Ginni-Index

Decision tree visualization through sklearn

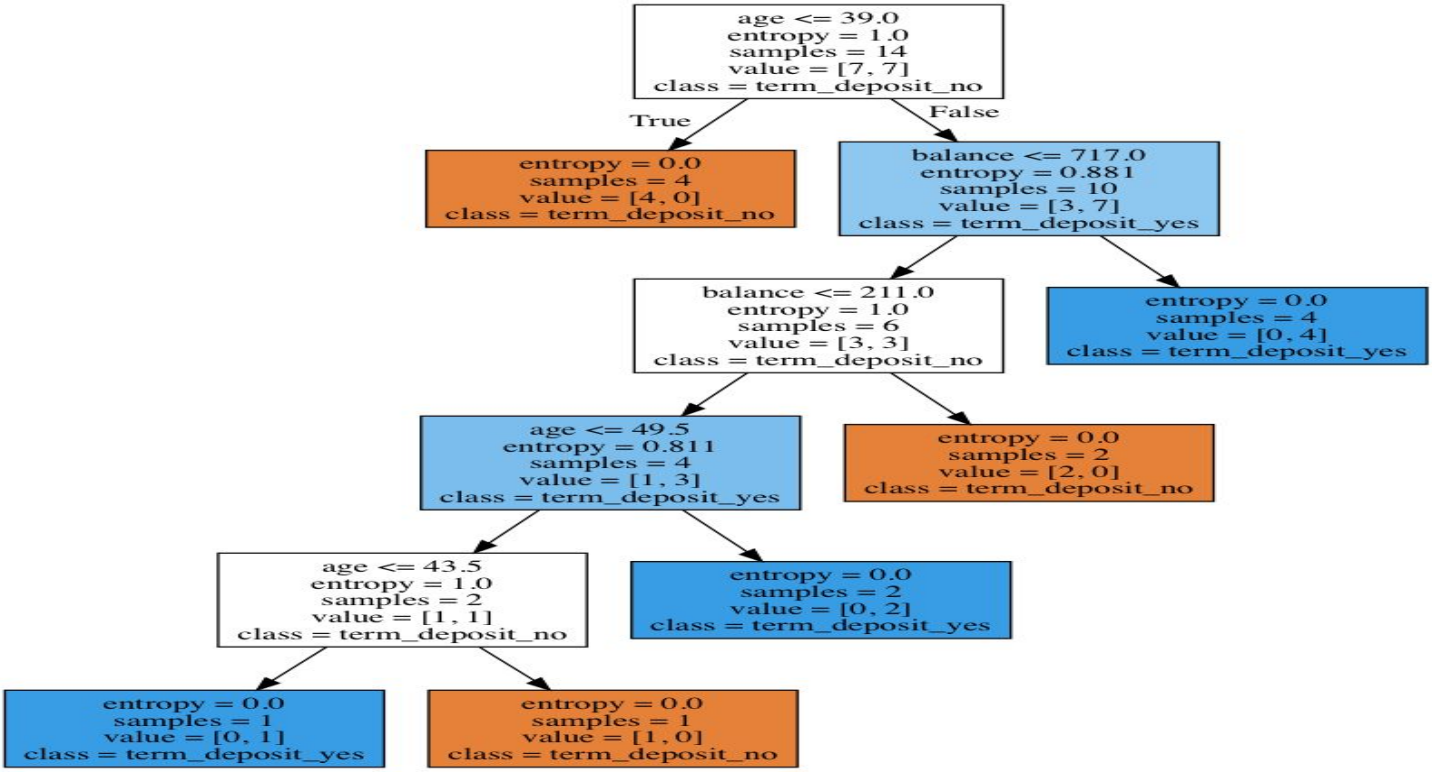
Now, that we know how to calculate information gain/gini on a given data to choose the splitting attribute, it is time to build the decision tree and visualize it through `sklearn`.

```
# Create decision tree classifer object
dt1 = tree.DecisionTreeClassifier(criterion='entropy')

# Train the model
dt1.fit(bank_small, bank_small_labels)

# Create DOT data for visualizing the tree
dot_data = tree.export_graphviz(dt1, out_file=None,
                                feature_names=bank_small.columns, filled = True,
                                class_names=['term_deposit_no','term_deposit_yes'])

# Draw graph
graph_small = pydotplus.graph_from_dot_data(dot_data)

# Show graph
Image(graph_small.create_png())
```

For the purpose of just running the decision tree and getting results, `fit()` and `predict()`/`score()` are sufficient. But, here we are also attempting to visualize the decision tree also. For that purpose, the lines of code from 7-16 will help.

To know more about `tree.export_graphviz` please refer to the documentation here.

Let's look at how our decision tree on the toy data looks like. We will learn more on how to interpret the data in the next topic.



Learning a decision tree on given data

In this task, you will generate a decision tree on the data and also visualize it.

Intructions

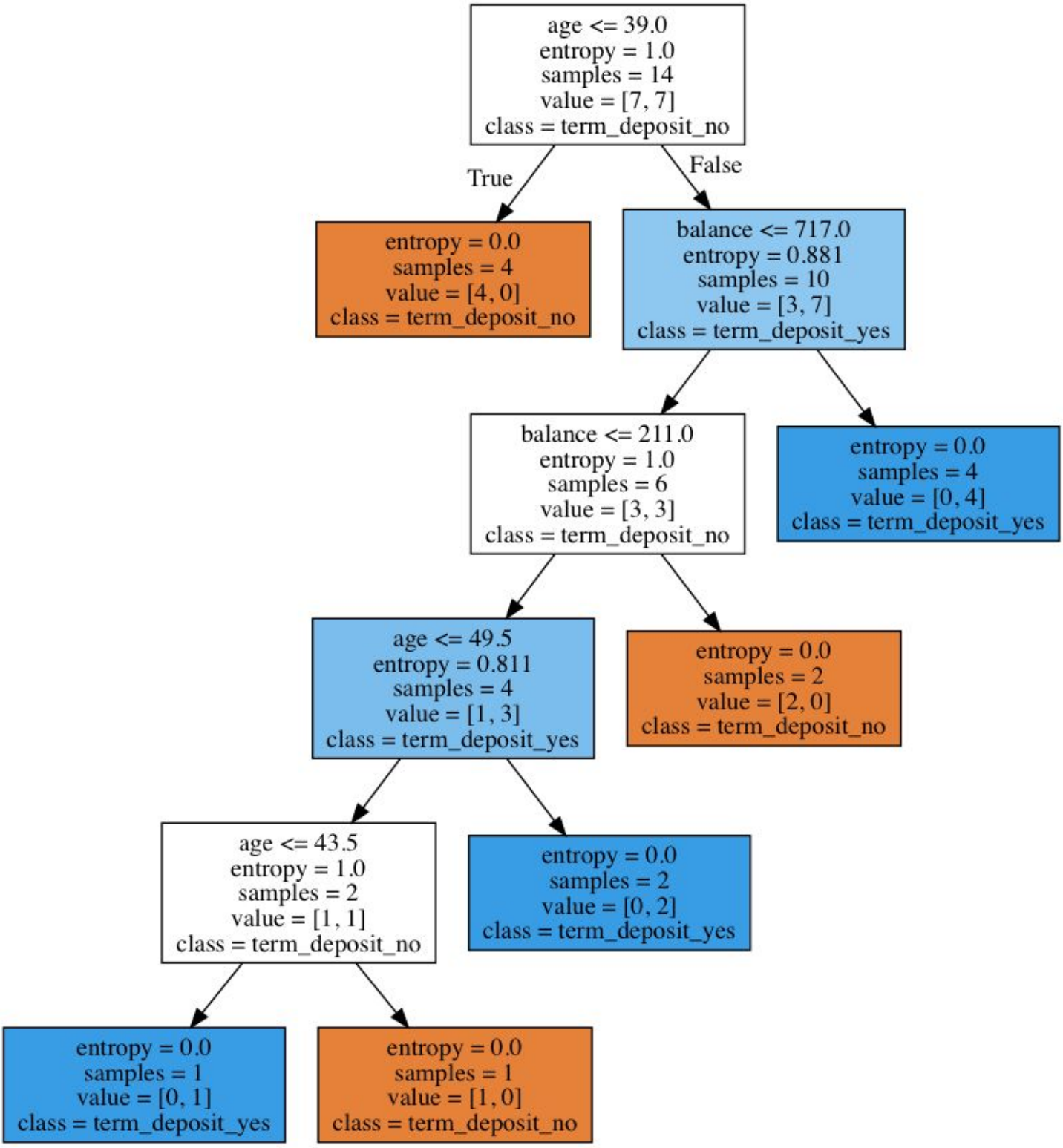
- Use the `train_test_split()` module to split the data and labels into train and test sets. Use 80% of the data for training and 20% of the data for testing. Save them as `data_train`(training features), `data_test` (test features), `label_train` (training labels) and `label_test` (test labels). Use random_state = 6.
- Use the `tree.DecisionTreeClassifier()` to initialize a decision tree object as `dt2`. Pass the parameter `max_depth=2` while initializing the object. (You can also pass the parameter `criterion=entropy`/`criterion=gini`. It is `criterion=gini` by default).
- Use the `fit()` method of the decision tree object to train the decision tree on the training data i.e. on `data_train` and `label_train`
- Use the `tree.export_graphviz()` module to create the graphic visualization object of the learned tree '`dt2`' .Input the parameters `dt2, out_file=None, feature_names=bank_full.columns, filled = True, class_names=['term_deposit_yes','term_deposit_no']` and save it as `dot_data`
- Use the `pydotplus.graph_from_dot_data()` module to draw the graph. Save it as `graph_big`
- To display the graph we save the above created image into a file and display it using matplotlib. We have given you the code for the same.

```
# import packages

import numpy as np
import matplotlib.pyplot as plt
from io import StringIO
from sklearn.tree import export_graphviz
from sklearn.model_selection import train_test_split
from sklearn import tree
from sklearn import metrics
from IPython.display import Image
import pydotplus

# solution begins
# splitting the data

data_train, data_test, label_train, label_test =
train_test_split(bank_full,bank_full_labels,test_size=0.2,random_state=6)

# Create decision tree classifer object use tree.DecisionTreeClassifier(max_depth=2)
dt2 = tree.DecisionTreeClassifier(criterion='entropy', max_depth=2)

# Train the model
dt2.fit(data_train, label_train)

# Create DOT data
```

```
dot_data = tree.export_graphviz(dt2, out_file=None,
                                feature_names=bank_full.columns, filled = True,
                                class_names=['term_deposit_yes','term_deposit_no'])

# Draw graph
graph_big = pydotplus.graph_from_dot_data(dot_data)

# show graph - do not delete/modify the code below this line

img_path = user_data_dir+'/file.png'
graph_big.write_png(img_path)

plt.figure(figsize=(20,15))
plt.imshow(plt.imread(img_path))
plt.axis('off')
plt.show()
#solution ends
```

Interpreting decision tree built in sklearn

Let us again look at the tree created on the toy dataset. Now, that we have built sufficient intuition on how the tree is built, how to choose the attribute on which to split the tree - let us understand the tree better and learn how to use the tree :) .

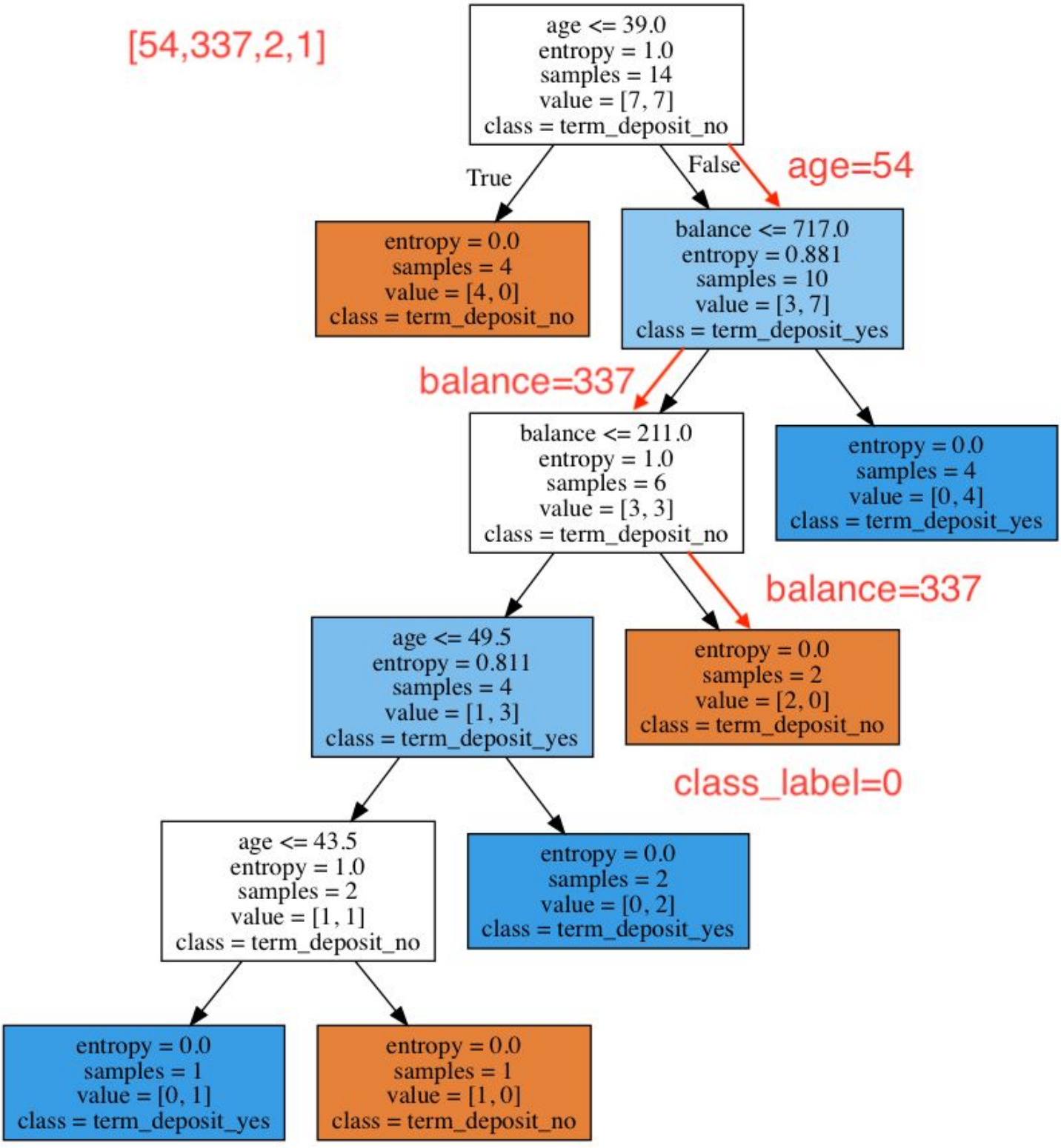


At each level, we have the following information :-

1. the condition of the split at the particular node.
2. entropy - the entropy of the data before the split
3. samples - the number of samples of data before the split (at that node)
4. value - the classwise distribution of the samples
5. class - the class decision for any element at that node. (if you were to assign a class label to a data point at that node, what label to assign.)

Predicting on new data

Let's run through an unseen new data point through a decision tree. We will visually see how we decide the class membership of a sample test point - [54,337,2,1].



Explaination

The following steps explain how the decision tree makes predictions:

- Primary splitting feature is `age` and its threshold is `39`. The age is `54` (greater than threshold) and so we go to the next node
- This time the splitting feature is `balance` with threshold `717`. Our new point has `balance` equal to `337` and so we go to the left node below
- Again the splitting feature is `balance` with threshold `211`. We know that data point has `337` as `balance` which leads to a node with pure split (right node in orange). It has class label of `0` which is our prediction.

Feature Importance in Decision Tree

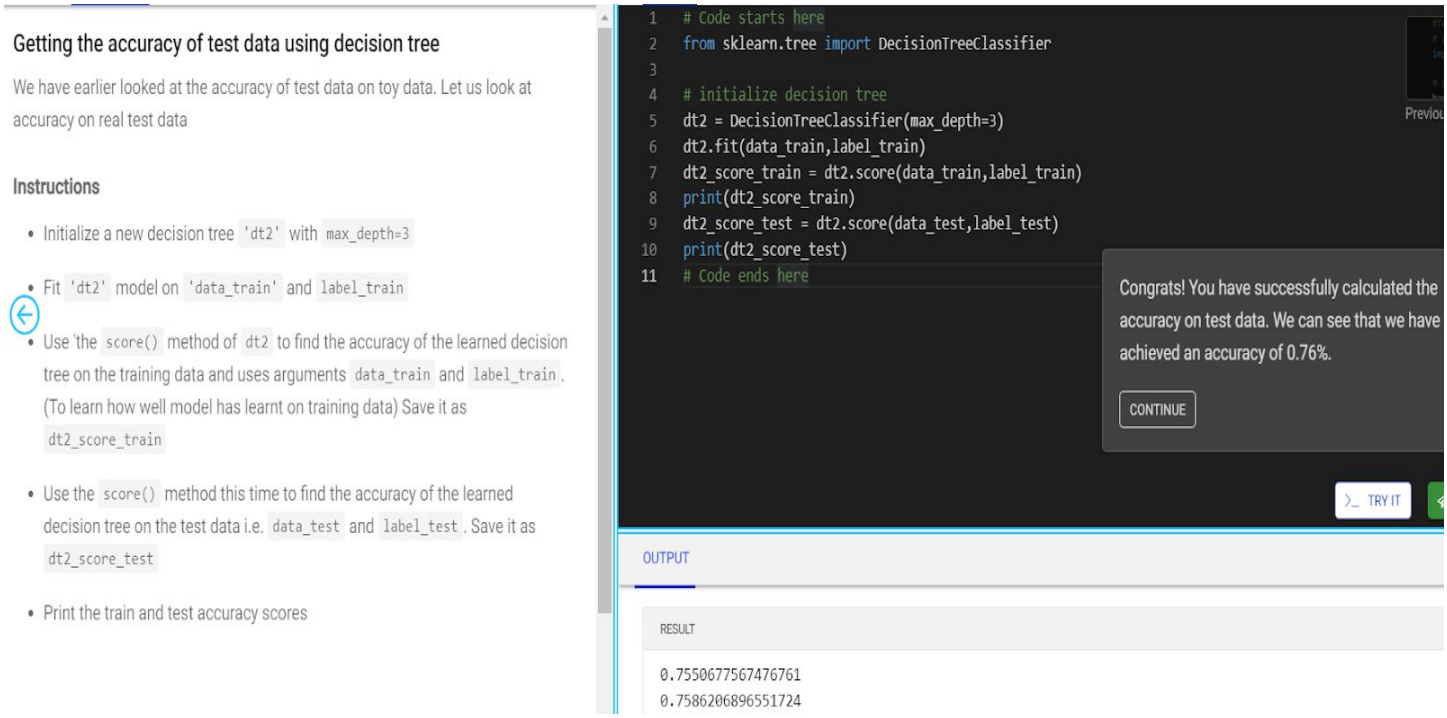
One advantage of decision tree is that you can look at the importance of features. Lets look how to achieve it with python's scikit-learn:

```
# import packages
from sklearn.tree import DecisionTreeClassifier

# initialize decision tree
tree = DecisionTreeClassifier(criterion='entropy', random_state=1)

# fit on data
tree.fit(toy_data, toy_label)

# feature importances
print("Feature importances are:", dict(zip(toy_data.columns,
np.round(tree.feature_importances_,3))))
```

The output is:

```
Feature importances are: {'age': 0.459, 'balance': 0.398, 'campaign': 0.143, 'housing_cat':
0.0}
```

Based on the output, we can infer that `age` is the most important feature and `housing_cat` is the least.



Indicators of overfitting

Interviewer: What's your biggest strength?

Me: I'm an expert in machine learning.

Inteviewer: What's 6 + 10?

Me: Zero.

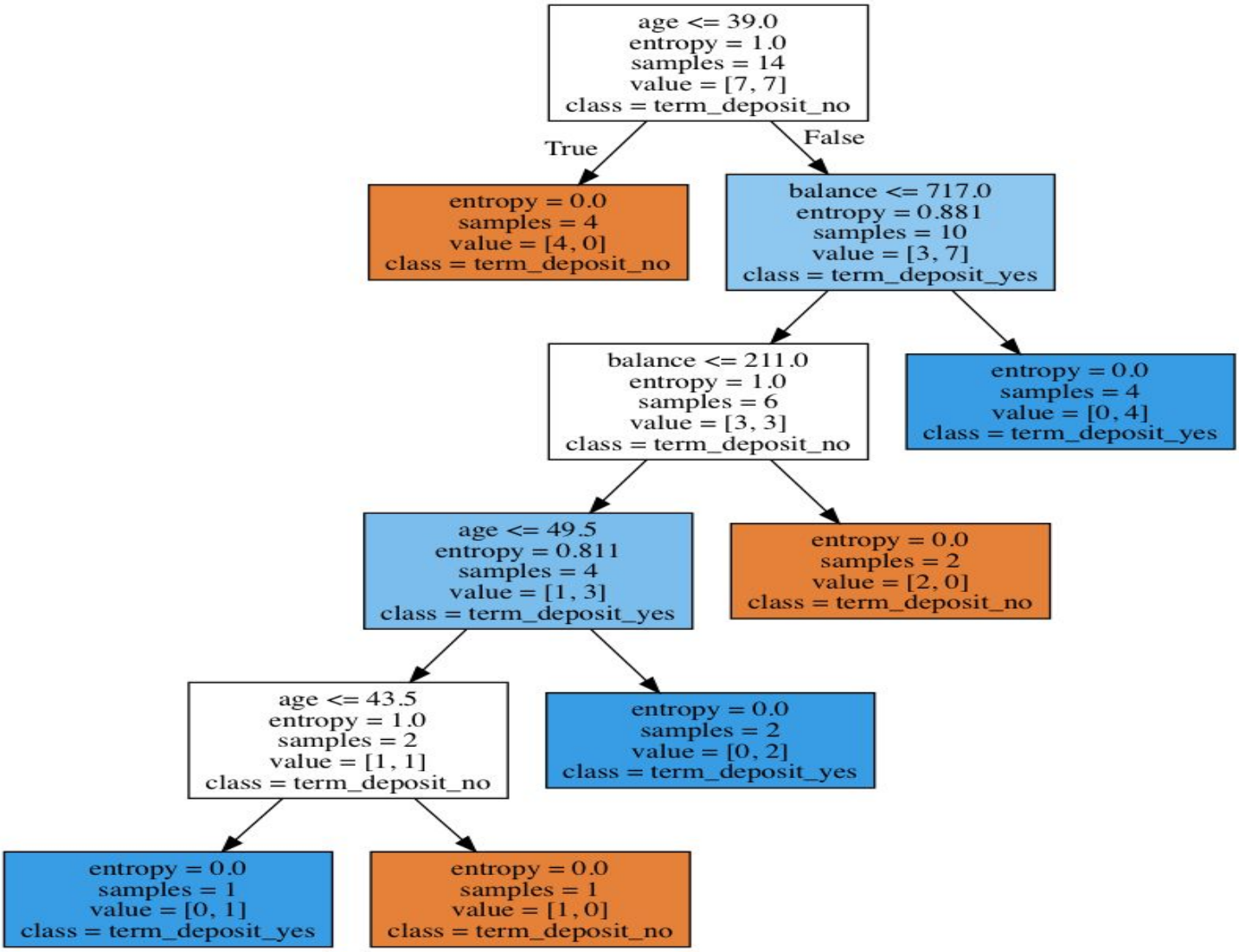
Interviewer: Nowhere near, it's 16.

Me: It's 16.

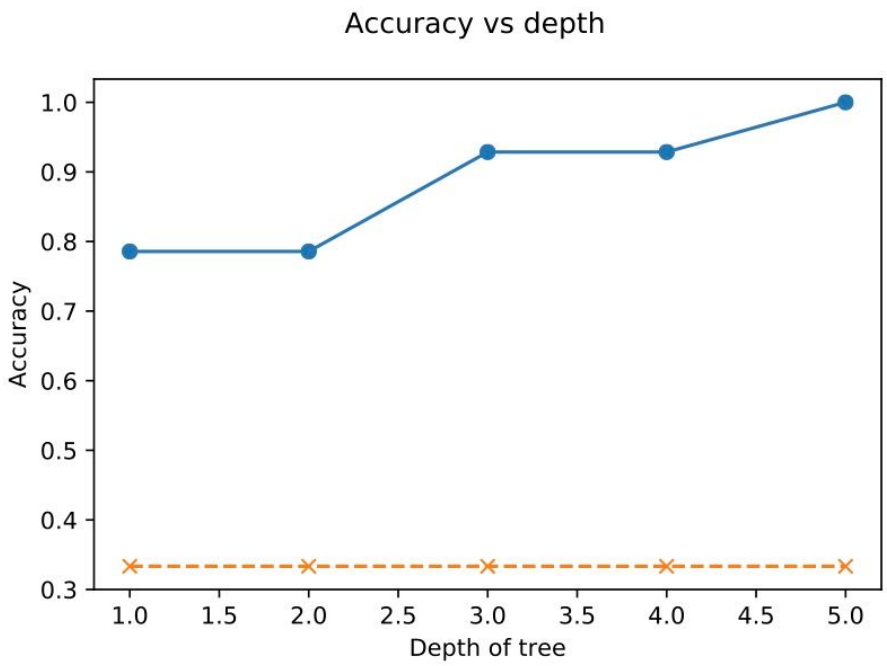
Interviewer: Ok... What's 10 + 20?

Me: It's 16.

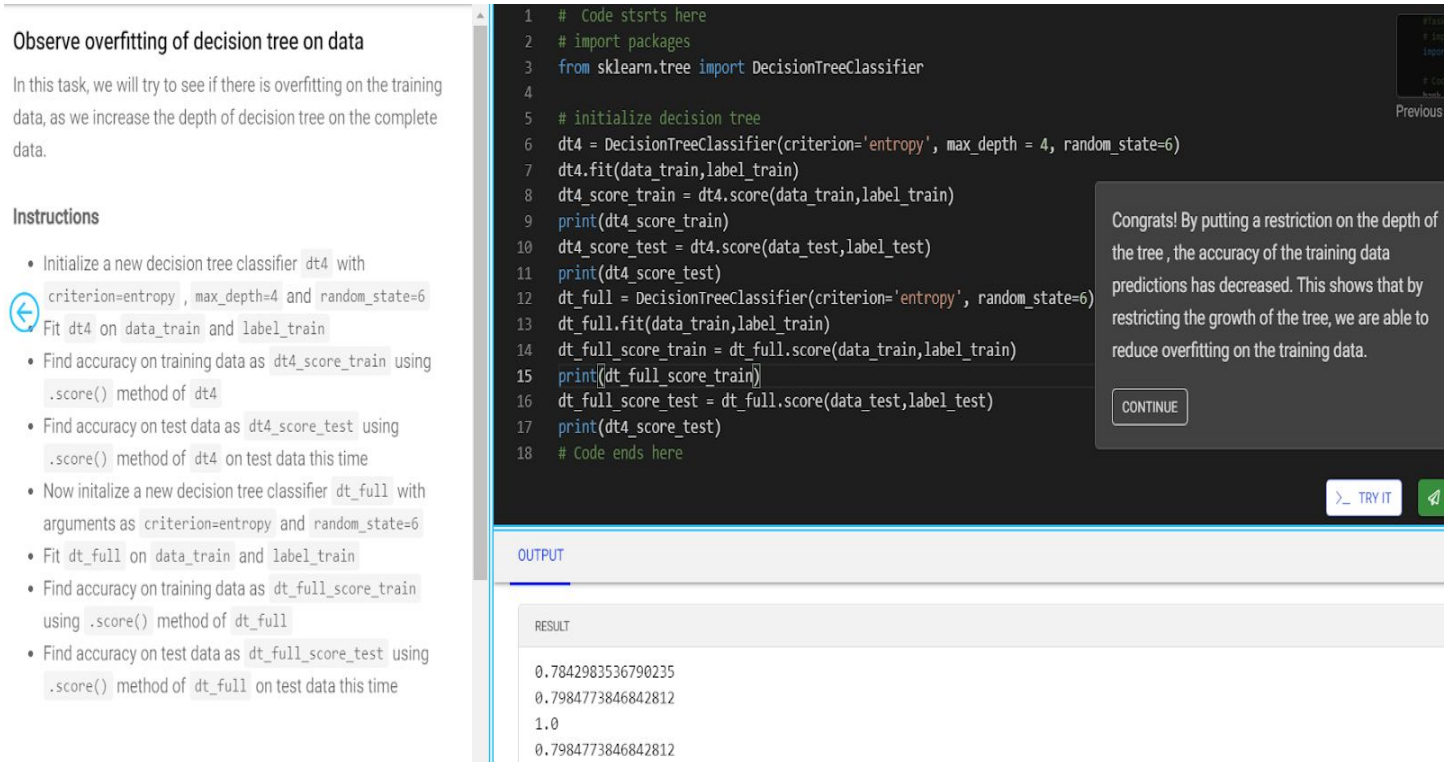
We have seen in the previous chapter that we have overfitted our data. We had looked at regularization in our previous concepts. What could be the ideal regularizer? What could be the indicator of complexity in the decision tree? Let us go back to the toy data tree and explore.



Do you remember we had a parameter for `max_depth` in the decision_tree creation. This parameter restricts the depth of a decision tree and does not allow splitting beyond the mentioned depth. Let us play around with it and see what happens. For the original tree, the depth is 5. Let's keep reducing the depth and see what happens to the train and test accuracy on toy data.



The blue line corresponds to training data accuracy and the orange line corresponds to test data accuracy with increasing depth of the decision tree. The training accuracy of data is around 0.9 at depth 3 and 4 which is pretty decent, while the test accuracy remains unchanged. A jump from 0.7 to 0.9 from depth 2 to 3 is a good improvement, but 0.9 to 1 for a small dataset might be overfitting. So little increase in the training accuracy with increase in depth is a good indicator of overfitting. Restricting the depth of the tree is one way to regularize the decision tree.



Pruning the tree

If you have done the tasks in the previous topic (if not yet done, highly recommend you do so!!!), you could see the principle of overfitting on the complete data. You had increase in training accuracy but no increase in accuracy on test data. We will now look at pruning strategies - i.e try to limit the tree from growing and overfitting onto the data.

Some of the pruning strategies are as follows:-

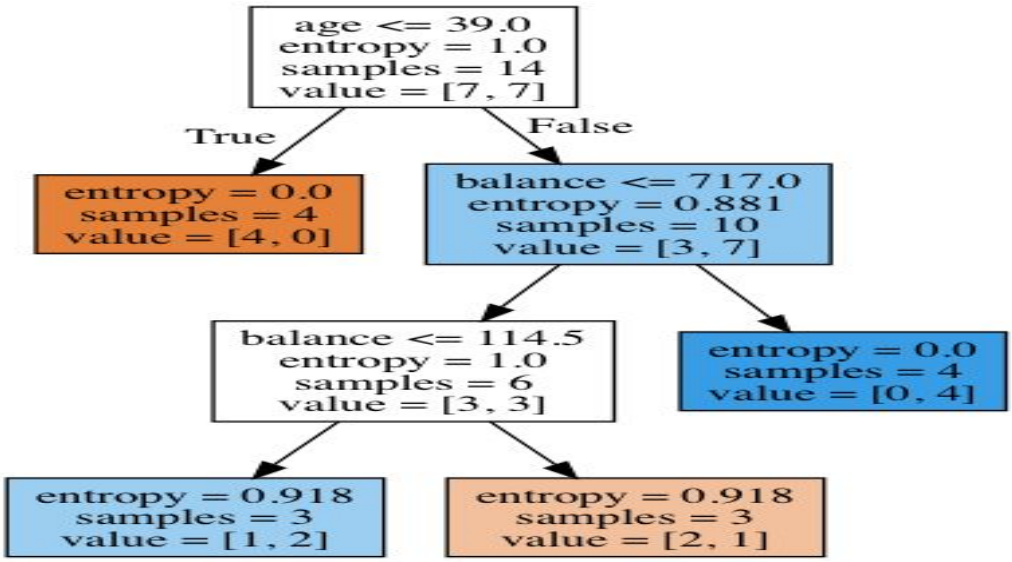
- limit the depth of a tree `max_depth` (we have seen this in the previous topic)
- put a condition on the minimum number of samples in a leaf node `min_samples_leaf`. If the number of datapoints in a node are too small, and there is still some impurity left, that is an indicator that overfitting is happening.

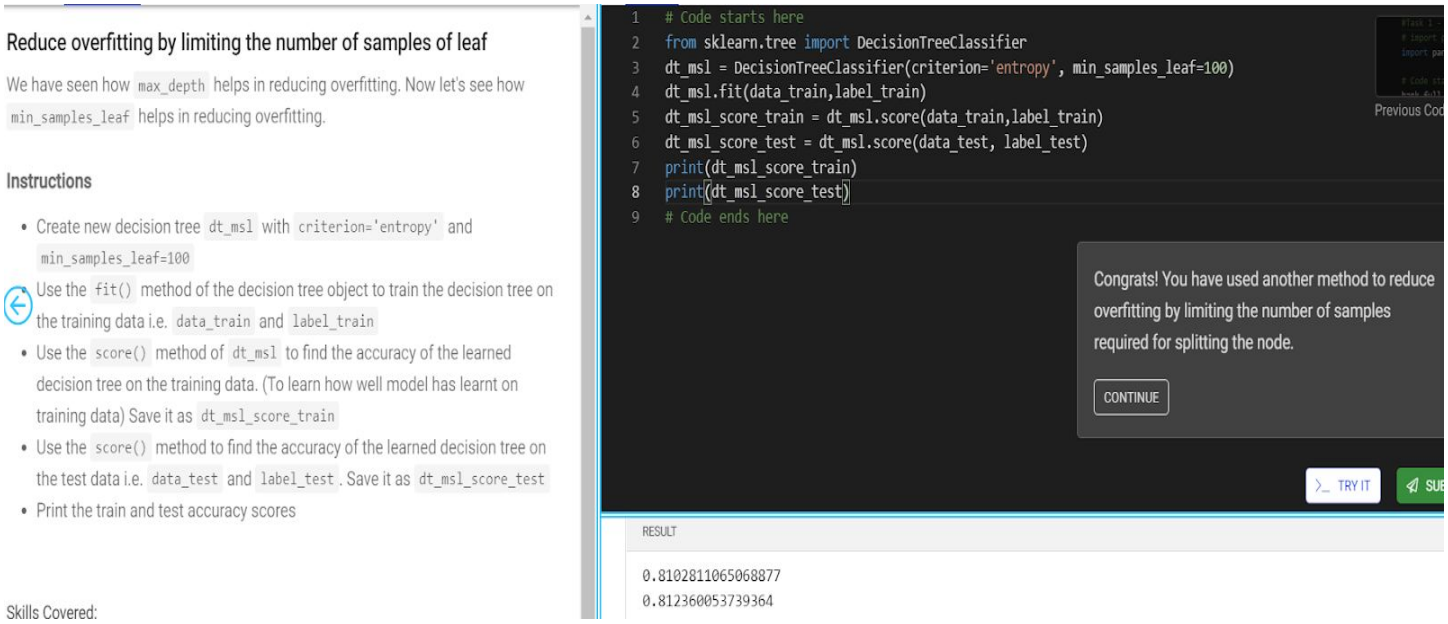
Some other parameters are present which can be used to control how the tree is built. If interested, you can browse the documentation to know more about these other features.

We will look at the code for pruning the tree by putting the limit on the number of samples in the leaf.

```
# Create decision tree classifer object
dt1 = tree.DecisionTreeClassifier(criterion='entropy', min_samples_leaf=3)

# Train the model
dt1.fit(bank_small, bank_small_labels)

# Create DOT data for visualizing the tree
dot_data = tree.export_graphviz(dt1, out_file=None,
                                feature_names=bank_small.columns, filled = True,
                                class_names=['term_deposit_no','term_deposit_yes'])
# Draw graph
graph_small = pydotplus.graph_from_dot_data(dot_data)

# Show graph
Image(graph_small.create_png())
```

Next, we will visualize the tree.





QUESTIONS	YOUR ANSWER	CORRECT ANSWER
1. How we can avoid the overfitting in Decision Tree?	Both the above methods	Both the above methods

Explanation:

In decision trees, over-fitting occurs when the tree is designed so as to perfectly fit all samples in the training data set. Thus it ends up with branches with strict rules of sparse data.

Both Pruning and CHAID modify the tree structure and hence potentially help resolve overfitting issue.

2.	Pre-pruning the decision tree may result in _____ .	Underfitting	Underfitting

Explanation:

An alternative method to prevent overfitting is to try and stop the tree-building process early, before it produces leaves with very small samples. This heuristic is known as pre-pruning decision trees. At each stage of splitting the tree,the cross-validation error is checked. If the error does not decrease significantly enough then we stop.

Early stopping may `underfit` by stopping too early.

Advantages and disadvantages of decision trees

Let's look at various advantages and disadvantages of decision trees.

Advantages of Decision Trees

1) Easy to Understand:

- Decision tree output is very easy to understand even for people from non-analytical background.
- It does not require any statistical knowledge to read and interpret them.
- Its graphical representation is very intuitive and users can easily relate their hypothesis.

2) Useful in Data exploration:

- Decision tree is one of the fastest way to identify most significant variables and relation between two or more variables. - With the help of decision trees, we can create new variables / features that has better power to predict target variable. - It can also be used in data exploration stage. For example, we are working on a problem where we have information available in hundreds of variables, there decision tree will help to identify most significant variable.

3) Less data cleaning required:

- It requires less data cleaning compared to some other modeling techniques.
- It is not influenced by outliers and missing values to a fair degree.

4) Data type is not a constraint:

- It can handle both numerical and categorical variables.

5) Non Parametric Method:

- Decision tree is considered to be a non-parametric method.
- This means that decision trees have no assumptions about the space distribution and the classifier structure.

Drawbacks of decision trees

- Decision-tree learners can create over-complex trees that do not generalise the data well. This is called overfitting.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

Applications and Summary

Decision Tree Use Cases

Some uses of decision trees are:

- Building knowledge management platforms for customer service that improve first call resolution, average handling time, and customer satisfaction rates
- In finance, forecasting future outcomes and assigning probabilities to those outcomes
- Binomial option pricing predictions and real option analysis
- Customer’s willingness to purchase a given product in a given setting, i.e. offline and online both
- Product planning; for example, Gerber Products, Inc. used decision trees to decide whether to continue planning PVC for manufacturing toys or not

- General business decision-making
- Loan approval

Summary

- A decision tree is inferred by growing it top-down, breadth-first, greedily based on information gain.
- A shorter tree with high-information gain attributes close to the root is preferred by the method.
- Overfitting is a problem for decision tree; can be caused by noise in the training data, or by small training sets.
- A number of techniques exist to avoid overfitting, like early-stopping criterion while building the true or pruning after building the tree.

To use or not to use a decision tree

When to use a decision tree

- Decision tree are very easy to understand and every result provided is interpretable. Whenever for the task, the end user needs justification as to why a label has been given, decision tree is the best choice.

When not to use a decision tree

- Decision trees tend to overfit on complex data with lots of attributes.

Real Life Use Case

Business Objective:

An IT start-up involved in developing artificial intelligence solutions wanted to use machine learning to automate it's hiring procedure for selecting candidates. As per the current procedure, the hiring manager would manually go through all the applications and then short-list the appropriate candidates for the next interview round. This process required a lot of manual effort, was time consuming and was not the preferred method going forward as the number of applicants grew exponentially.

Solution Methodology:

Any candidate wishing to apply for a position at the firm would have to fill in his/her details on the company's application form on any of the various application portals and these details would then be stored on the company's database. These details included various categorical and continuous attributes of the candidates like

- highest level of education (Graduate, Post-Graduate, Ph.D)
- ranking of the institute (Tier I, Tier II, Tier III)
- number of internships secured (greater than or equal to zero)

- number of research papers published (greater than or equal to zero)
- current employment status (fresher, employed or unemployed)
- number of years of experience (0-2 years, 2-5 years, 5-10 years, >10 years)
- previous employers (name of the employers)
- current notice period (Immediate joining, 0-15 days, 15-30 days, 1 month)
- current salary (0-3LPA, 3-8LPA, >8LPA)

The candidates whose applications had been reviewed were assigned a new attribute indicating whether they were selected for the next round or not. Thus the company had a dataset consisting of the attributes of candidates as feature variables and a target feature indicating their selection status.

A decision tree algorithm was implemented on this data to predict the outcome of selection as a function of the given input features. Based on the training data, the entropy and information gain was calculated to decide the splitting feature for establishing the decision tree and growing it further.

The features like current salary, previous employers, number of years of experience, current notice period contributed majorly in recruitment analysis and features like level of education, number of research papers, ranking of institute enabled the recruiters to determine the how well versed the candidates were in their field.

Business Impact:

A pilot version of this model was implemented on a limited new dataset of candidates. The model entirely performed the job of screening through the list of candidates, learning their feature attributes, segregating the data as per these features and then identified whether they should be selected for the next round or not. This was drastically able to reduce the time and effort that had to be otherwise invested in the process and enabled the hiring manager to only focus only on promising candidates as the model automatically filtered out the rest of the candidates.

Going further, the next objective was to build a resume screening algorithm using NLP that could further rank the resume of candidates selected for the next round and further provide a filter to select only those candidates having a strong resume.

Target variable distribution

Now let's look at the distribution of `'paid.back.loan'`

- Save the value counts of `paid.back.loan` (stored in y_train) in a variable called `'fully_paid'` using `"value_counts()"`.
- Plot a bar graph of `'fully_paid'`.

Skills Covered:

Data Wrangling

Reference Solution

```
#Importing header files
import matplotlib.pyplot as plt
#Code starts here
#Storing value counts of target variable in 'fully_pa
fully_paid=y_train.value_counts()
#Plotting bar plot
plt.bar(fully_paid.index, fully_paid)
plt.show()
```

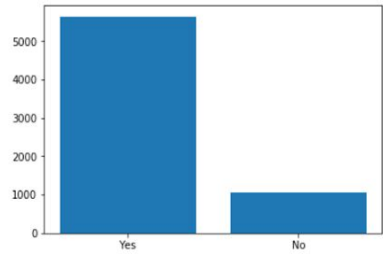
We can see that 5639 people have paid back loan while 1065 people not paid back the loan.

CONTINUE

TRY IT

OUTPUT

RESULT



Feature Split

In this task along with one step of feature engineering, we will split the features based on the data type for visualization purposes.

- From the column `int.rate` of `'X_train'`, remove the `%` character and convert the column into float.
- After that divide the values of `int.rate` with 100 and store the result back to the column `'int.rate'`
- Do the same with the `int.rate` of `'X_test'`
- Create a subset of only numerical columns of `'X_train'` using `"select_dtypes()"` and save it to a variable called `'num_df'`
- Create a subset of only categorical(object) columns of `'X_train'` using `"select_dtypes()"` and save it to a variable called `'cat_df'`

```
#Importing header files
import numpy as np
from sklearn.preprocessing import LabelEncoder
# Code starts here
X_train['int.rate'] = (X_train['int.rate'].str.replace("%","").astype(float))/100
X_test['int.rate'] = (X_test['int.rate'].str.replace("%","").astype(float))/100
num_df = X_train.select_dtypes(include='number')
print(num_df.shape)
cat_df = X_train.select_dtypes(include='object')
print(cat_df.shape)
# Code ends here
```

OUTPUT

RESULT

```
(6704, 9)
(6704, 4)
{
    "name": "stderr",
    "text": "/opt/greyatom/kernel-gateway/runtime-environments/lib/python3.6/site-pa
}
```

Numerical Features Visualisation

Let's plot and see the relation of numerical features with the target variable.

- Create a list called `cols` which is a list of all the column names of `'num_df'`.
- Create subplot with `(nrows = 9 , ncols = 1)` and store it in variable's `fig ,axes`
- Create `for` loop to iterate through rows.
- Inside the loop, using seaborn, plot the `boxplot` where `x=y_train`, `y=num_df[cols[i]]` and `ax=axes[i]`.

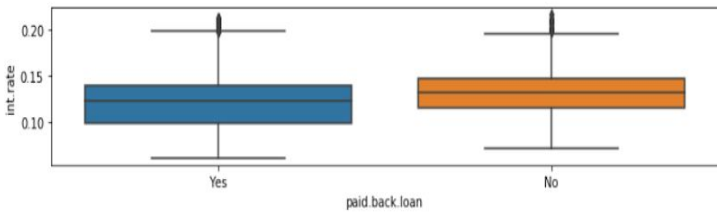
Skills Covered:

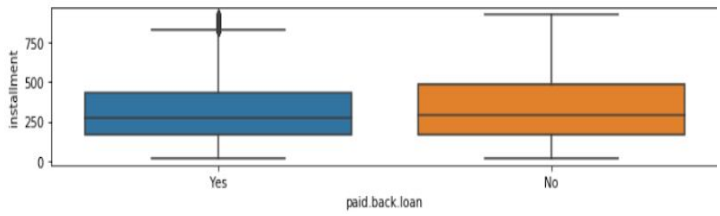
Visualization

Reference Solution

```
#Importing header files
import seaborn as sns
cols = num_df.columns
# Code starts here
fig, axes = plt.subplots(nrows=9,ncols=1,figsize=(10,20))
for i in range(0,9):
    sns.boxplot(x = y_train, y = num_df[cols[i]],ax = axes[i])
    fig.tight_layout()
# Code ends here
```

TRY





Categorical Features Visualisation

Let's plot and see the relation of categorical features with the target variable.

- Create a list called `cols` which is a list of all the column names of `'cat_df'`.
- Create subplot with `(nrows = 2 , ncols = 2)` and store it in variable's `fig ,axes`
- Create two `for` loops to access rows and columns
- Using seaborn plot the `countplot` where `x=X_train[cols[i*2+j]]`, `hue=y_train` and `ax=axes[i,j]`

Skills Covered:

Visualization

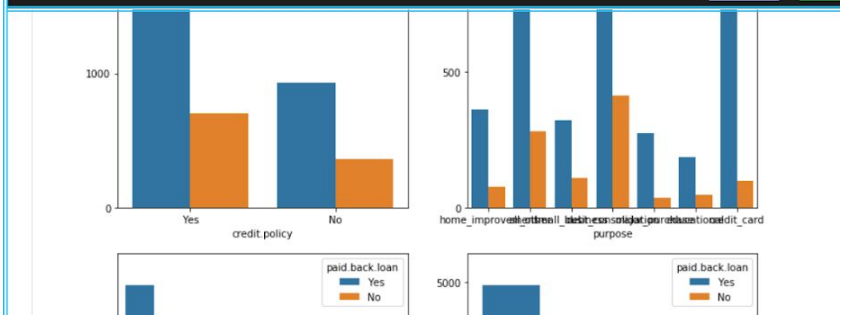
Reference Solution

```
# Code starts here
cols = cat_df.columns
fig, axes = plt.subplots(2,2,figsize=(10,20))
for i in range(0,2):
    for j in range(0,2):
        sns.countplot(x = X_train[cols[i*2+j]], hue =
        fig.tight_layout()
# Code ends here
```

Congrats! You have successfully plotted (bar plot) against the target variable. We can see that the major reason that stands common for the majority of customers who have applied for a loan is debt_consolidation which means taking one loan to payoff there other loans.

CONTINUE

TRY IT



Model Building

Let's finally get to the part of decision tree modeling.

- Iterate a for loop over categorical columns `cat_df`.
- Fill the `X_train` null values with `NA`.
- Create a `LabelEncoder` object of `sklearn` and store it in a variable called `'le'`
- Use `le` to fit and transform all categorical columns `cat_df` of `'X_train'`
- Fill the `X_test` null values with `NA`.
- Transform all categorical columns of `'X_test'` using `le`.
- Replace `No` with `0` and `Yes` with `1` in both `'y_train'` and `'y_test'`
- Create a `DecisionTreeClassifier` object of `sklearn` with `random_state=0` and save it in a variable called `'model'`.
- Fit `'X_train'` and `'y_train'` with `'model'`
- Find the accuracy score of `model` with `'X_test'` and `'y_test'` and store it in a variable called `acc`

```
from sklearn.tree import DecisionTreeClassifier
#Looping through categorical columns
for col in cat_df.columns:
    #Filling null values with 'NA'
    X_train[col].fillna('NA',inplace=True)
    #Initalising a label encoder object
    le=LabelEncoder()
    #Fitting and transforming the column in X_train with 'le'
    X_train[col]=le.fit_transform(X_train[col])
    #Filling null values with 'NA'
    X_test[col].fillna('NA',inplace=True)
    #Fitting the column in X_test with 'le'
    X_test[col]=le.transform(X_test[col])
# Replacing the values of y_train
y_train.replace({'No':0,'Yes':1},inplace=True)
# Replacing the values of y_test
y_test.replace({'No':0,'Yes':1},inplace=True)
#Initialising 'Decision Tree' model
model=DecisionTreeClassifier(random_state=0)
#Training the 'Decision Tree' model
```

Previous C

Congrats! You have successfully trained data with Decision Tree and found out its accuracy to be 0.73%.

CONTINUE

TRY IT

RESULT

```
{
    "name": "stderr",
    "text": "/opt/greyatom/kernel-gateway/runtime-environments/lib/python3.6/site-packages/pandas/core/
}
0.7334725121781489
```

CODE :

```
#Importing header files
from sklearn.tree import DecisionTreeClassifier

#Code starts here

#Looping through categorical columns
for col in cat_df.columns:

    #Filling null values with 'NA'
    X_train[col].fillna('NA',inplace=True)

    #Initalising a label encoder object
    le=LabelEncoder()

    #Fitting and transforming the column in X_train with 'le'
    X_train[col]=le.fit_transform(X_train[col])
```

```
    #Filling null values with 'NA'
    X_test[col].fillna('NA',inplace=True)

    #Fitting the column in X_test with 'le'
    X_test[col]=le.transform(X_test[col])

# Replacing the values of y_train
y_train.replace({'No':0,'Yes':1},inplace=True)

# Replacing the values of y_test
y_test.replace({'No':0,'Yes':1},inplace=True)

#Initialising 'Decision Tree' model
model=DecisionTreeClassifier(random_state=0)

#Training the 'Decision Tree' model
model.fit(X_train, y_train)

#Finding the accuracy of 'Decision Tree' model
acc=model.score(X_test, y_test)

#Printing the accuracy
print(acc)

#Code ends here
```

Decision Tree Pruning

Let's see if pruning of decision tree improves its accuracy. We will use grid search to do the optimum pruning

- Parameter grid `param_grid` with the parameters to pass in Grid Search is given
- Create a `DecisionTreeClassifier` object of `sklearn` with `random_state=0` and save it in a variable called `'model_2'`.
- Create a `GridSearchCV` object from `sklearn` object with `estimator=model_2`, `param_grid=parameter_grid` and `cv=5` and save it in a variable called `'p_tree'`
- Fit `'X_train'` and `'y_train'` with `'p_tree'`
- Find the accuracy score of `p_tree` with `'X_test'` and `'y_test'` and store it in a variable called `acc_2`

Skills Covered:

```
#Importing header files
from sklearn.model_selection import GridSearchCV

#Parameter grid
parameter_grid = {'max_depth': np.arange(3,10), 'min_samples_leaf': range(10,50,10)}

# Code starts here
model_2 = DecisionTreeClassifier(random_state=0)
p_tree = GridSearchCV(estimator=model_2,param_grid = parameter_grid, cv=5)
p_tree.fit(X_train, y_train)
acc_2 = p_tree.score(X_test, y_test)
print(acc_2)
# Code ends here
```

OUTPUT

RESULT

```
0.837160751565762
```

Tree visualising

Let's now see how the pruned decision tree looks like using graphviz.

- Create a `export_graphviz` object of `sklearn` with the hyper-parameters `decision_tree=p_tree.best_estimator_`, `out_file=None`, `feature_names=X.columns`, `filled = True` and `class_names=['loan_paid_back_yes','loan_paid_back_no']` and save it in a variable called `'dot_data'`

- To draw the decision-tree(graph), use `"pydotplus.graph_from_dot_data()"` and pass `'dot_data'` as its parameter and save the result in a variable called `'graph_big'`

- In order to display the above created tree we need to save it as an image and display the same using matplotlib. We have provided you with the code for the same.

Do not modify/delete the code given to you for displaying the tree

Skills Covered:

Machine Learning

Previous Code

```
#Importing header files
from io import StringIO
from sklearn.tree import export_graphviz
from sklearn import tree
from sklearn import metrics
from IPython.display import Image
import pydotplus
# Code starts here
dot_data = export_graphviz(decision_tree=p_tree.best_estimator_, out_file=None, feature_names=X.columns,
filled = True, class_names=['loan_paid_back_yes','loan_paid_back_no'])
graph_big = pydotplus.graph_from_dot_data(dot_data)
# show graph - do not delete/modify the code below this line
img_path = user_data_dir+'/file.png'
graph_big.write_png(img_path)
plt.figure(figsize=(20,15))
plt.imshow(plt.imread(img_path))
plt.axis('off')
plt.show()
```

TRY IT SUBMIT