Implementation of Decision Tree Classification in Python

Dataset URL: https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database

As a marketing manager, you want a set of customers who are most likely to purchase your product.

This is how you can save your marketing budget by finding your audience.

As a loan manager, you need to identify risky loan applications to achieve a lower loan default rate.

This process of classifying customers into a group of potential and non-potential customers or safe or risky loan applications is known as a classification problem.

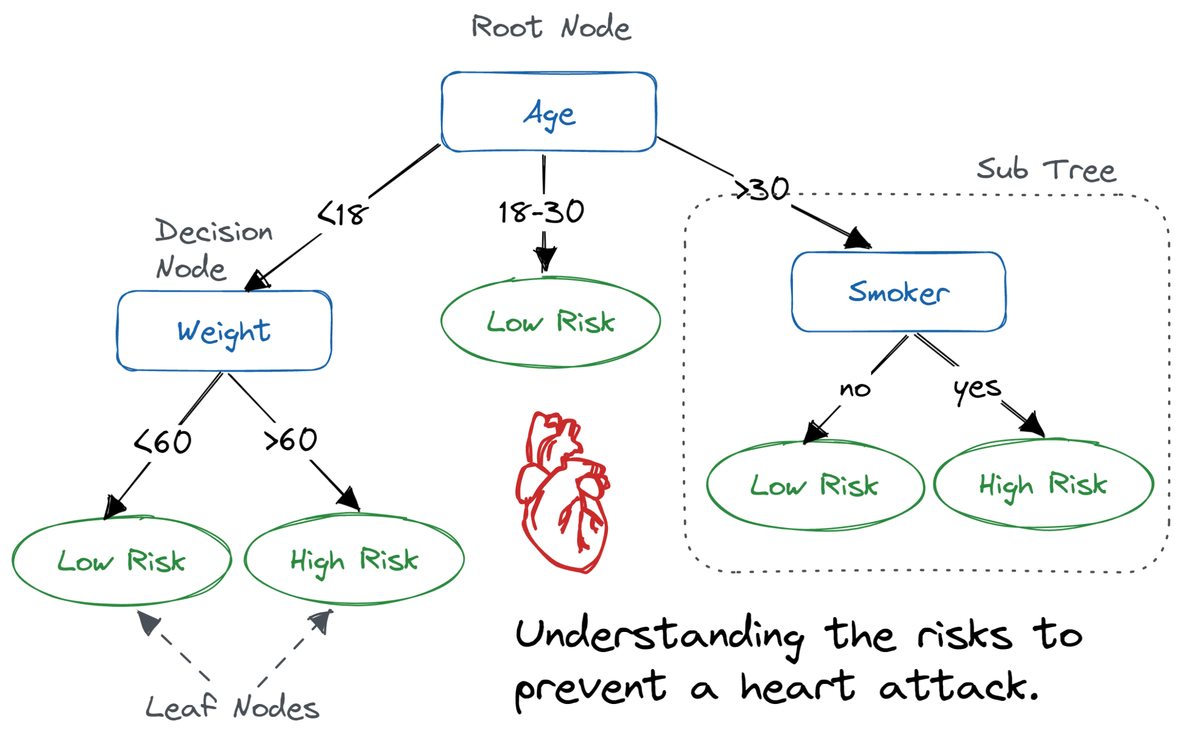
Classification is a two-step process; a learning step and a prediction step. In the learning step, the model is developed based on given training data. In the prediction step, the model is used to predict the response to given data.

A Decision tree is one of the easiest and most popular classification algorithms used to understand and interpret data. It can be utilized for both classification and regression problems.

The Decision Tree Algorithm

A decision tree is a flowchart-like tree structure where an internal node represents a feature(or attribute), the branch represents a decision rule, and each leaf node represents the outcome.

The topmost node in a decision tree is known as the root node. It learns to partition on the basis of the attribute value. It partitions the tree in a recursive manner called recursive partitioning. This flowchart-like structure helps you in decision-making. It's visualization like a flowchart diagram which easily mimics the human level thinking. That is why decision trees are easy to understand and interpret.



A decision tree is a white box type of ML algorithm. It shares internal decision-making logic, which is not available in the black box type of algorithms such as with a **neural network**. Its training time is faster compared to the neural network algorithm.

The time complexity of decision trees is a function of the number of records and attributes in the given data. The decision tree is a distribution-free or non-parametric method which does not depend upon probability distribution assumptions. Decision trees can handle high-dimensional data with good accuracy.

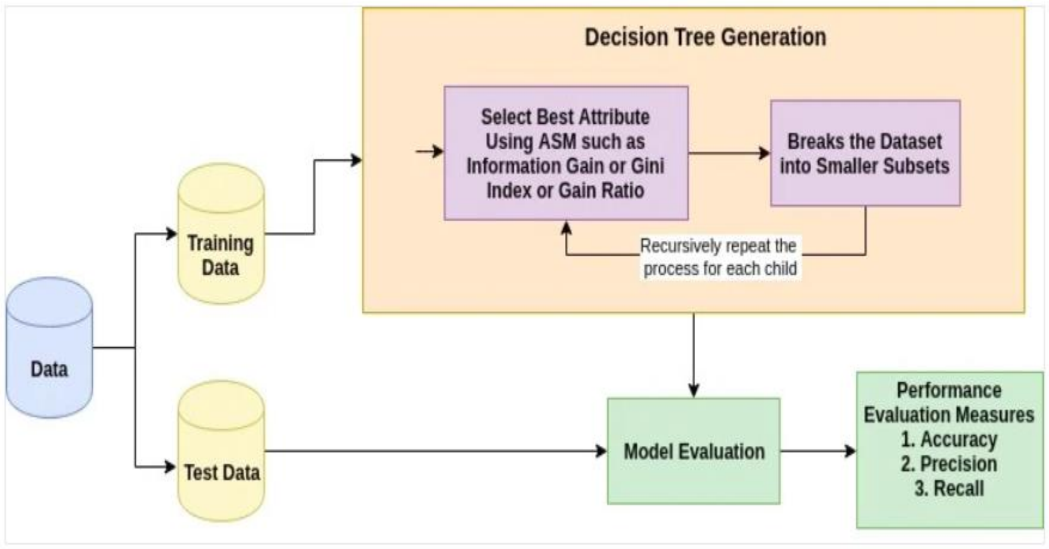
How Does the Decision Tree Algorithm Work?

The basic idea behind any decision tree algorithm is as follows:

1. Select the best attribute using Attribute Selection Measures (ASM) to split the records.

2. Make that attribute a decision node and breaks the dataset into smaller subsets.
3. Start tree building by repeating this process recursively for each child until one of the conditions will match:

- All the tuples belong to the same attribute value.
- There are no more remaining attributes.
- There are no more instances.


Decision Tree Generation
Select Best Attribute Using ASM such as Information Gain or Gini Index or Gain Ratio
Breaks the Dataset into Smaller Subsets
Recursively repeat the process for each child
Training Data
Data
Test Data
Model Evaluation
Performance Evaluation Measures
1. Accuracy
2. Precision
3. Recall

Attribute Selection Measures

Attribute selection measure is a heuristic for selecting the splitting criterion that partitions data in the best possible manner. It is also known as splitting rules because it helps us to determine breakpoints for tuples on a given node. ASM provides a rank to each feature (or attribute) by explaining the given dataset. The best score attribute will be selected as a splitting attribute (). In the case of a continuous-valued attribute, split points for branches also need to define.

The most popular selection measures are Information Gain, Gain Ratio, and Gini Index.

Information Gain

Claude Shannon invented the concept of entropy, which measures the impurity of the input set. In physics and mathematics, entropy is referred to as the randomness or the impurity in a system. In information theory, it refers to the impurity in a group of examples. **Information gain is the decrease in entropy. Information gain computes the difference between entropy before the split and average entropy after the split of the dataset based on given attribute values.**

ID3 (Iterative Dichotomiser) decision tree algorithm uses information gain.

$$\text{Info}(D) = -\sum_{i=1}^{m} pi \log_2 pi$$

Where Pi is the probability that an arbitrary tuple in D belongs to class Ci.

$$\text{Info}_A(D) = \sum_{j=1}^{V} \frac{|Dj|}{|D|} \times \text{Info}(D_j)$$

$$\text{Gain}(A) = \text{Info}(D) - \text{Info}_A(D)$$

Where:

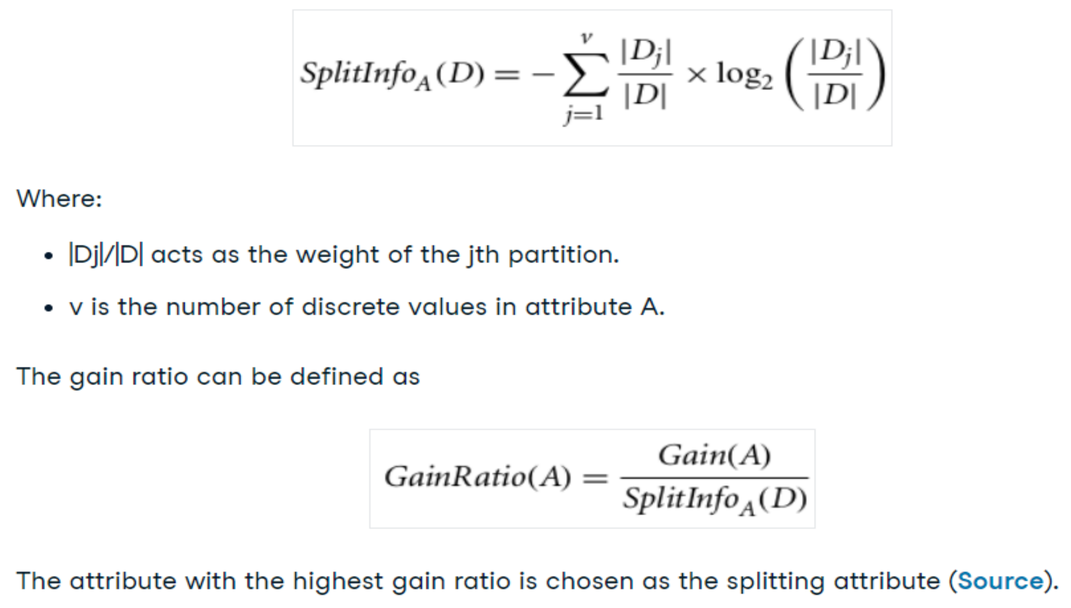
- Info(D) is the average amount of information needed to identify the class label of a tuple in D.
- |Dj|/|D| acts as the weight of the jth partition.
- InfoA(D) is the expected information required to classify a tuple from D based on the partitioning by A.

The attribute A with the highest information gain, Gain(A), is chosen as the splitting attribute at node N().

Gain Ratio

Information gain is biased for the attribute with many outcomes. It means it prefers the attribute with a large number of distinct values. For instance, consider an attribute with a unique identifier, such as customer_ID, that has zero info(D) because of pure partition. This maximizes the information gain and creates useless partitioning.

C4.5, an improvement of ID3, uses an extension to information gain known as the gain ratio. Gain ratio handles the issue of bias by normalizing the information gain using Split Info. Java implementation of the C4.5 algorithm is known as J48, which is available in WEKA data mining tool.



$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2\left(\frac{|D_j|}{|D|}\right)$$

Where:

- |Dj|/|D| acts as the weight of the jth partition.
- v is the number of discrete values in attribute A.

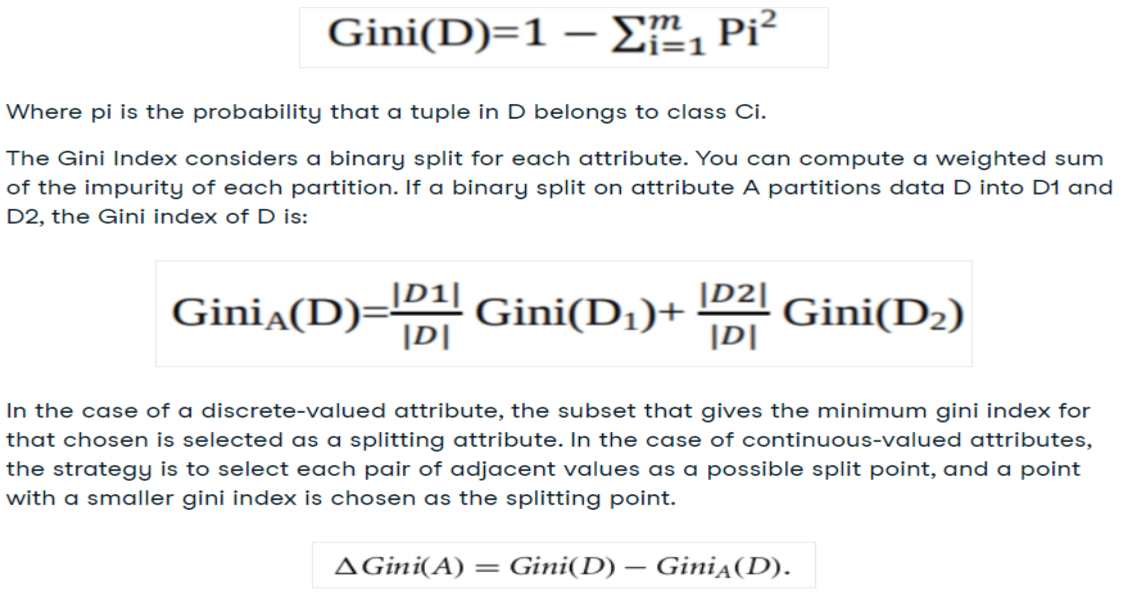
The gain ratio can be defined as

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo_A(D)}$$

The attribute with the highest gain ratio is chosen as the splitting attribute (Source).

Gini index

Another decision tree algorithm CART (Classification and Regression Tree) uses the Gini method to create split points.



$$\mathrm{Gini}(D)=1-\sum_{i=1}^{m} Pi^2$$

Where pi is the probability that a tuple in D belongs to class Ci.

The Gini Index considers a binary split for each attribute. You can compute a weighted sum of the impurity of each partition. If a binary split on attribute A partitions data D into D1 and D2, the Gini index of D is:

$$\mathrm{Gini}_A(D)=\frac{|D1|}{|D|}\,\mathrm{Gini}(D_1)+\frac{|D2|}{|D|}\,\mathrm{Gini}(D_2)$$

In the case of a discrete-valued attribute, the subset that gives the minimum gini index for that chosen is selected as a splitting attribute. In the case of continuous-valued attributes, the strategy is to select each pair of adjacent values as a possible split point, and a point with a smaller gini index is chosen as the splitting point.

$$\Delta Gini(A) = Gini(D) - Gini_A(D).$$

The attribute with the minimum Gini index is chosen as the splitting attribute.

Decision Tree Classifier Building in Scikit-learn

Importing Required Libraries

Let's first load the required libraries.

```
# Load libraries

import pandas as pd

from sklearn.tree import DecisionTreeClassifier # Import Decision Tree Classifier

from sklearn.model_selection import train_test_split # Import train_test_split
function

from sklearn import metrics #Import scikit-learn metrics module for accuracy
calculation
```

EXPLANATION:

This code imports necessary libraries for building a decision tree classifier model.

• **pandas** is a library used for data manipulation and analysis.

• **DecisionTreeClassifier** is a class from the **sklearn.tree** module that is used to build a decision tree classifier model.

• **train_test_split** is a function from the **sklearn.model_selection** module that is used to split the dataset into training and testing sets.

• **metrics** is a module from the **sklearn** library that provides various metrics for evaluating the performance of a machine learning model.

• By importing these libraries, the user can use their functions and classes to build and evaluate a decision tree classifier model.

Loading Data

Let's first load the required Pima Indian Diabetes dataset using pandas' read CSV function. You can download the **dataset** from

https://www.kaggle.com/datasets/uciml/pima-indians-diabetes-database

to follow along.

```
col_names = ['pregnant', 'glucose', 'bp', 'skin', 'insulin', 'bmi', 'pedigree', 'age',
'label']

# load dataset

pima = pd.read_csv("diabetes.csv", header=None, names=col_names)
```

EXPLANATION:

This code creates a list of column names called **col_names** which will be used to label the columns of a dataset.

• Then, it loads a dataset called "diabetes.csv" into a Pandas DataFrame called **pima**.

• The **header=None** argument specifies that the dataset does not have a header row, and the **names=col_names** argument assigns the column names from the **col_names** list to the DataFrame.

```
pima.head()
```

EXPLANATION:

This code is written in Python.

• The **pima.head()** function is used to display the first few rows of the dataset **pima**.

• This is useful for getting a quick overview of the data and checking if it has been loaded correctly.

• By default, the **head()** function displays the first 5 rows of the dataset, but you can specify a different number of rows to display by passing an integer argument to the function.

	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0

	pregnant	glucose	bp	skin	insulin	bmi	pedigree	age	label
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Feature Selection

Here, you need to divide given columns into two types of variables dependent(or target variable) and independent variable(or feature variables).

```
#split dataset in features and target variable

feature_cols = ['pregnant', 'insulin', 'bmi', 'age','glucose','bp','pedigree']

X = pima[feature_cols] # Features

y = pima.label # Target variable
```

EXPLANATION:

This code is written in Python and it is used to split a dataset into features and target variable.

• The first line defines a list of feature columns that will be used to create the feature matrix.

• The list contains the names of the columns that will be used as features, which are 'pregnant', 'insulin', 'bmi', 'age', 'glucose', 'bp', and 'pedigree'.

• The second line creates a feature matrix X by selecting the columns specified in the feature_cols list from the pima dataset.

• The feature matrix X will contain the values of the selected columns for each observation in the dataset.

• The third line creates a target variable y by selecting the 'label' column from the pima dataset.

• The target variable y will contain the values of the 'label' column for each observation in the dataset.

• Overall, this code is used to prepare the data for machine learning by separating the features and target variable into separate variables.

Splitting Data

To understand model performance, dividing the dataset into a training set and a test set is a good strategy.

Let's split the dataset by using the function train_test_split(). You need to pass three parameters features; target, and test_set size.

```
# Split dataset into training set and test set

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=1) # 70% training and 30% test
```

EXPLANATION:

This code uses the **train_test_split** function from the **sklearn.model_selection** module to split a dataset into a training set and a test set.

• The **X** and **y** variables represent the features and target variable of the dataset, respectively.

• The **test_size** parameter is set to 0.3, which means that 30% of the data will be used for testing and 70% will be used for training.

• The **random_state** parameter is set to 1, which ensures that the same random split is generated each time the code is run.

• The function returns four arrays: **X_train**, **X_test**, **y_train**, and **y_test**.

• **X_train** and **y_train** represent the training set, while **X_test** and **y_test** represent the test set.

• These arrays can be used to train and evaluate a machine learning model.

Building Decision Tree Model

Let's create a decision tree model using Scikit-learn.

```
# Create Decision Tree classifer object

clf = DecisionTreeClassifier()



# Train Decision Tree Classifer

clf = clf.fit(X_train,y_train)



#Predict the response for test dataset

y_pred = clf.predict(X_test)
```

EXPLANATION:

This code is written in Python and it creates a decision tree classifier object using the **DecisionTreeClassifier()** function.

• Then, it trains the classifier using the **fit()** method with the training data **X_train** and **y_train**.

• Finally, it uses the trained classifier to predict the response for the test dataset **X_test** and stores the predictions in **y_pred**.

Evaluating the Model

Let's estimate how accurately the classifier or model can predict the type of cultivars.

Accuracy can be computed by comparing actual test set values and predicted values.

```
# Model Accuracy, how often is the classifier correct?

print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

EXPLANATION:

This code snippet is written in Python.

• The **metrics.accuracy_score()** function is used to calculate the accuracy of a classification model.

• It takes two arguments: **y_test** and **y_pred**.

• **y_test** is the true labels of the test set, and **y_pred** is the predicted labels of the test set.

• The **print()** function is used to display the accuracy score on the console.

• The output will be a string that says "Accuracy:" followed by the actual accuracy score.

```
Accuracy: 0.6753246753246753
```

EXPLANATION:

This code snippet is simply displaying the accuracy score of a model.

• The value 0.6753246753246753 represents the accuracy score, which is a metric used to evaluate the performance of a machine learning model.

• The higher the accuracy score, the better the model is at making correct predictions.

• There is no specific programming language mentioned in this code snippet, as it is simply displaying a numerical value.

We got a classification rate of 67.53%, which is considered as good accuracy. You can improve this accuracy by tuning the parameters in the decision tree algorithm.

Visualizing Decision Trees

You can use Scikit-learn's *export_graphviz* function for display the tree within a Jupyter notebook. For plotting the tree, you also need to install graphviz and pydotplus.

```
pip install graphviz
```

```
pip install pydotplus
```

The *export_graphviz* function converts the decision tree classifier into a dot file, and pydotplus converts this dot file to png or displayable form on Jupyter.

```
from sklearn.tree import export_graphviz

from sklearn.externals.six import StringIO
from IPython.display import Image

import pydotplus



dot_data = StringIO()

export_graphviz(clf, out_file=dot_data,

                filled=True, rounded=True,

                special_characters=True,feature_names = 
feature_cols,class_names=['0','1'])

graph = pydotplus.graph_from_dot_data(dot_data.getvalue())

graph.write_png('diabetes.png')

Image(graph.create_png())
```

EXPLANATION:

This code snippet is used to visualize a decision tree model created using scikit-learn's **clf** object.

• First, the necessary libraries are imported: **export_graphviz** from **sklearn.tree**, **StringIO** from **sklearn.externals.six**, **Image** from **IPython.display**, and **pydotplus**.

• Then, a **StringIO** object is created to store the dot data generated by **export_graphviz**.

• The **export_graphviz** function is called with the **clf** object as the first argument and various parameters to customize the appearance of the tree.

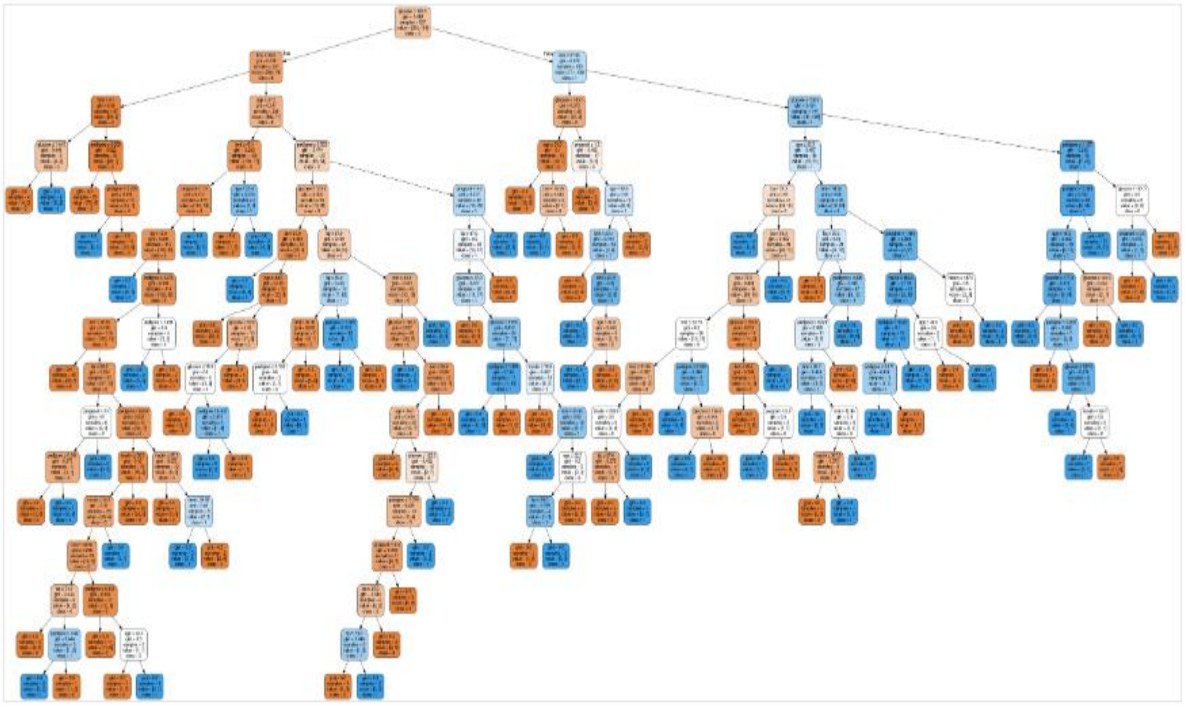
• The dot data is then written to the **StringIO** object.

• Next, **pydotplus** is used to create a graph from the dot data stored in the **StringIO** object.

• The resulting graph is saved as a PNG file named "diabetes.png".

• Finally, the graph is displayed using **Image** from **IPython.display**.

• Overall, this code generates a visual representation of a decision tree model, which can be useful for understanding how the model makes predictions and identifying areas for improvement.



In the decision tree chart, each internal node has a decision rule that splits the data. Gini, referred to as Gini ratio, measures the impurity of the node. You can say a node is pure when all of its records belong to the same class, such nodes known as the leaf node.

Here, the resultant tree is unpruned. This unpruned tree is unexplainable and not easy to understand. In the next section, let's optimize it by pruning.

Optimizing Decision Tree Performance

- **criterion : optional (default=”gini”) or Choose attribute selection measure.** This parameter allows us to use the different-different attribute selection measure. Supported criteria are “gini” for the Gini index and “entropy” for the information gain.
- **splitter : string, optional (default=”best”) or Split Strategy.** This parameter allows us to choose the split strategy. Supported strategies are “best” to choose the best split and “random” to choose the best random split.
- **max_depth : int or None, optional (default=None) or Maximum Depth of a Tree.** The maximum depth of the tree. If None, then nodes are expanded until all the leaves contain less than min_samples_split samples. The higher value of maximum depth causes overfitting, and a lower value causes underfitting.

In Scikit-learn, optimization of decision tree classifier performed by only pre-pruning. Maximum depth of the tree can be used as a control variable for pre-pruning. In the following the example, you can plot a decision tree on the same data with max_depth=3. Other than pre-pruning parameters, You can also try other attribute selection measure such as entropy.

```
# Create Decision Tree classifer object

clf = DecisionTreeClassifier(criterion="entropy", max_depth=3)



# Train Decision Tree Classifer

clf = clf.fit(X_train,y_train)



#Predict the response for test dataset

y_pred = clf.predict(X_test)



# Model Accuracy, how often is the classifier correct?
```

```
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

EXPLANATION:

This code is used to create a decision tree classifier and train it on a dataset.

• First, the DecisionTreeClassifier class is imported and an object of this class is created with the criterion set to "entropy" and the maximum depth of the tree set to 3.

• Next, the fit() method is called on the classifier object with the training data (X_train and y_train) as arguments.

• This trains the decision tree on the provided data.

• After training, the predict() method is called on the classifier object with the test data (X_test) as an argument.

• This generates predictions for the test data based on the trained decision tree.

• Finally, the accuracy of the classifier is calculated using the accuracy_score() method from the metrics module and printed to the console.

• This gives an indication of how well the decision tree is able to classify the test data.

```
Accuracy: 0.7705627705627706
```

EXPLANATION:

This code snippet is not actually a code, but rather a result of a model's accuracy.

• The number 0.7705627705627706 represents the accuracy of the model, which is a measure of how well the model is able to predict the correct outcome.

• The higher the accuracy, the better the model is at making predictions.

• This result could have been obtained by running a machine learning model on a dataset and evaluating its performance.

Well, the classification rate increased to 77.05%, which is better accuracy than the previous model.

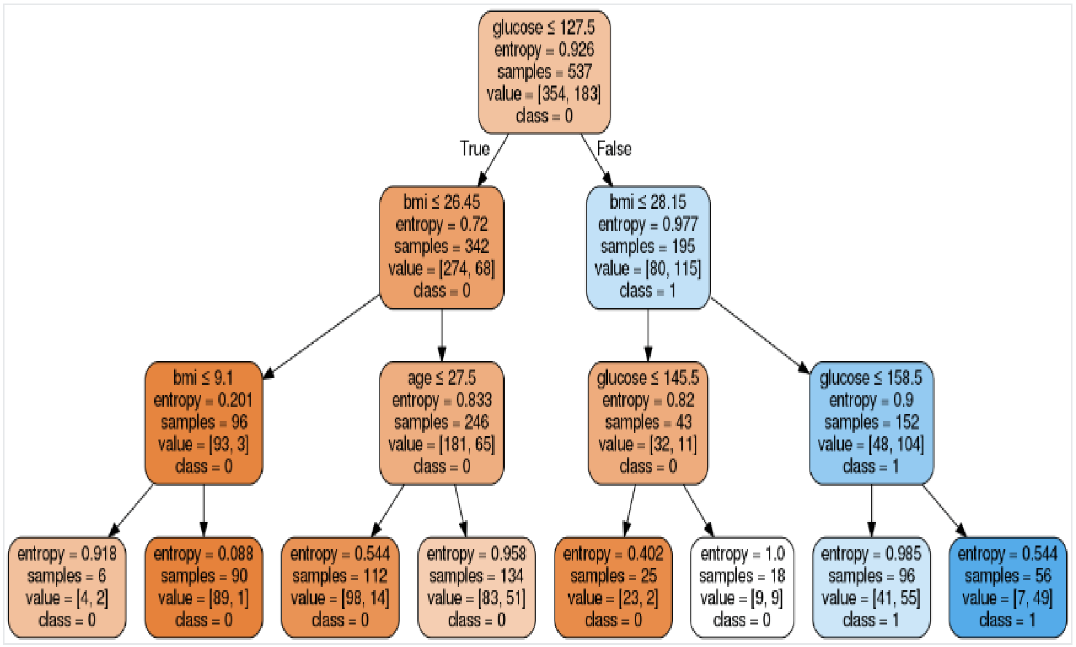
Visualizing Decision Trees

Let's make our decision tree a little easier to understand using the following code:

```
from six import StringIO from IPython.display import Image from sklearn.tree
import export_graphviz import pydotplus dot_data = StringIO()
export_graphviz(clf, out_file=dot_data, filled=True, rounded=True,
special_characters=True, feature_names = feature_cols,class_names=['0','1']) graph
= pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('diabetes.png') Image(graph.create_png())
```

Here, we've completed the following steps:

- Imported the required libraries.
- Created a StringIO object called dot_data to hold the text representation of the decision tree.
- Exported the decision tree to the dot format using the export_graphviz function and write the output to the dot_data buffer.
- Created a pydotplus graph object from the dot format representation of the decision tree stored in the dot_data buffer.
- Written the generated graph to a PNG file named "diabetes.png".
- Displayed the generated PNG image of the decision tree using the Image object from the IPython.display module.



As you can see, this pruned model is less complex, more explainable, and easier to understand than the previous decision tree model plot.

Decision Tree Pros

- Decision trees are easy to interpret and visualize.
- It can easily capture Non-linear patterns.
- It requires fewer data preprocessing from the user, for example, there is no need to normalize columns.
- It can be used for feature engineering such as predicting missing values, suitable for variable selection.
- The decision tree has no assumptions about distribution because of the non-parametric nature of the algorithm.

Decision Tree Cons

- Sensitive to noisy data. It can overfit noisy data.

- The small variation(or variance) in data can result in the different decision tree. This can be reduced by bagging and boosting algorithms.
- Decision trees are biased with imbalance dataset, so it is recommended that balance out the dataset before creating the decision tree.