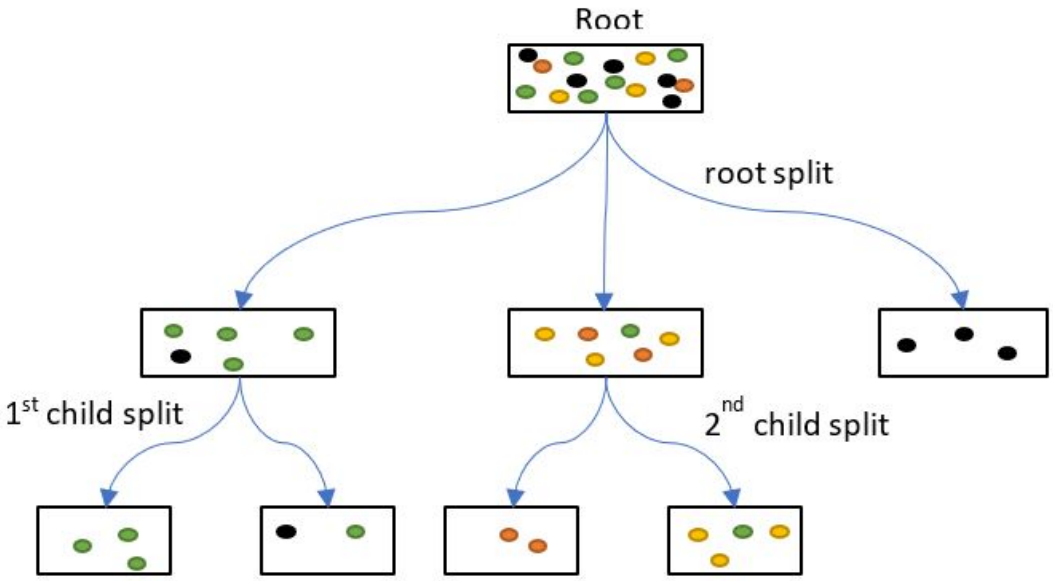
Decision Trees

The trusty decision tool



INTRODUCTION

Decision tree methods are amongst the most widely used methods in the field of decision making. It has a very basic architecture of conditional if-flow control statements that form the rule-based procedure for “filtering” data into segments by asking a particular question about a feature at each node, until reaching the desired end state (often referred to as leaf nodes). The focus of this blog will be the decision tree algorithm known as CART (**c**lassification **a**nd **r**egression **t**rees). This algorithm basically produces only binary trees (each node only has two children) as the question at each node only has a *yes* or *no* answer. Please refer to the Appendix section for general terms used and their meaning.

ADVANTAGES OF DECISION TREES

The decision tree algorithm is a very versatile and intuitive tool in the field of data science that is used for both classification and regression tasks. Inherently this algorithm can also perform multiclass classification without the need for any problem transformation. It requires very little data preprocessing, since this algorithm is scale independent, can work with numerical features and categorical text features and is fairly robust to outliers (the CART implementation does not support categorical features in scikit-learn so here a transformation is required).

The algorithm can fit complex data sets (non-linear relationships) and it scales well with larger datasets. It is also great due to its visual representation as the output it produces is interpretable and also provides insights into the most important features.

The algorithm also plays a crucial role as the basis in more complex ensemble methods like bagging and boosting.

DISADVANTAGES OF DECISION TREES

The algorithm is from the class of greedy algorithms as it does not consider future moves for improving the performance, it only takes the greedy (best approach) at the current split and keeps doing this recursively, which might not converge to the global optimum. As opposed to dynamic programming algorithms that take future moves/combinations into account, but these algorithms border on brute force and typically have a much

larger time complexity.

Decision trees can grow fairly large if not regularized (trimmed) which will result in trees that do not generalize well, hence overfitting the sample data. Luckily we can control how the tree grows during hyperparameter tuning.

The algorithm will suffer in performance when the dimensionality starts becoming too large. Decision trees are also unstable (very sensitive), since a small variance in the data will impact the outcome. It is this variance that we try to reduce via the ensemble models during bagging and boosting. The two most popular being Random Forests and Gradient Boosting. These ensemble models limit the instability by averaging predictions over many trees.

The nature of the decision boundary is discrete in the sense that there are only axis parallel cuts to divide the solution space into rectangular bounding boxes (there are no smooth contours).

REGRESSION AND CLASSIFICATION INTERPRETATION

When the response variable (dependent variable or label) is continuous we have a regression problem and when it is categorical we have a classification problem. We predict the response variable in these two cases very similar to how we would in the KNN algorithm.

REGRESSION

In KNN we would look at the labels of the *x* nearest neighbors and use descriptive statistics like *mean* or *median* over these neighbor label values to determine the predicted value. Similarly in decision trees we would look at the samples within the leaf node and follow the same approach.

CLASSIFICATION

In KNN we would look at the labels of the *x* nearest neighbors and use descriptive statistics like *mode* over these neighbor label values to determine the majority class which is known as *hard voting*. We could also follow a weighted approach via either weighted probabilities or distance to determine the predicted class, which is known as

soft voting. Similarly in decision trees we would look at the samples within the leaf node and follow the same approach.

TRAIN AND RUN TIME COMPLEXITY

Given a training data set with n observations and d features.

Training time ~ O(nlog(n)d)

Space ~ O(#nodes)

Run time ~ O(depth of tree)

THE CART ALGORITHM

As mentioned before this algorithm is a greedy recursive algorithm that splits the data into two subsets using a *single feature* and *threshold* at each node (this is where a question is asked about an observation using **this** particular feature).

When constructing a decision tree the information gain (classification task) or MSE (regression task) is calculated for every independent feature. The feature with the highest information gain (or smallest MSE - depending on the problem being solved) will be selected as the first and most important feature to branch on. This process is repeated recursively until all the features are processed, all pure nodes (has observation pertaining to only one class) have been reached or hyperparameters (regularization) enforced early stopping.

The important part to understand is how the *feature* and *threshold* are chosen at each node?

REGRESSION

Minimize MSE (mean squared error).

Given a dataset X ε {$x_i \varepsilon R^d$, $y_i \varepsilon R$ }, such that i ε {1,2,3.....n} and j ε {1,2,3.....d}

For every feature in X i.e (f_1,f_2,.....f_d) - which are *quantitative features*, and for different thresholds (t εR) create two (binary) mutually exclusive sets satisfying the threshold split such that:

$$S_1 = R_1(f_j, t) = \{x_i \mid f_j \leq t\}$$

$$S_2 = R_2(f_j, t) = \{x_i \mid f_j > t\}$$

If any of the features in (f_1,f_2,....,f_d) are *qualitative* (categorical) then perform one additional step before applying the approach mentioned above. First perform response variable encoding or one-hot encoding to convert the categorical feature into a numeric or *quantitative* feature.

Basically the data is split into two sets trying out various combinations of features and different threshold split values of those features. For each of these combinations we compute the cost function as a weighted average over the two sets.

$$\widehat{y}_{S_1} = \frac{1}{|S_1|} * \sum_{i \, \varepsilon \, S_1} y_i \quad \text{.....[1]}$$

$$\widehat{y}_{S_2} = \frac{1}{|S_2|} * \sum_{i \, \varepsilon \, S_2} y_i \quad \text{.....[2]}$$

$$MSE_{S_1} = \frac{1}{|S_1|} \sum_{i \, \varepsilon \, S_1} \left(\widehat{y}_{S_1} - y_i\right)^2 \quad \text{.....[3]}$$

$$MSE_{S_2} = \frac{1}{|S_2|} \sum_{i \, \varepsilon \, S_2} \left(\widehat{y}_{S_2} - y_i\right)^2 \quad \text{.....[4]}$$

$$Cost\,function = \frac{|S_1|}{|pn|} * MSE_{S_1} + \frac{|S_2|}{|pn|} * MSE_{S_2} \quad \text{.....[5]}$$

*pn = parent node

Equation [1] calculates the predicted value at node S1, which is taken as the mean of the observations in S1.

Equation [2] calculates the predicted value at node S2, which is taken as the mean of the observations in S2.

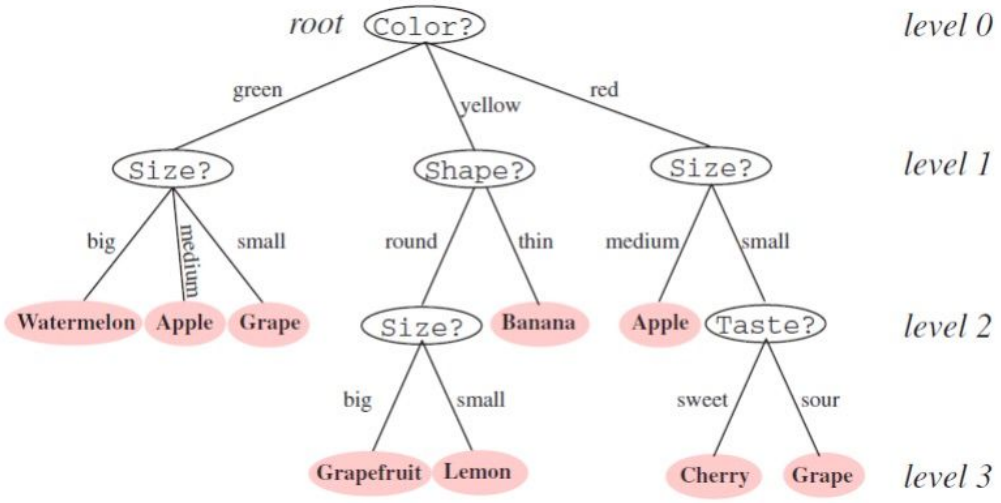
Equation [3] calculates the mean squared error at node S1

Equation [4] calculates the mean squared error at node S2

Equation [5] calculates the weighted cost error

In the end for each feature and threshold combination there will be an associated cost value. This can be represented as a table with three columns (features, thresholds, cost).

The greedy nature of the algorithm will pick the feature and threshold combination that resulted in the current lowest cost. This will then become the most important feature and the first to split the root node into two child nodes. It's important to note that the feature that has been used to split for a particular branch, can be reused to branch observations in another part of the tree, an example of this is shown below [note the use of the *Size* variable].



This procedure is repeated recursively from these newly formed child nodes each one at a time defining a feature and threshold combination. The algorithm will stop when a pure node has been reached or some stopping criteria like the minimum number of observations per node has been reached.

CLASSIFICATION

For classification there are mostly two measures used, they are provided in the table.

Impurity Measure	Equation
Gini impurity (G)	$G_{node} = 1 - \sum_{m=1}^{k} p(y_m)^2$
Entropy (H)	$H_{node} = -\sum_{m=1}^{k} p(y_m) * log(p(y_m))$

Gini impurity is a measure of node purity - a small value indicates that a node predominantly consists of observations from one class (pure node). The maximum value is at 0.5 when there is an equal representation of each class within a node. It will be equal to zero if we have a pure node.

Entropy also takes on a small value when the node is pure or more imbalanced. The maximum value is at 1 when there is an equal representation of each class within a node. It will be equal to zero if we have a pure node.

The value of $p(y_m)$ is the probability of observing class *m* at each node.

Given a dataset X ε $\{x_i \varepsilon R^d , y_i \varepsilon R\}$, such that i ε {1,2,3.....n} and j ε {1,2,3.....d}

For every feature in X i.e (f_1,f_2,.....f_d) - which are *quantitative features*, and for different thresholds (t εR) create two (binary) mutually exclusive sets satisfying the threshold split such that:

$$S_1 = R_1(f_j, t) = \{x_i \mid f_j \leq t\}$$

$$S_2 = R_2(f_j, t) = \{x_i \mid f_j > t\}$$

If any of the features in (f_1,f_2,....,f_d) are *qualitative* (categorical) then perform one additional step before applying the approach mentioned above. First perform response variable encoding or one-hot encoding to convert the categorical feature into a numeric or *quantitative* feature.

Basically the data is split into two sets trying out various combinations of features and different threshold split values of those features. For each of these combinations we compute the cost function as a weighted average over the two sets.

$$Cost\,function = \frac{|S_1|}{|pn|} * G_{S_1} + \frac{|S_2|}{|pn|} * G_{S_2} \text{ .....[1]}$$

$$Cost\,function = \frac{|S_1|}{|pn|} * H_{S_1} + \frac{|S_2|}{|pn|} * H_{S_2} \text{ .....[2]}$$

Equation [1] above is the cost function using gini impurity and equation [2] is the cost function using entropy. Now that we have the average weighted cost of the two child nodes we introduce the concept of *information gain.*

INFORMATION GAIN

It is this measure that the algorithm tries to optimize for **classification** tasks - it ensures that the features with the most information (tells us more about the response variable) is chosen first and that subsequent features hold less information. The further the tree branches downward the less information a feature provides. This is why decision trees are used for feature importance - we only need to look at the first few features at the top of the tree to understand which set of features are most important.

So back to the equation of information gain:

$$I_g = G(parent\ node) - \sum_{j\ \varepsilon\ child\ node} \frac{|S_j|}{|pn|} * G_{S_j} \text{ .....[1]}$$

$$I_g = H(parent\ node) - \sum_{j\ \varepsilon\ child\ node} \frac{|S_j|}{|pn|} * H_{S_j} \text{ .....[2]}$$

Equation [1] computes the difference in gini impurity between the parent and weighted average gini of the two child nodes. Equation [2] computes the difference in entropy between the parent and weighted average entropy of the two child nodes. For the example discussion equation [1] will be used.

EXAMPLE CLASSIFICATION

*Please note that within this example I do not perform class balancing or normalisation.

Let's say we have a small training dataset like this:

Weight	Height	Label
8	8	dog
50	40	dog
8	9	cat
15	12	dog
9	10	cat

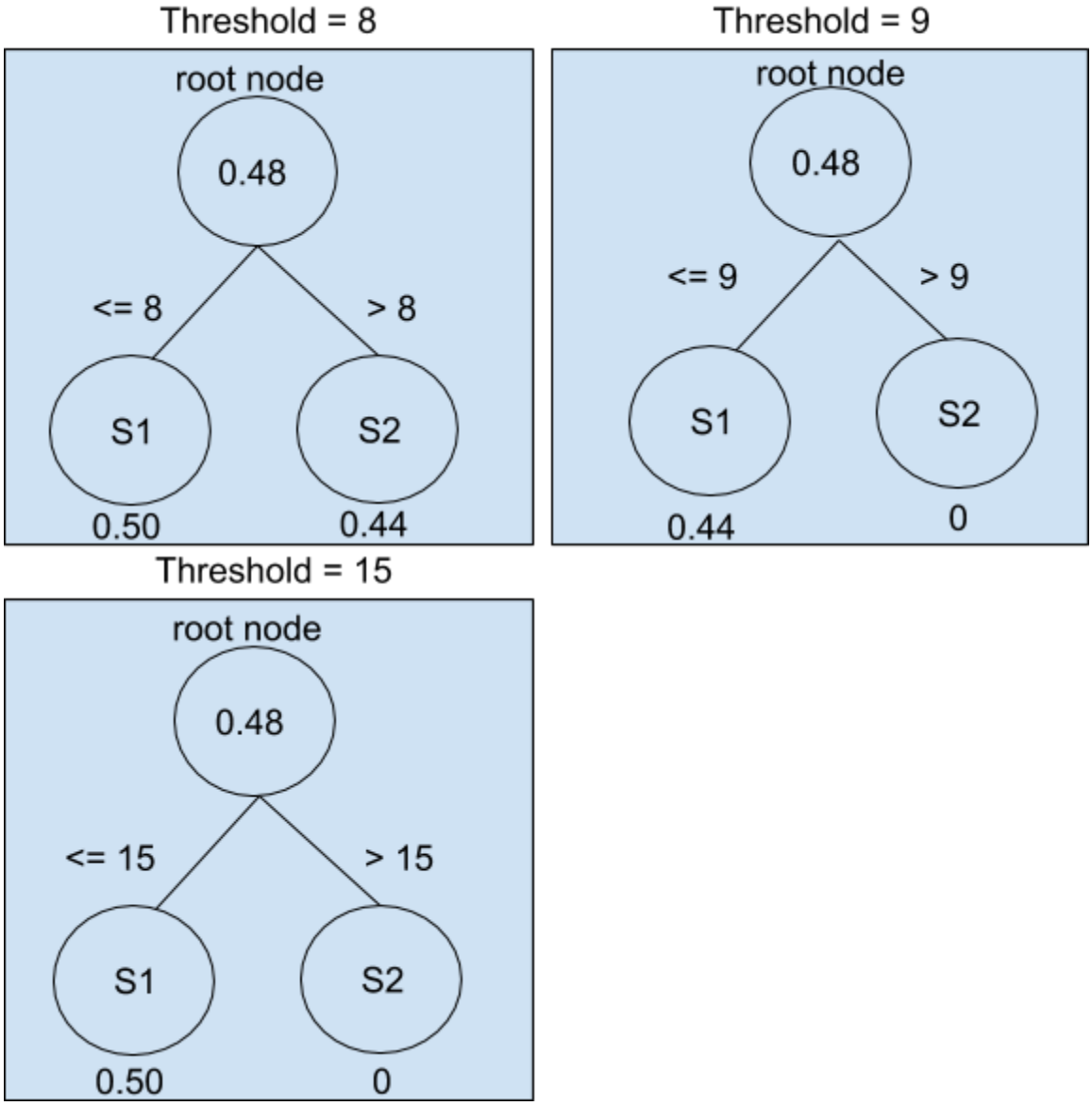
Consisting out of two independent features (weight and height) and we use these to predict a label in one of two classes (cat or dog). Making this a binary classification problem.

All the training data is available at the root node, so as a first step we compute the gini impurity for the root node.

$$G_{root} = 1 - [(2/5)^2 + (3/5)^2] = 0.48$$

The next step would be to consider all the independent features and try different threshold values for each - splitting the root node into two child nodes.

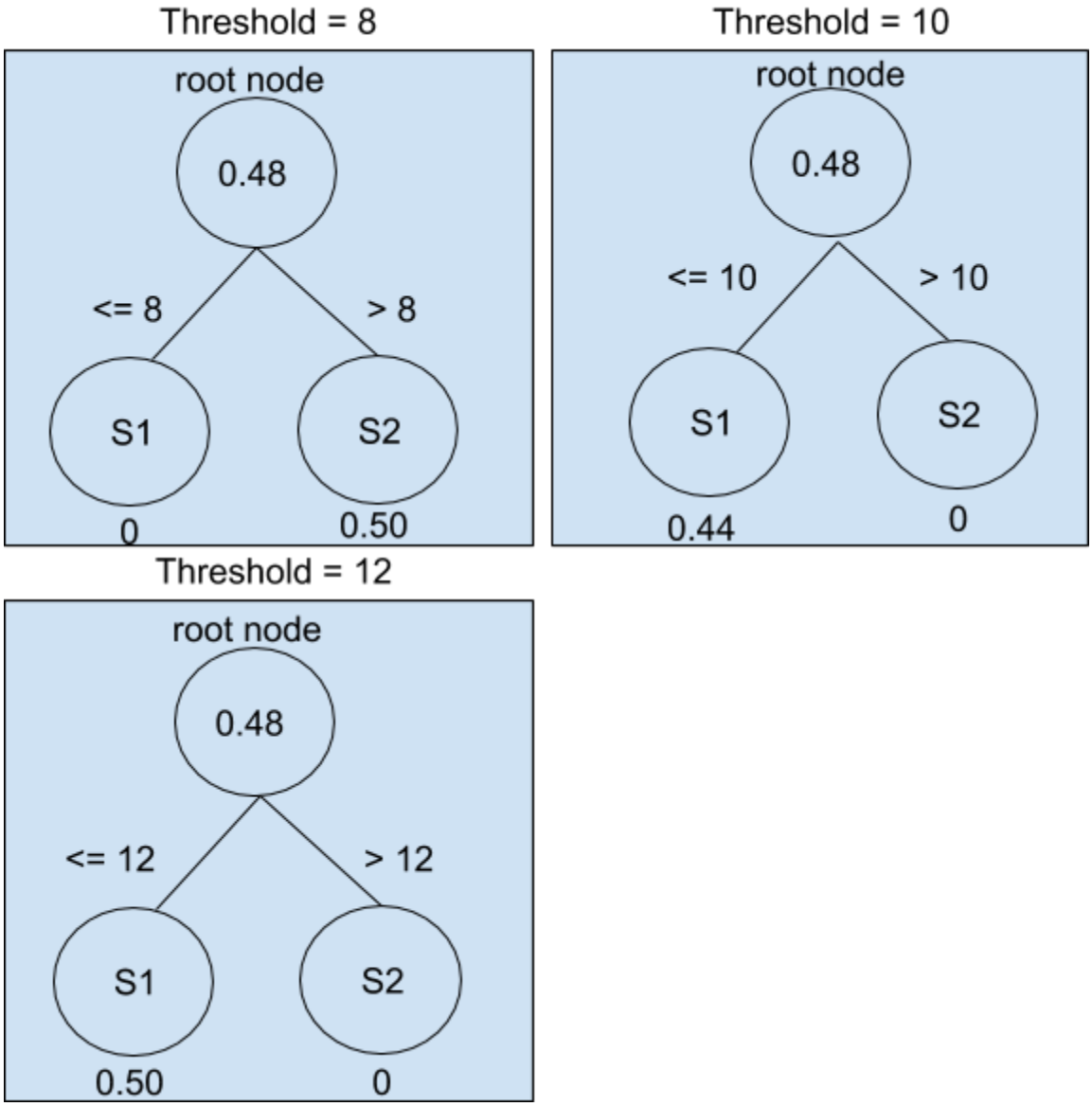
Possible splits - weight feature



For each threshold calculate the size of each child node along with the gini value.

Split (threshold)	|S1|	|S2|	G(S1)	G(S2)
8	2	3	1 - (½)^2 - (½)^2 = 0.50	1 - (⅓)^2 - (⅔)^2 = 0.44
9	3	2	1 - (⅔)^2 - (⅓)^2 = 0.44	1 - (0)^2 - (1)^2 = 0
15	4	1	1 - (½)^2 - (½)^2 = 0.5	1 - (0)^2 - (1)^2 = 0

Possible splits - height feature



For each threshold calculate the size of each child node along with the gini value.

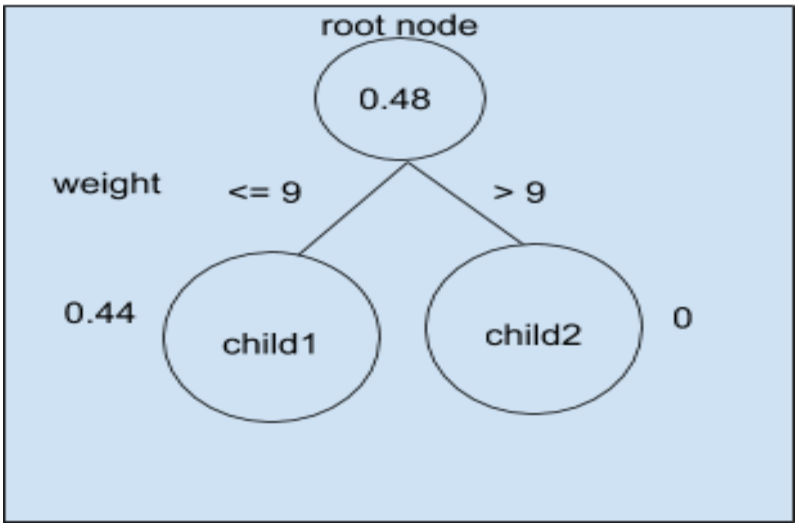
Split (threshold)	|S1|	|S2|	G(S1)	G(S2)
8	1	4	1 - (0)^2 - (1)^2 = 0	1 - (2/4)^2 - (2/4)^2 = 0.5
10	3	2	1 - (⅔)^2 - (⅓)^2 = 0.44	1 - (0)^2 - (1)^2 = 0
12	4	1	1 - (½)^2 - (½)^2 = 0.5	1 - (0)^2 - (1)^2 = 0

Keep track of the feature, threshold and cost combination.

feature	threshold	Information gain
weight	*8*	*0.48 - [(⅘)*0.5 + (⅕)*0.44] = 0.016*
weight	*9*	*0.48 - [(⅗)*0.44 + (⅖)*0] = 0.216*
weight	*15*	*0.48 - [(⅘)*0.5 + (⅕)*0] = 0.08*
height	*8*	*0.48 - [(½)*0 + (½)*0.5] = 0.08*
height	*10*	*0.48 - [(⅗)*0.44 + (⅖)*0] = 0.216*
height	*12*	*0.48 - [(⅘)*0.5 + (⅕)*0] = 0.08*

The above table shows two red entries that resulted in a tie for the highest information gain value. If there was a clear winner the greedy algorithm would have picked that feature and threshold as the first feature to branch on, but in this case we can pick anyone. So let's pick the weight feature first.

Then our tree would look like this



Now there is only the height feature left on which we will apply the same steps as before since the algorithm is recursive. The only difference is it will branch from the newly formed child nodes instead of the root node.

Let's see what observations currently reside in child node 1 and child node 2

Child node 1 observations

Weight	Height	Label
8	8	dog
8	9	cat
9	10	cat

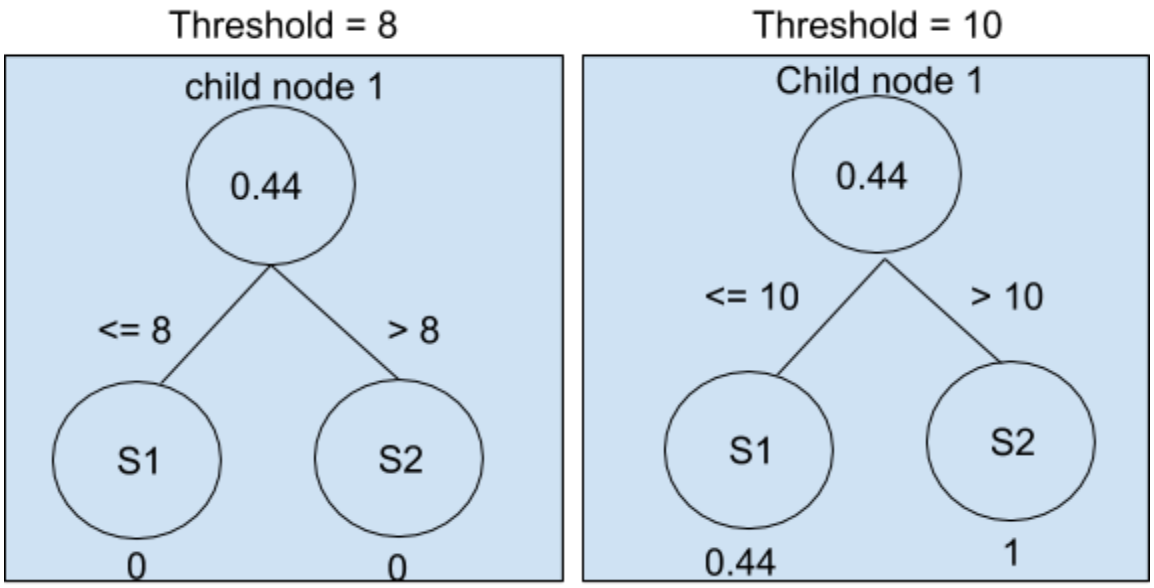
Child node 2 observations

Weight	Height	Label
50	40	dog
15	12	dog

Since child node 2 only contains observations of one class (pure node) there is no need to further branch from here - the tree will stop growing from this node down. For child node 1 if the algorithm was regularized - by containing a minimum number of observations (let's say 3) then it would terminate here as well and the predicted class would then have just been the most observed class (in this case - cat). For this example the algorithm is not regularized (we will overfit) so we will continue branching…

The computed gini value at child node 1 (as from the previous iteration) is 0.44. We will use this new value to compute information gain for the next split. Let's consider the height feature with two different threshold splits.

Possible splits - height feature



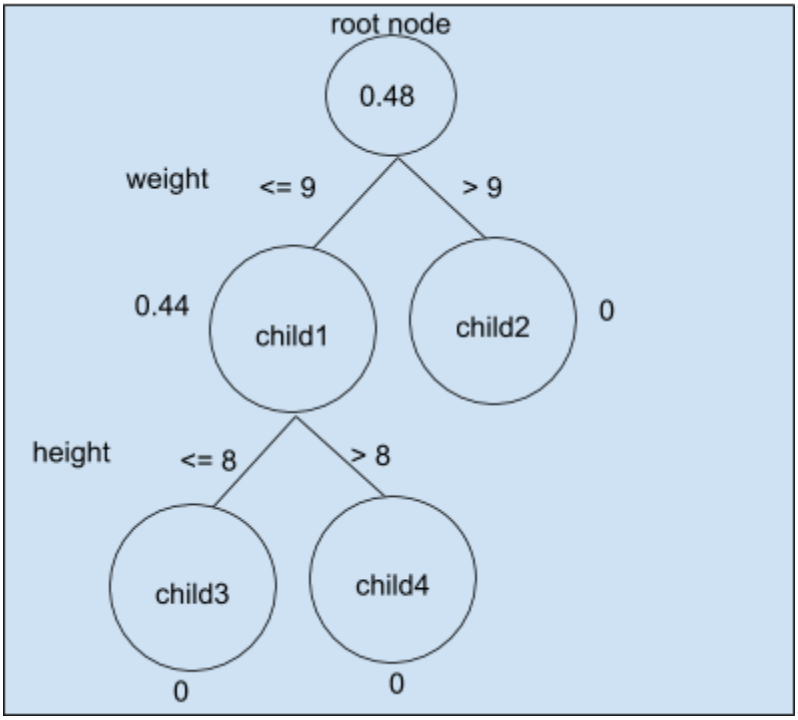
For each threshold calculate the size of each child node along with the gini value.

Split (threshold)	|S1|	|S2|	G(S1)	G(S2)
8	1	2	1 - (0)^2 - (1)^2 = 0	1 - (1)^2 - (0)^2 = 0
10	3	0	1 - (⅔)^2 - (⅓)^2 = 0.44	1 - (0)^2 - (0)^2 = 1

Keep track of the feature, threshold and cost combination.

feature	threshold	Information gain
height	*8*	*0.44 - [(½)*0 + (½)*0] = 0.44*
height	*10*	*0.44 - [(3/3)*0.44 + (0/3)*1] = 0*

In this iteration the greedy algorithm will pick the threshold value of 8. Now the constructed tree will look as follows:



Let's see what observations currently reside in child node 3 and child node 4

Child node 3 observations

Weight	Height	Label
8	8	dog

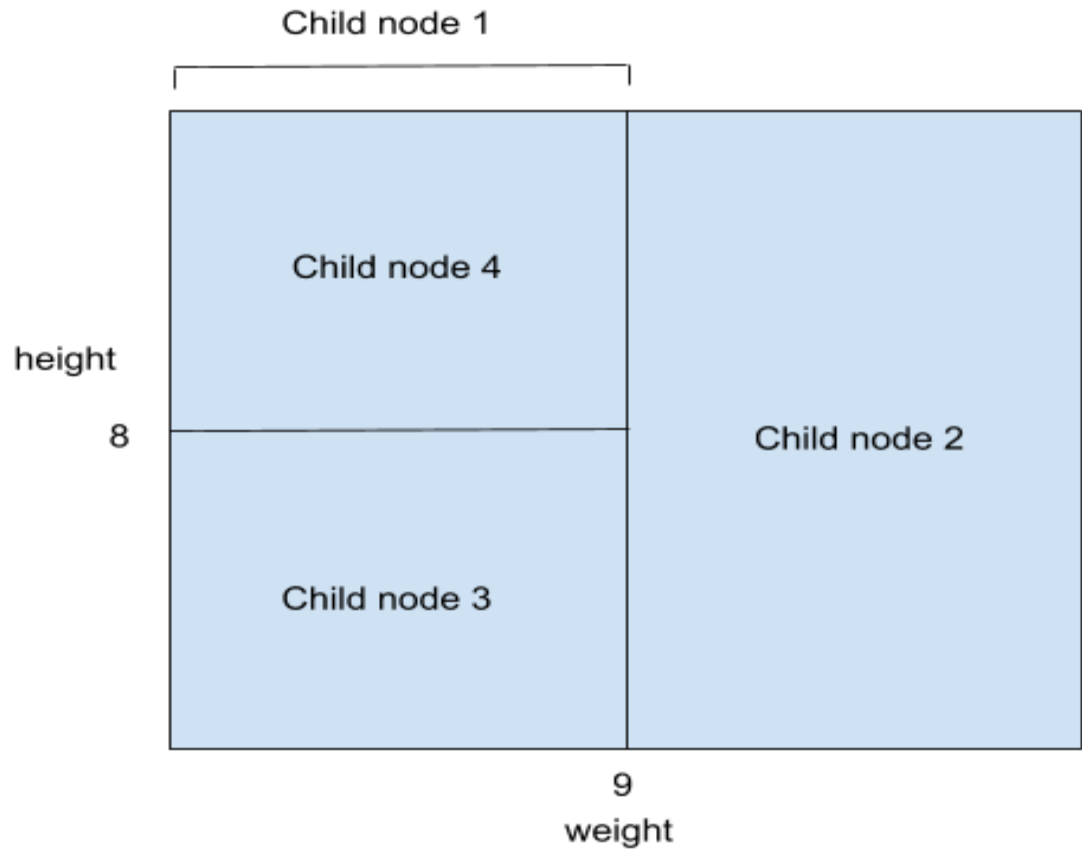
Child node 4 observations

Weight	Height	Label
8	9	cat
9	10	cat

Each leaf node is now only pure nodes and the algorithm will stop.

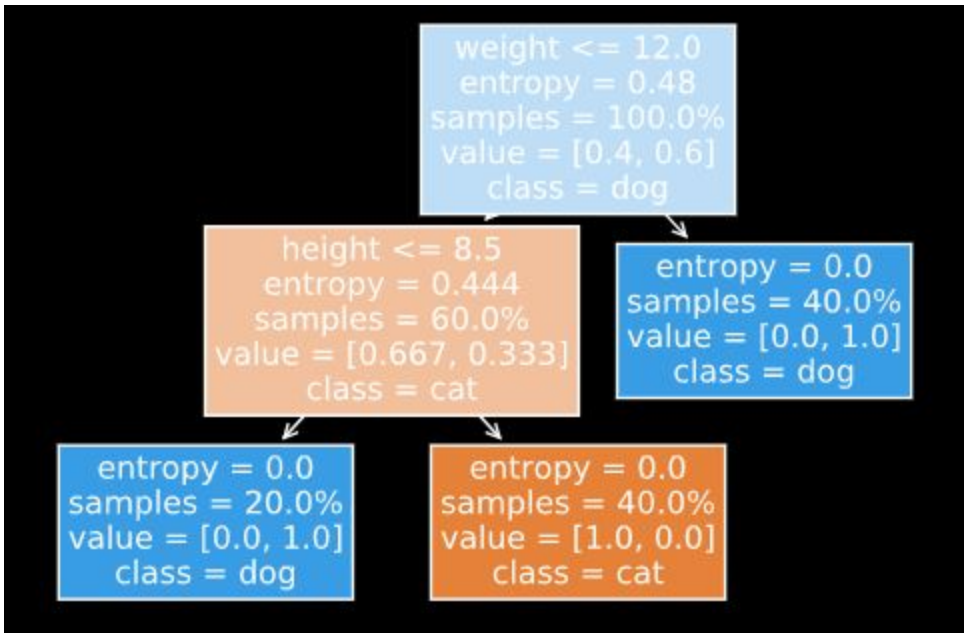
DECISION BOUNDARY

Each of the regions in the image below represents a node within the tree. The greedy algorithm will filter observations and bucket them into these type of boxed regions. In this particular image there are only two quantitative features - weight and height. The vertical and horizontal lines show where the threshold values are.



IN CODE

```
# import libraries:

import pandas as pd

from sklearn.tree import DecisionTreeClassifier

from sklearn.tree import plot_tree

# build data:

data =
pd.DataFrame({'weight':[8,50,8,15,9],'height':[8,40,9,12,10],'la
bel':['dog','dog','cat','dog','cat']})

# define base model (no regularization):

dt_model = DecisionTreeClassifier()

# Pass the data to the model (no train - CV - test splitting):

dt_model.fit(X=data.loc[:,['weight','height']],y=data.loc[:,'lab
el'])

# Show the tree structure

plot_tree(  dt_model,

            feature_names=data.columns.values[0:2],

            class_names=list(set(data['label'].values)),

            filled=True,

            proportion=True)
```



REGRESSION IMPLEMENTATION

For a use case, California housing price data set will be used and the decision tree algorithm will be applied to this data set. This will be a very basic procedure of how to use Python to implement such a model. By no means is this a fully applied EDA process as the intent of the blog is to explain how the decision algorithm works and not a blog for EDA processes.

Here we load some libraries and collect the data.

In [1]:
```
import pandas as pd
import numpy as np
from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_selection import chi2, f_regression, SelectKBest
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.manifold import TSNE

%matplotlib inline
```

In [2]:
```
X,Y = datasets.fetch_california_housing(return_X_y=True)
X = pd.DataFrame(X,columns=['MedInc median income in block',
                            'HouseAge median house age in block',
                            'AveRooms average number of rooms',
                            'AveBedrms average number of bedrooms',
                            'Population block population',
                            'AveOccup average house occupancy',
                            'Latitude house block latitude',
                            'Longitude house block longitude'])
```

Split the data into train, CV and test sets

In [3]:
```
X_train,X_test,Y_train,Y_test = train_test_split(X,Y,train_size=0.8)
X_train,X_cv,Y_train,Y_cv = train_test_split(X_train,Y_train,train_size=0.8)

print('train',X_train.shape,Y_train.shape)
print('cv',X_cv.shape,Y_cv.shape)
print('test',X_test.shape,Y_test.shape)

```

```
train (13209, 8) (13209,)
cv (3303, 8) (3303,)
test (4128, 8) (4128,)
```

In [4]:
```
X_train.sample(n=5)
```

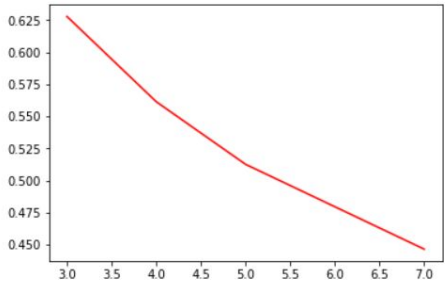
Out[4]:

	MedInc median income in block	HouseAge median house age in block	AveRooms average number of rooms	AveBedrms average number of bedrooms	Population block population	AveOccup average house occupancy	Latitude house block latitude	Longitude house block longitude
13313	1.4464	35.0	4.231810	1.101523	1182.0	2.000000	34.07	-117.65
7699	4.8523	36.0	5.941667	0.979167	698.0	2.908333	33.96	-118.12
5899	3.9464	37.0	4.928736	1.025287	860.0	1.977011	34.16	-118.31
2270	2.0300	38.0	4.686275	1.117647	1131.0	2.772059	36.78	-119.79
19941	2.5000	32.0	4.862857	0.994286	1016.0	2.902857	36.25	-119.46

Perform basic parameter tuning. In this case only the depth of the tree is tuned

In [5]:
```
# define base model (no regularization):
mse = list()
depths = [3,4,5,7]
for depth in depths:
    dt_model = DecisionTreeRegressor(min_samples_leaf=5,max_depth=depth)
    dt_model.fit(X=X_train,y=Y_train)
    mse.append(mean_squared_error(y_true=Y_cv,y_pred=dt_model.predict(X=X_cv)))
```

In [6]:
```
plt.plot(depths,mse,color='r')
plt.show()
```

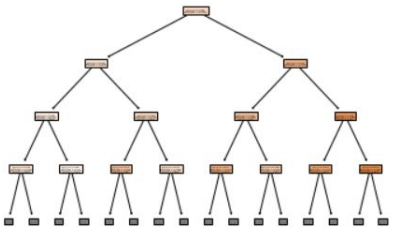


Use the best stable model and apply it to the test set for making predictions. Below the model is also plotted to show the greedy algorithm approach and order in which features were used to split the observations.

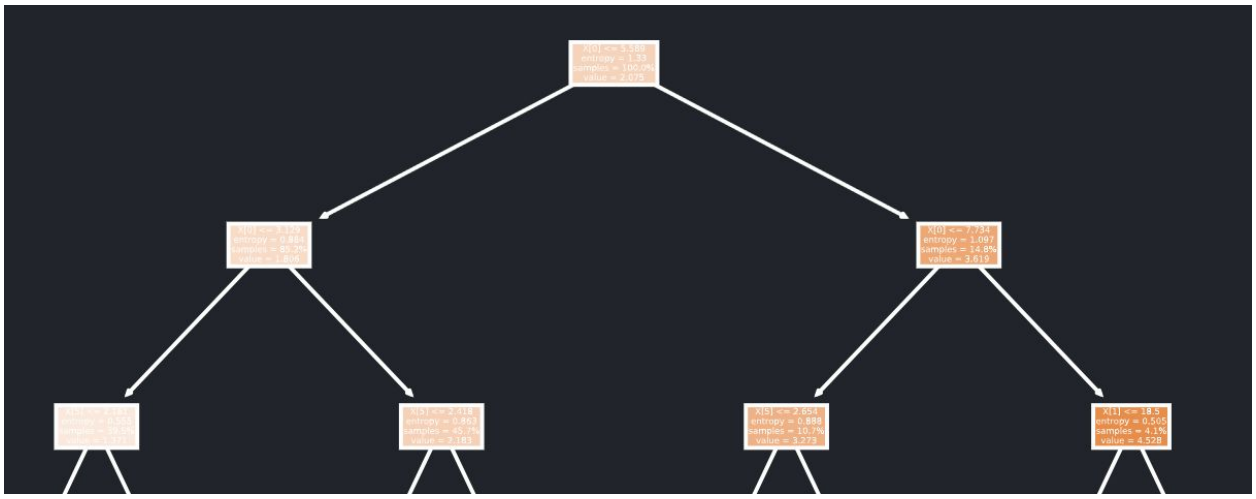
In [7]:
```
best_model = DecisionTreeRegressor(min_samples_leaf=5,max_depth=7)
best_model.fit(X=X_train,y=Y_train)
error = mean_squared_error(y_true=Y_cv,y_pred=best_model.predict(X=X_cv))
print('performance:',error)
```

```
performance: 0.4463799332169963
```

In [8]:
```
# Show the tree structure
plot_tree(best_model,filled=True,proportion=True,max_depth=3)
plt.show()

```



Enlargement of the top layer of splits



Using two different approaches to determine the top 3 features. Approach 1 was to use the model features and approach 2 was to use chi2 test. It can be noted that both approaches return very similar values.

In [9]:
```
model_best_features = X.columns.values[[j for i,j in sorted([(j,i) for i,j in list(enumerate(best_model.feature_importances_
model_best_features
```

Out[9]:
```
array(['MedInc median income in block',
       'AveOccup average house occupancy',
       'Latitude house block latitude'], dtype=object)
```

In [10]:
```
top_3_features = SelectKBest(score_func=f_regression,k=3)
top_3_features.fit_transform(X=X,y=Y)
stats_best_features = X.columns.values[top_3_features.get_support()]
stats_best_features
```

Out[10]:
```
array(['MedInc median income in block',
       'AveRooms average number of rooms',
       'Latitude house block latitude'], dtype=object)
```

CONCLUSION

Decision trees has a lot going for them, and it remains my favourite decision tool. Please see the references below, these resources helped me and I'm sure it might help you in case you get stuck on some aspect of decision trees.

REFERENCES

https://medium.com/greyatom/decision-trees-a-simple-way-to-visualize-a-decision-dc506a403aeb

https://medium.com/machine-learning-researcher/decision-tree-algorithm-in-machine-learning-248fb7de819e

O'reilly, Hands-On machine learning with Scikit-learn & Tensorflow, [pg.167-178]

An introduction to statistical learning with applications in R, [pg.318-329]

https://scikit-learn.org/stable/modules/tree.html

https://towardsdatascience.com/decision-tree-an-algorithm-that-works-like-the-human-brain-8bc0652f1fc6

https://www.appliedaicourse.com/

APPENDIX

Common terms used with Decision trees:

1. **Root Node:** It represents entire population or sample and this further gets divided into two or more homogeneous sets.
2. **Splitting:** It is a process of dividing a node into two or more sub-nodes.
3. **Decision Node:** When a sub-node splits into further sub-nodes, then it is called decision node.
4. **Leaf/ Terminal Node:** Nodes that do not split is called Leaf or Terminal node.
5. **Pruning:** When we remove sub-nodes of a decision node, this process is called pruning.
6. **Branch / Sub-Tree:** A subsection of entire tree is called branch or sub-tree.
7. **Parent and Child Node:** A node, which is divided into sub-nodes is called parent

node of sub-nodes whereas sub-nodes are the children of parent node.