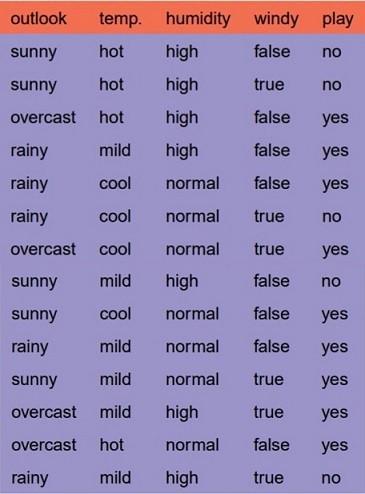
**FIT 5202 – Data Processing for Big Data**

**Activity: Classification Using Decision Tree and Random Forest**

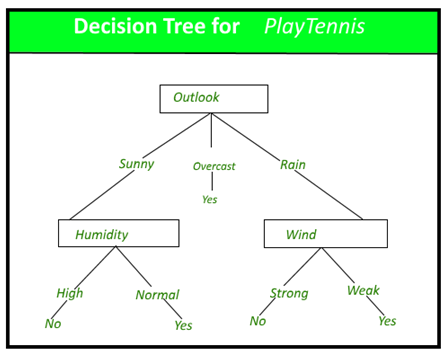
**Exercise 1:**

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

In this exercise, we will be using Apache spark to create a decision tree. Basically the datasets lists (Figure 1) the conditions which impacts if a game of tennis can be played outside or not. The values of *outlook, temperature, humidity* and *wind* are described and *outcome* that the game was played or not under these conditions. We will build the decision tree like Figure 2.



**Figure 1 Dataset for Decision Tree**

****

**Figure 2 Decision Tree**

We will go through the code and explain which part of the code is doing what in the codebase. (To run the following code create a new notebook.)

**1. Include the required library**

|  |
| --- |
| from \_\_future\_\_ import division  import operator  from pyspark import SparkContext  from pyspark.sql import SQLContext  from pyspark.sql.functions import col  import numpy as np  import networkx as nx  from matplotlib import pyplot as plt |

**2. Instantiate the spark context**

|  |
| --- |
| sc = SparkContext(master="local", appName="Decision Tree")  sqlContext = SQLContext(sc)  attr\_name\_info\_gain = {}  G = nx.DiGraph() |

**3. Declare schema of the dataset. I have created 2 variables for storing the schema and types.**

|  |
| --- |
| attrs = ["outlook","temp","humidity","wind"]  attrs\_type = {"outlook":"string","temp":"string","humidity":"string","wind":"string"} |

**4. Calculate the Information gain**

|  |
| --- |
| def calculate\_info\_gain(entropy, joined\_df, total\_elements):  attr\_entropy = 0.0  for anAttributeData in joined\_df.rdd.collect():  yes\_class\_count = anAttributeData[1]  no\_class\_count = anAttributeData[2]  if yes\_class\_count is None:  yes\_class\_count = 0  elif no\_class\_count is None:  no\_class\_count = 0  count\_of\_class = yes\_class\_count + no\_class\_count  # do the summation part e.g. if age is 56, 60, 45 then its sum of entropy for each of these element  classmap = {'y' : yes\_class\_count, 'n' : no\_class\_count}  attr\_entropy = attr\_entropy + ((count\_of\_class / total\_elements) \*\  calculate\_entropy(count\_of\_class, classmap))  gain = entropy - attr\_entropy  return gain |

**4. Attribute information gain data preparation function**

|  |
| --- |
| def get\_attr\_info\_gain\_data\_prep(attr\_name, data, entropy, total\_elements, where\_condition):  if not where\_condition:  attr\_grp\_y = data.where(col('y') == 'yes').groupBy(attr\_name).agg({"y": 'count'})\  .withColumnRenamed('count(y)','played\_count')  else:  attr\_grp\_y = data.where(" y like '%yes%' " + where\_condition).groupBy(attr\_name).agg({"y": 'count'})\  .withColumnRenamed('count(y)','played\_count')  if not where\_condition:  attr\_grp\_n = data.where(col('y') == 'no').groupBy(attr\_name).agg({"y": 'count'})\  .withColumnRenamed(attr\_name,'n\_' + attr\_name)\  .withColumnRenamed('count(y)','not\_played\_count')  else:  attr\_grp\_n = data.where(" y like '%no%' " + where\_condition).groupBy(attr\_name).agg({"y": 'count'})\  .withColumnRenamed(attr\_name,'n\_' + attr\_name)\  .withColumnRenamed('count(y)','not\_played\_count')  joined\_df = attr\_grp\_y.join(attr\_grp\_n, on = [col(attr\_grp\_y.columns[0]) == col(attr\_grp\_n.columns[0])], how='outer' )\  .withColumn("total", col(attr\_grp\_y.columns[0]) + col(attr\_grp\_n.columns[0]))\  .select(attr\_grp\_y.columns[0], attr\_grp\_y.columns[1],\  attr\_grp\_n.columns[1]) \  gain\_for\_attribute = calculate\_info\_gain(entropy, joined\_df, total\_elements)  attr\_name\_info\_gain[attr\_name] = gain\_for\_attribute |

**5. Calculate the entropy of the elements**

|  |
| --- |
| def calculate\_entropy(total\_elements, elements\_in\_each\_class):  # for target set S having 2 class 0 and 1, the entropy is -p0logp0 -p1logp1  # here the log is of base 2  # elements\_in\_each\_class is a dictionary where the key is class label and the  # value is number of elements in that class  keysInMap = list(elements\_in\_each\_class.keys())  entropy = 0.0  for aKey in keysInMap:  number\_of\_elements\_in\_class = elements\_in\_each\_class.get(aKey)  if number\_of\_elements\_in\_class == 0:  continue  ratio = number\_of\_elements\_in\_class/total\_elements  entropy = entropy - ratio \* np.log2(ratio)  return entropy |

It reads data from the file. As we build the tree, we will need to get data corresponding to that branch of the tree only. The ‘where\_condition’ attribute will contain these predicates.

We group the records in the file which have outcome as ‘yes’ for the attribute names passed

* For first time, the where\_condition will be blank,
* Second iteration onwards, after root of the tree is found, we will have where\_condition

**6. Process the data**

|  |
| --- |
| def process\_dataset(excludedAttrs, data, played, notplayed, where\_condition):  total\_elements = played + notplayed  subs\_info = {"played" : played, "notplayed" : notplayed}  entropy = calculate\_entropy(total\_elements, subs\_info)  print ("entropy is " + str(entropy))  global attr\_name\_info\_gain  attr\_name\_info\_gain = dict()  for attr in attrs:  if attr not in excludedAttrs:  get\_attr\_info\_gain\_data\_prep(attr, data, entropy, total\_elements, where\_condition) |
|  |

* excludedAtttts will contain the list of attributes which are already processed so that we dont need to process again.
* data is the spark dataframe for this file
* played — count when match was played
* notplayed — count when match was not played
* Where\_condition — condition used to select the data, as and when attributes are processed we will keep chaging this condition

**7. Build the Tree**

|  |
| --- |
| def build\_tree(max\_gain\_attr, processed\_attrs, data, where\_condition):  attrValues = sqlContext.sql("select distinct " + max\_gain\_attr + " from data where 1==1 " + where\_condition)  orig\_where\_condition = where\_condition  for aValueForMaxGainAttr in attrValues.rdd.collect():  adistinct\_value\_for\_attr = aValueForMaxGainAttr[0]  G.add\_edges\_from([(max\_gain\_attr, adistinct\_value\_for\_attr)])  if attrs\_type[max\_gain\_attr] == "string":  where\_condition = str(orig\_where\_condition + " and " + max\_gain\_attr + "=='" + adistinct\_value\_for\_attr + "'")  else:  where\_condition = str(orig\_where\_condition + " and " + max\_gain\_attr + "==" + adistinct\_value\_for\_attr)  played\_for\_attr = sqlContext.sql("select \* from data where y like '%yes%' " + where\_condition).count()  notplayed\_for\_attr = sqlContext.sql("select \* from data where y like '%no%' " + where\_condition).count()  # if either has zero value then entropy for this attr will be zero and its the last attr in the tree  leaf\_values = []  if played\_for\_attr == 0 or notplayed\_for\_attr == 0:  leaf\_node = sqlContext.sql("select distinct y from data where 1==1 " + where\_condition)  for leaf\_node\_data in leaf\_node.rdd.collect():  G.add\_edges\_from([(adistinct\_value\_for\_attr, str(leaf\_node\_data[0]))])  continue  process\_dataset(processed\_attrs, data, played\_for\_attr, notplayed\_for\_attr, where\_condition)  if not attr\_name\_info\_gain: # we processed all attributes  # attach leaf node  leaf\_node = sqlContext.sql("select distinct y from data where 1==1 " + where\_condition)  for leaf\_node\_data in leaf\_node.rdd.collect():  G.add\_edges\_from([(adistinct\_value\_for\_attr, str(leaf\_node\_data[0]))])  continue # we are done for this branch of tree  # get the attr with max info gain under aValueForMaxGainAttr  # sort by info gain  sorted\_by\_info\_gain = sorted(attr\_name\_info\_gain.items(), key=operator.itemgetter(1), reverse=True)  new\_max\_gain\_attr = sorted\_by\_info\_gain[0][0]  if sorted\_by\_info\_gain[0][1] == 0:  # under this where condition, records dont have entropy  leaf\_node = sqlContext.sql("select distinct y from data where 1==1 " + where\_condition)  # there might be more than one leaf node  for leaf\_node\_data in leaf\_node.rdd.collect():  G.add\_edges\_from([(adistinct\_value\_for\_attr, str(leaf\_node\_data[0]))])  continue # we are done for this branch of tree  G.add\_edges\_from([(adistinct\_value\_for\_attr, new\_max\_gain\_attr)])  processed\_attrs.append(new\_max\_gain\_attr)  build\_tree(new\_max\_gain\_attr, processed\_attrs, data, where\_condition) |

**8. Load the dataset and draw the graph**

Download the “simple\_dataset” file from LMS and put it in “/home/student/FIT5202/” directory .

|  |
| --- |
| def main():  data = sqlContext.read.format('com.databricks.spark.csv').option('header', 'true')\  .option('delimiter', ';').load("/home/student/FIT5202/simple\_dataset")  data.registerTempTable('data')  played = sqlContext.sql("select \* from data WHERE y like '%y%' ").count()  notplayed = sqlContext.sql("select \* from data WHERE y like '%n%' ").count()  process\_dataset([], data, played, notplayed, '')  # sort by info gain  sorted\_by\_info\_gain = sorted(attr\_name\_info\_gain.items(), key=operator.itemgetter(1), reverse=True)  processed\_attrs = []  max\_gain\_attr = sorted\_by\_info\_gain[0][0]  processed\_attrs.append(max\_gain\_attr)  build\_tree(max\_gain\_attr, processed\_attrs, data, '')  nx.draw(G, with\_labels=True)  plt.tight\_layout()  plt.savefig("Graph.png", format="png")  main() |

**Exercise 2: Compare the accuracy of Decision Tree vs Random Forest algorithm**

In this exercise, we will be using the same dataset(e.g., iris.data) and run both decision tree and random forest algorithms and calculate their accuracy.

**1. Load library and instantiate the environment**

|  |
| --- |
| from pyspark import SparkContext  from pyspark.sql import SQLContext  from pyspark.sql.session import SparkSession  from pyspark.sql.functions import col  sc = SparkContext("local", "Decision Tree Feature")  spark = SparkSession(sc) |

**2. Data Loading**

Download the “iris.data” file from LMS and put it in “/home/student/FIT5202/” directory .

|  |
| --- |
| df = spark.read.csv("iris.data", inferSchema=True)\  .toDF("sep\_len", "sep\_wid", "pet\_len", "pet\_wid", "label")  df.show(5) |

**3. Data Processing**

|  |
| --- |
| from pyspark.ml.linalg import Vectors  from pyspark.ml.feature import VectorAssembler  vector\_assembler = VectorAssembler(\  inputCols=["sep\_len", "sep\_wid", "pet\_len", "pet\_wid"],\  outputCol="features")  df\_temp = vector\_assembler.transform(df)  df\_temp.show(3)  df = df\_temp.drop('sep\_len', 'sep\_wid', 'pet\_len', 'pet\_wid')  df.show(3)  from pyspark.ml.feature import StringIndexer  l\_indexer = StringIndexer(inputCol="label", outputCol="labelIndex")  df = l\_indexer.fit(df).transform(df) |

**4. Splitting the dataset between training and test dataset**

|  |
| --- |
| (trainingData, testData) = df.randomSplit([0.7, 0.3]) |

**5. Run and calculate the accuracy of decision tree algorithm**

|  |
| --- |
| from pyspark.ml.classification import DecisionTreeClassifier  from pyspark.ml.evaluation import MulticlassClassificationEvaluator  dt = DecisionTreeClassifier(labelCol="labelIndex", featuresCol="features")  model = dt.fit(trainingData)  predictions = model.transform(testData)  predictions.select("prediction", "labelIndex").show(5)  evaluator = MulticlassClassificationEvaluator(\  labelCol="labelIndex", predictionCol="prediction",\  metricName="accuracy")  accuracy = evaluator.evaluate(predictions)  print("Test Error = %g " % (1.0 - accuracy)) |

**5. Run and calculate the accuracy of random forest algorithm**

|  |
| --- |
| from pyspark.ml.classification import RandomForestClassifier  rf = RandomForestClassifier(labelCol="labelIndex",\  featuresCol="features", numTrees=10)  model = rf.fit(trainingData)  predictions = model.transform(testData)  predictions.select("prediction", "labelIndex").show(5)  evaluator =\  MulticlassClassificationEvaluator(labelCol="labelIndex",\  predictionCol="prediction", metricName="accuracy")  accuracy = evaluator.evaluate(predictions)  print("Test Error = %g" % (1.0 - accuracy)) |