Decision Tree Classification Algorithm

* Decision Tree is a **Supervised learning technique**that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree-structured classifier, where**internal nodes represent the features of a dataset, branches represent the decision rules** and **each leaf node represents the outcome.**
* In a Decision tree, there are two nodes, which are the **Decision Node** and**Leaf Node.** Decision nodes are used to make any decision and have multiple branches, whereas Leaf nodes are the output of those decisions and do not contain any further branches.
* The decisions or the test are performed on the basis of features of the given dataset.
* ***It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.***
* It is called a decision tree because, similar to a tree, it starts with the root node, which expands on further branches and constructs a tree-like structure.
* In order to build a tree, we use the **CART algorithm,** which stands for **Classification and Regression Tree algorithm.**
* A decision tree simply asks a question, and based on the answer (Yes/No), it further split the tree into subtrees.
* Below diagram explains the general structure of a decision tree:

Note: A decision tree can contain categorical data (YES/NO) as well as numeric data.



Why use Decision Trees?

There are various algorithms in Machine learning, so choosing the best algorithm for the given dataset and problem is the main point to remember while creating a machine learning model. Below are the two reasons for using the Decision tree:

* Decision Trees usually mimic human thinking ability while making a decision, so it is easy to understand.
* The logic behind the decision tree can be easily understood because it shows a tree-like structure.

Decision Tree Terminologies

 **Root Node:** Root node is from where the decision tree starts. It represents the entire dataset, which further gets divided into two or more homogeneous sets.

 **Leaf Node:** Leaf nodes are the final output node, and the tree cannot be segregated further after getting a leaf node.

 **Splitting:** Splitting is the process of dividing the decision node/root node into sub-nodes according to the given conditions.

 **Branch/Sub Tree:** A tree formed by splitting the tree.

 **Pruning:** Pruning is the process of removing the unwanted branches from the tree.

 **Parent/Child node:** The root node of the tree is called the parent node, and other nodes are called the child nodes.

**How does the Decision Tree algorithm Work?**

In a decision tree, for predicting the class of the given dataset, the algorithm starts from the root node of the tree. This algorithm compares the values of root attribute with the record (real dataset) attribute and, based on the comparison, follows the branch and jumps to the next node.

For the next node, the algorithm again compares the attribute value with the other sub-nodes and move further. It continues the process until it reaches the leaf node of the tree.

## Advantages of the Decision Tree

1) It is simple to implement and it follows a flow chart type structure that resembles human-like decision making.

2)It proves to be very useful for decision-related problems.

3) It helps to find all of the possible outcomes for a given problem.

4) There is very little need for data cleaning in decision trees compared to other Machine Learning algorithms.

5) Handles both numerical as well as categorical values

## Disadvantages of the Decision Tree

1) Too many layers of decision tree make it extremely complex sometimes.

2) It may result in overfitting (which can be resolved using the **Random Forest algorithm)**

3) For large number of the class labels, the computational complexity of the decision tree increases.

Bank Marketing Data - A Decision Tree Approach

Aim:

The aim of this attempt is to predict if the client will subscribe (yes/no) to a term deposit, by building a classification model using Decision Tree.

In [1]:

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.cluster import KMeans

from sklearn import datasets

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

%matplotlib inline

In [2]:

*# Load data file*

bank=pd.read\_csv('bank.csv')

bank.head()

Summay of data

Categorical Variables :

**[1] job :** admin,technician, services, management, retired, blue-collar, unemployed, entrepreneur, housemaid, unknown, self-employed, student  
**[2] marital :** married, single, divorced  
**[3] education:** secondary, tertiary, primary, unknown  
**[4] default :** yes, no  
**[5] housing :** yes, no  
**[6] loan :** yes, no  
**[7] deposit :** yes, no **(Dependent Variable)**  
**[8] contact :** unknown, cellular, telephone  
**[9] month :** jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec  
**[10] poutcome:** unknown, other, failure, success

Numerical Variables:

**[1] age  
[2] balance  
[3] day  
[4] duration  
[5] campaign  
[6] pdays  
[7] previous**

In [3]:

*# Check if the data set contains any null values - Nothing found!*

bank[bank.isnull().any(axis=1)].count()

In [4]:

bank.describe()

In [5]:

*# Boxplot for 'age'*

g = sns.boxplot(x=bank["age"])

In [6]:

*# Distribution of Age*

sns.distplot(bank.age, bins=100)

In [7]:

*# Boxplot for 'duration'*

g = sns.boxplot(x=bank["duration"])

In [8]:

sns.distplot(bank.duration, bins=100)

Convert categorical data

In [9]:

*# Make a copy for parsing*

bank\_data = bank.copy()

------------------------------ job ------------------------------

In [10]:

*# Explore People who made a deposit Vs Job category*

jobs = ['management','blue-collar','technician','admin.','services','retired','self-employed','student',\

'unemployed','entrepreneur','housemaid','unknown']

for j **in** jobs:

print("**{:15}** : **{:5}**". format(j, len(bank\_data[(bank\_data.deposit == "yes") & (bank\_data.job ==j)])))

In [11]:

*# Different types of job categories and their counts*

bank\_data.job.value\_counts()

int64

In [12]:

*# Combine similar jobs into categiroes*

bank\_data['job'] = bank\_data['job'].replace(['management', 'admin.'], 'white-collar')

bank\_data['job'] = bank\_data['job'].replace(['services','housemaid'], 'pink-collar')

bank\_data['job'] = bank\_data['job'].replace(['retired', 'student', 'unemployed', 'unknown'], 'other')

In [13]:

*# New value counts*

bank\_data.job.value\_counts()

----------------------------- poutcome ------------------------------

In [14]:

bank\_data.poutcome.value\_counts()

In [15]:

*# Combine 'unknown' and 'other' as 'other' isn't really match with either 'success' or 'failure'*

bank\_data['poutcome'] = bank\_data['poutcome'].replace(['other'] , 'unknown')

bank\_data.poutcome.value\_counts()

------------------------------ contact ------------------------------

In [16]:

*# Drop 'contact', as every participant has been contacted.*

bank\_data.drop('contact', axis=1, inplace=True)

------------------------------ default ------------------------------

In [17]:

*# values for "default" : yes/no*

bank\_data["default"]

bank\_data['default\_cat'] = bank\_data['default'].map( {'yes':1, 'no':0} )

bank\_data.drop('default', axis=1,inplace = True)

------------------------------ housing ------------------------------

In [18]:

*# values for "housing" : yes/no*

bank\_data["housing\_cat"]=bank\_data['housing'].map({'yes':1, 'no':0})

bank\_data.drop('housing', axis=1,inplace = True)

------------------------------ loan ------------------------------

In [19]:

*# values for "loan" : yes/no*

bank\_data["loan\_cat"] = bank\_data['loan'].map({'yes':1, 'no':0})

bank\_data.drop('loan', axis=1, inplace=True)

------------------------------ month, day ------------------------------

In [20]:

*# day : last contact day of the month*

*# month: last contact month of year*

*# Drop 'month' and 'day' as they don't have any intrinsic meaning*

bank\_data.drop('month', axis=1, inplace=True)

bank\_data.drop('day', axis=1, inplace=True)

------------------------------ deposit ------------------------------

In [21]:

*# values for "deposit" : yes/no*

bank\_data["deposit\_cat"] = bank\_data['deposit'].map({'yes':1, 'no':0})

bank\_data.drop('deposit', axis=1, inplace=True)

------------------------------ pdays ------------------------------

In [22]:

*# pdays: number of days that passed by after the client was last contacted from a previous campaign*

*# -1 means client was not previously contacted*

print("Customers that have not been contacted before:", len(bank\_data[bank\_data.pdays==-1]))

print("Maximum values on padys :", bank\_data['pdays'].max())

Customers that have not been contacted before: 8324

Maximum values on padys : 854

In [23]:

*# Map padys=-1 into a large value (10000 is used) to indicate that it is so far in the past that it has no effect*

bank\_data.loc[bank\_data['pdays'] == -1, 'pdays'] = 10000

In [24]:

*# Create a new column: recent\_pdays*

bank\_data['recent\_pdays'] = np.where(bank\_data['pdays'], 1/bank\_data.pdays, 1/bank\_data.pdays)

*# Drop 'pdays'*

bank\_data.drop('pdays', axis=1, inplace = True)

In [25]:

bank\_data.tail()

------------------------------ Convert to dummy values ------------------------------

In [26]:

*# Convert categorical variables to dummies*

bank\_with\_dummies = pd.get\_dummies(data=bank\_data, columns = ['job', 'marital', 'education', 'poutcome'], \

prefix = ['job', 'marital', 'education', 'poutcome'])

bank\_with\_dummies.head()

In [27]:

bank\_with\_dummies.shape

(11162, 27)

In [28]:

bank\_with\_dummies.describe()

Observations on whole population

In [29]:

*# Scatterplot showing age and balance*

bank\_with\_dummies.plot(kind='scatter', x='age', y='balance');

*# Across all ages, majority of people have savings of less than 20000.*

In [30]:

bank\_with\_dummies.plot(kind='hist', x='poutcome\_success', y='duration');

Analysis on people who sign up for a term deposite

In [31]:

*# People who sign up to a term deposite*

bank\_with\_dummies[bank\_data.deposit\_cat == 1].describe()

In [32]:

*# People signed up to a term deposite having a personal loan (loan\_cat) and housing loan (housing\_cat)*

len(bank\_with\_dummies[(bank\_with\_dummies.deposit\_cat == 1) & (bank\_with\_dummies.loan\_cat) & (bank\_with\_dummies.housing\_cat)])

265

In [33]:

*# People signed up to a term deposite with a credit default*

len(bank\_with\_dummies[(bank\_with\_dummies.deposit\_cat == 1) & (bank\_with\_dummies.default\_cat ==1)])

Out[33]:

52

In [34]:

*# Bar chart of job Vs deposite*

plt.figure(figsize = (10,6))

sns.barplot(x='job', y = 'deposit\_cat', data = bank\_data)

In [35]:

*# Bar chart of "previous outcome" Vs "call duration"*

plt.figure(figsize = (10,6))

sns.barplot(x='poutcome', y = 'duration', data = bank\_data)

Classification

In [36]:

*# make a copy*

bankcl = bank\_with\_dummies

In [37]:

*# The Correltion matrix*

corr = bankcl.corr()

corr

In [38]:

*# Heatmap*

plt.figure(figsize = (10,10))

cmap = sns.diverging\_palette(220, 10, as\_cmap=True)

sns.heatmap(corr, xticklabels=corr.columns.values, yticklabels=corr.columns.values, cmap=cmap, vmax=.3, center=0, square=True, linewidths=.5, cbar\_kws={"shrink": .82})

plt.title('Heatmap of Correlation Matrix')

In [39]:

*# Extract the deposte\_cat column (the dependent variable)*

corr\_deposite = pd.DataFrame(corr['deposit\_cat'].drop('deposit\_cat'))

corr\_deposite.sort\_values(by = 'deposit\_cat', ascending = False)

Build the Data Model

In [40]:

*# Train-Test split: 20% test data*

data\_drop\_deposite = bankcl.drop('deposit\_cat', 1)

label = bankcl.deposit\_cat

data\_train, data\_test, label\_train, label\_test = train\_test\_split(data\_drop\_deposite, label, test\_size = 0.2, random\_state = 50)

In [41]:

*# Decision tree with depth = 2*

dt2 = tree.DecisionTreeClassifier(random\_state=1, max\_depth=2)

dt2.fit(data\_train, label\_train)

dt2\_score\_train = dt2.score(data\_train, label\_train)

print("Training score: ",dt2\_score\_train)

dt2\_score\_test = dt2.score(data\_test, label\_test)

print("Testing score: ",dt2\_score\_test)

Training score: 0.728525030799

Testing score: 0.726824899239

In [42]:

*# Decision tree with depth = 3*

dt3 = tree.DecisionTreeClassifier(random\_state=1, max\_depth=3)

dt3.fit(data\_train, label\_train)

dt3\_score\_train = dt3.score(data\_train, label\_train)

print("Training score: ",dt3\_score\_train)

dt3\_score\_test = dt3.score(data\_test, label\_test)

print("Testing score: ",dt3\_score\_test)

Training score: 0.770411020271

Testing score: 0.757277205553

In [43]:

*# Decision tree with depth = 4*

dt4 = tree.DecisionTreeClassifier(random\_state=1, max\_depth=4)

dt4.fit(data\_train, label\_train)

dt4\_score\_train = dt4.score(data\_train, label\_train)

print("Training score: ",dt4\_score\_train)

dt4\_score\_test = dt4.score(data\_test, label\_test)

print("Testing score: ",dt4\_score\_test)Testing score: 0.774294670846

In [44]:

*# Decision tree with depth = 6*

dt6 = tree.DecisionTreeClassifier(random\_state=1, max\_depth=6)

dt6.fit(data\_train, label\_train)

dt6\_score\_train = dt6.score(data\_train, label\_train)

print("Training score: ",dt6\_score\_train)

dt6\_score\_test = dt6.score(data\_test, label\_test)

print("Testing score: ",dt6\_score\_test)

Training score: 0.808041214022

Testing score: 0.779668607255

In [45]:

*# Decision tree: To the full depth*

dt1 = tree.DecisionTreeClassifier()

dt1.fit(data\_train, label\_train)

dt1\_score\_train = dt1.score(data\_train, label\_train)

print("Training score: ", dt1\_score\_train)

dt1\_score\_test = dt1.score(data\_test, label\_test)

print("Testing score: ", dt1\_score\_test)

Training score: 1.0

Compare Training and Testing scores for various tree depths used

In [46]:

print('**{:10}** **{:20}** **{:20}**'.format('depth', 'Training score','Testing score'))

print('**{:10}** **{:20}** **{:20}**'.format('-----', '--------------','-------------'))

print('**{:1}** **{:>25}** **{:>20}**'.format(2, dt2\_score\_train, dt2\_score\_test))

print('**{:1}** **{:>25}** **{:>20}**'.format(3, dt3\_score\_train, dt3\_score\_test))

print('**{:1}** **{:>25}** **{:>20}**'.format(4, dt4\_score\_train, dt4\_score\_test))

print('**{:1}** **{:>25}** **{:>20}**'.format(6, dt6\_score\_train, dt6\_score\_test))

print('**{:1}** **{:>23}** **{:>20}**'.format("max", dt1\_score\_train, dt1\_score\_test))

depth Training score Testing score 1.0 0.7411553963278101

It could be seen that, higher the depth, training score increases and matches perfects with the training data set. However higher the depth the tree goes, it overfit to the training data set. So it's no use keep increasing the tree depth. According to above observations, tree with a depth of 2 seems more reasonable as both training and test scores are reasonably high.

In [47]:

*# Let's generate the decision tree for depth = 2*

*# Create a feature vector*

features = bankcl.columns.tolist()

*# Uncomment below to generate the digraph Tree.*

*#tree.export\_graphviz(dt2, out\_file='tree\_depth\_2.dot', feature\_names=features)*

**Contents of "tree\_depth\_2.dot":**  
digraph Tree {  
node [shape=box] ;  
0 [label="duration <= 206.5\ngini = 0.4986\nsamples = 8929\nvalue = [4700, 4229]"] ;  
1 [label="poutcome\_failure <= 0.5\ngini = 0.3274\nsamples = 3612\nvalue = [2867, 745]"] ;  
0 -> 1 [labeldistance=2.5, labelangle=45, headlabel="True"] ;  
2 [label="gini = 0.2733\nsamples = 3380\nvalue = [2828, 552]"] ;  
1 -> 2 ;  
3 [label="gini = 0.2797\nsamples = 232\nvalue = [39, 193]"] ;  
1 -> 3 ;  
4 [label="duration <= 441.5\ngini = 0.4518\nsamples = 5317\nvalue = [1833, 3484]"] ;  
0 -> 4 [labeldistance=2.5, labelangle=-45, headlabel="False"] ;  
5 [label="gini = 0.4996\nsamples = 2762\nvalue = [1340, 1422]"] ;  
4 -> 5 ;  
6 [label="gini = 0.3114\nsamples = 2555\nvalue = [493, 2062]"] ;  
4 -> 6 ;  
}

Thee decision tree for depth =2

Based on the decision tree results, it could be seen that higher the "duration", bank is able to sign up more people to term deposits.

In [48]:

*# Two classes: 0 = not signed up, 1 = signed up*

dt2.classes\_

In [49]:

*# Create a feature vector*

features = data\_drop\_deposite.columns.tolist()

features

In [50]:

*# Investigate most important features with depth =2*

dt2 = tree.DecisionTreeClassifier(random\_state=1, max\_depth=2)

*# Fit the decision tree classifier*

dt2.fit(data\_train, label\_train)

fi = dt2.feature\_importances\_

l = len(features)

for i **in** range(0,len(features)):

print('**{:.<20}** **{:3}**'.format(features[i],fi[i]))

age................. 0.0

balance.............

Predictions

In [51]:

*# According to feature importance results, most importtant feature is the "Duration"*

*# Let's calculte statistics on Duration*

print("Mean duration : ", data\_drop\_deposite.duration.mean())

print("Maximun duration: ", data\_drop\_deposite.duration.max())

print("Minimum duration: ", data\_drop\_deposite.duration.min())

Minimum duration: 2

In [52]:

*# Predict: Successful deposite with a call duration = 371 sec*

print(dt2.predict\_proba(np.array([0, 0, 371, 0, 0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0]).reshape(1, -1)))

print(dt2.predict(np.array([0, 0, 371, 0, 0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0]).reshape(1, -1)))

*# column 0: probability for class 0 (not signed for term deposite) & column 1: probability for class 1*

*# Probability of Successful deposite = 0.51484432*

[1]

In [53]:

*# Predict: Successful deposite with a maximun call duration = 3881 sec*

print(dt2.predict\_proba(np.array([0, 0, 3881, 0, 0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0]).reshape(1, -1)))

print(dt2.predict(np.array([0, 0, 3881, 0, 0, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0]).reshape(1, -1)))

]

[1]

In [54]:

*# Get a row with poutcome\_success = 1*

*#bank\_with\_dummies[(bank\_with\_dummies.poutcome\_success == 1)]*

data\_drop\_deposite.iloc[985]

In [55]:

*# Predict: Probability for above*

print(dt2.predict\_proba(np.array([46,3354,522,1,1,0,1,0,0.005747,0,0,1,0,0,0,0,1,0,0,0,1,0,0,0,1,0]).reshape(1, -1)))

*#print(ctree.predict(np.array([46,3354,522,1,1,0,1,0,0.005747,0,0,1,0,0,0,0,1,0,0,0,1,0,0,0,1,0]).reshape(1, -1)))*

[[ 0.19295499 0.80704501]]

In [56]:

*# Make predictions on the test set*

preds = dt2.predict(data\_test)

*# Calculate accuracy*

print("**\n**Accuracy score: **\n{}**".format(metrics.accuracy\_score(label\_test, preds)))

*# Make predictions on the test set using predict\_proba*

probs = dt2.predict\_proba(data\_test)[:,1]

*# Calculate the AUC metric*

print("**\n**Area Under Curve: **\n{}**".format(metrics.roc\_auc\_score(label\_test, probs))

# seaborn.heatmap

**seaborn.heatmap(*data*, *\**, *vmin=None*, *vmax=None*, *cmap=None*, *center=None*, *robust=False*, *annot=None*, *fmt='.2g'*, *annot\_kws=None*, *linewidths=0*, *linecolor='white'*, *cbar=True*, *cbar\_kws=None*, *cbar\_ax=None*, *square=False*, *xticklabels='auto'*, *yticklabels='auto'*, *mask=None*, *ax=None*, *\*\*kwargs*)**

Plot rectangular data as a color-encoded matrix.

This is an Axes-level function and will draw the heatmap into the currently-active Axes if none is provided to the ax argument. Part of this Axes space will be taken and used to plot a colormap, unless cbar is False or a separate Axes is provided to cbar\_ax.

**Parameters**

**Data : *rectangular dataset***

2D dataset that can be coerced into an ndarray. If a Pandas DataFrame is provided, the index/column information will be used to label the columns and rows.

**vmin, vmax : *floats, optional***

Values to anchor the colormap, otherwise they are inferred from the data and other keyword arguments.

**Cmap : *matplotlib colormap name or object, or list of colors, optional***

The mapping from data values to color space. If not provided, the default will depend on whether center is set.

**Center : *float, optional***

The value at which to center the colormap when plotting divergant data. Using this parameter will change the default cmap if none is specified.

**Robust : *bool, optional***

If True and vmin or vmax are absent, the colormap range is computed with robust quantiles instead of the extreme values.

**Annot : *bool or rectangular dataset, optional***

If True, write the data value in each cell. If an array-like with the same shape as data, then use this to annotate the heatmap instead of the data. Note that DataFrames will match on position, not index.

**Fmt : *str, optional***

String formatting code to use when adding annotations.

**annot\_kws : *dict of key, value mappings, optional***

Keyword arguments for **[matplotlib.axes.Axes.text()](https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.text.html" \l "matplotlib.axes.Axes.text" \o "(in Matplotlib v3.3.3))** when annot is True.

**linewidths*float, optional***

Width of the lines that will divide each cell.

**Linecolor : *color, optional***

Color of the lines that will divide each cell.

**Cbar : *bool, optional***

Whether to draw a colorbar.

**cbar\_kws : *dict of key, value mappings, optional***

Keyword arguments for **[matplotlib.figure.Figure.colorbar()](https://matplotlib.org/api/_as_gen/matplotlib.figure.Figure.html" \l "matplotlib.figure.Figure.colorbar" \o "(in Matplotlib v3.3.3))**.

**cbar\_ax : *matplotlib Axes, optional***

Axes in which to draw the colorbar, otherwise take space from the main Axes.

**Square : *bool, optional***

If True, set the Axes aspect to “equal” so each cell will be square-shaped.

**xticklabels, yticklabels : *“auto”, bool, list-like, or int, optional***

If True, plot the column names of the dataframe. If False, don’t plot the column names. If list-like, plot these alternate labels as the xticklabels. If an integer, use the column names but plot only every n label. If “auto”, try to densely plot non-overlapping labels.

**Mask : *bool array or DataFrame, optional***

If passed, data will not be shown in cells where mask is True. Cells with missing values are automatically masked.

**Ax : *matplotlib Axes, optional***

Axes in which to draw the plot, otherwise use the currently-active Axes.

**Kwargs : *other keyword arguments***

All other keyword arguments are passed to **[matplotlib.axes.Axes.pcolormesh()](https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.pcolormesh.html" \l "matplotlib.axes.Axes.pcolormesh" \o "(in Matplotlib v3.3.3))**.

**Returns**

**Returns: Ax : *matplotlib Axes***

Axes object with the heatmap

0.7880