CSB 310: Artificial Intelligence

Lab 10: Machine Learning

Submitted By:

Name: **UMANG KUMAR**

Roll No: **201210051**

Branch: **CSE**

Semester: **5th Sem**

Submitted To: ***Dr. Chandra Prakash***

Release Date: 21/11/2022 **Submitted Date:** 26/11/2022

NATIONAL INSTITUTE OF TECHNOLOGY DELHI

Department of Computer Science and Engineering

2022

PART A: Exposition Problems

1. Consider the below dataset as discussed in class with 4 attributes/features and two class (Play and not Play Tennis). Write a program to classify a new sample X using Bayesian Classifier when :

PlayTennis: training examples

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

- outlook = sunny, temperature = cool, humidity = high and windy = false
- X = rain, hot, high, false

Compare the your results with the in-build library of Bayesian Classifier in python .

Output:

Using built-in Libraries:

```
predicted_1 = model.predict([[2, 0, 0, 1]]) # Sunny, Cool, High, Weak
predicted_2 = model.predict([[1, 1, 0, 1]]) # Rain, Hot, High, Weak

print(predicted_1)
print(predicted_2)
```

```
[0]
[1]
```

From the above output we can say that for the first case we can play tennis while in the second we cannot.

Using Custom Methods

```
# Sunny, Cool, High, Weak

# [P(Sunny|Yes)P(Cool|Yes)P(High|Yes)P(Weak|Yes)]P(Play=Yes)
predicted_1 = outlook_learning[("Sunny", "Yes")] * temperature_learning[("Cool", "Yes")] * humidity_learning[("High", "Yes")] * wind_learning[("Weak", "Yes")] * play_pr
# [P(Sunny|No)P(Cool|No)P(High|No)P(Weak|No)]P(Play=No)
predicted_2 = outlook_learning[("Sunny", "No")] * temperature_learning[("Cool", "No")] * humidity_learning[("High", "No")] * wind_learning[("Weak", "No")] * no_play_prc

print("P(Yes|x'):", predicted_1)
print("P(No|x'): ", predicted_2)
if predicted_2 > predicted_1:
  print("Since P(No|x') > P(Yes|x'), resultant is No: ", [0])
else:
  print("Since P(Yes|x') > P(No|x'), resultant is Yes: ", [1])
print()

# Rain, Hot, High, Weak

# [P(Rain|Yes)P(Hot|Yes)P(High|Yes)P(Weak|Yes)]P(Play=Yes)
predicted_1 = outlook_learning[("Rain", "Yes")] * temperature_learning[("Hot", "Yes")] * humidity_learning[("High", "Yes")] * wind_learning[("Weak", "Yes")] * play_prob
# [P(Rain|No)P(Hot|No)P(High|No)P(Weak|No)]P(Play=No)
predicted_2 = outlook_learning[("Rain", "No")] * temperature_learning[("Hot", "No")] * humidity_learning[("High", "No")] * wind_learning[("Weak", "No")] * play_prob

print("P(Yes|x'):", predicted_1)
print("P(No|x'): ", predicted_2)
if predicted_2 > predicted_1:
  print("Since P(No|x') > P(Yes|x'), resultant is No: ", [0])
else:
  print("Since P(Yes|x') > P(No|x'), resultant is Yes: ", [1])
```

```
P(Yes|x'): 0.010582010582010581
P(No|x'):  0.013714285714285715
Since P(No|x') > P(Yes|x'), resultant is No:  [0]

P(Yes|x'): 0.010582010582010581
P(No|x'):  0.03291428571428573
Since P(No|x') > P(Yes|x'), resultant is No:  [0]
```

From the above output we can say that we can't play tennis in any case.

Code:

Using Built-in Libraries

```
import pandas as pd
```

```
dataSet = {
    "Outlook": ["Sunny", "Sunny", "Overcast", "Rain", "Rain", "Rain", "Overcast", "Sunny", "Sunny", "Rain", "Sunny", "Overcast", "Overcast", "Rain"],
    "Temperature": ["Hot", "Hot", "Hot", "Mild", "Cool", "Cool", "Cool", "Mild", "Cool", "Mild", "Mild", "Mild", "Hot", "Mild"],
    "Humidity": ["High", "High", "High", "High", "Normal", "Normal", "Normal", "High", "Normal", "Normal", "Normal", "High", "Normal", "High"],
    "Wind": ["Weak", "Strong", "Weak", "Weak", "Weak", "Strong", "Strong", "Weak", "Weak", "Weak", "Strong", "Strong", "Weak", "Strong"],
    "PlayTennis": ["No", "No", "Yes", "Yes", "Yes", "No", "Yes", "No", "Yes", "Yes", "Yes", "Yes", "Yes", "No"]
}

data = pd.DataFrame(dataSet)

data
```

	Outlook	Temperature	Humidity	Wind	PlayTennis
0	Sunny	Hot	High	Weak	No
1	Sunny	Hot	High	Strong	No
2	Overcast	Hot	High	Weak	Yes
3	Rain	Mild	High	Weak	Yes
4	Rain	Cool	Normal	Weak	Yes
5	Rain	Cool	Normal	Strong	No
6	Overcast	Cool	Normal	Strong	Yes
7	Sunny	Mild	High	Weak	No
8	Sunny	Cool	Normal	Weak	Yes
9	Rain	Mild	Normal	Weak	Yes
10	Sunny	Mild	Normal	Strong	Yes
11	Overcast	Mild	High	Strong	Yes
12	Overcast	Hot	Normal	Weak	Yes
13	Rain	Mild	High	Strong	No

```
from sklearn import preprocessing

# Label encoding
outlook, temperature, humidity, wind, play = [], [], [], [], []
# Label encoder can be used to transform non-numerical labels
# (as long as they are hashable and comparable) to numerical labels.
label_encoder = preprocessing.LabelEncoder()

# Encoding outlook
cur=[]
for index, row in data.iterrows():
  cur.append(row["Outlook"])
  if(index == 13):
    outlook = cur
outlook_encoded = label_encoder.fit_transform(outlook)

# Encoding Temperature
cur=[]
for index, row in data.iterrows():
  cur.append(row["Temperature"])
  if(index == 13):
    temperature = cur
temperature_encoded = label_encoder.fit_transform(temperature)

# Encoding Humidity
cur=[]
for index, row in data.iterrows():
  cur.append(row["Humidity"])
  if(index == 13):
    humidity = cur
humidity_encoded = label_encoder.fit_transform(humidity)

# Encoding Wind
cur=[]
for index, row in data.iterrows():
  cur.append(row["Wind"])
  if(index == 13):
    wind = cur
wind_encoded = label_encoder.fit_transform(wind)

# Encoding PlayTennis
cur=[]
for index, row in data.iterrows():
  cur.append(row["PlayTennis"])
  if(index == 13):
    play = cur
play_encoded = label_encoder.fit_transform(play)
```

```
print("Outlook:", outlook_encoded)
print("Temperature: ", temperature_encoded)
print("Humidity: ", humidity_encoded)
print("Wind: ", wind_encoded)
print("Play: ", play_encoded)
```

```
Outlook: [2 2 0 1 1 1 0 2 2 1 2 0 0 1]
Temperature:  [1 1 1 2 0 0 0 2 0 2 2 2 1 2]
Humidity:  [0 0 0 0 1 1 1 0 1 1 1 0 1 0]
Wind:  [1 0 1 1 1 0 0 1 1 1 0 0 1 0]
Play:  [0 0 1 1 1 0 1 0 1 1 1 1 1 0]
```

```
# zip() function returns a zip object, which is an iterator of tuples on which the play
# column depends
features = list(zip(outlook_encoded, temperature_encoded, humidity_encoded, wind_encoded))

features
```

```
[(2, 1, 0, 1),
 (2, 1, 0, 0),
 (0, 1, 0, 1),
 (1, 2, 0, 1),
 (1, 0, 1, 1),
 (1, 0, 1, 0),
 (0, 0, 1, 0),
 (2, 2, 0, 1),
 (2, 0, 1, 1),
 (1, 2, 1, 1),
 (2, 2, 1, 0),
 (0, 2, 0, 0),
 (0, 1, 1, 1),
 (1, 2, 0, 0)]
```

```
# Gaussian Naïve Bayes classifier assumes that the data from each label is drawn
# from a simple Gaussian distribution. The Scikit-learn provides sklearn.naive_bayes.GaussianNB
# to implement the Gaussian Naïve Bayes algorithm for classification.
from sklearn.naive_bayes import GaussianNB

model = GaussianNB()

# Fit Gaussian Naive Bayes according to X, y.
model.fit(features,play_encoded)

# From the encoded data we can decifer the following:
# Outlook:     Sunny-2, Overcast-0, Rain-1
# Temperature: Hot-1, Mild-2, Cool-0
# Humidity:    High-0, Normal-1
# Wind:        Weak-1, Strong-0
# Play:        No-0, Yes-1
predicted_1 = model.predict([[2, 0, 0, 1]]) # Sunny, Cool, High, Weak
predicted_2 = model.predict([[1, 1, 0, 1]]) # Rain, Hot, High, Weak

print(predicted_1)
print(predicted_2)
```

```
[0]
[1]
```

Using Custom Methods

```
# Total number of entries in the model
tot = 14

# This function will count the number of times
# val appears in the list arr
def count1(arr, val):
  res = 0
  for x in arr:
    res += x==val
  return res

def count2(arr1, val1, arr2, val2):
  res = 0
  for i in range(0, tot):
    res += (arr1[i] == val1 and arr2[i]==val2)
  return res
```

```
# Number of times we can play as per the model
play_times = count1(play_encoded, 1)
# Number of times we can't play
no_play_times = tot - play_times

# P(Play=Yes)
play_prob = play_times/tot
# P(Play=No)
no_play_prob = no_play_times/tot
```

```
# Learning Phase

possible_outcomes = ["Yes", "No"]

outlook_items = ["Sunny", "Overcast", "Rain"]
outlook_learning = {}

for outlook_item in outlook_items:
  for outcome in possible_outcomes:
    # This will count the number of times we have a particular outlook for a
    # specific outcome of Yes/No
    num = count2(data["Outlook"], outlook_item, data["PlayTennis"], outcome)
    # Count the number of times outcome appears in the model to
    # find the conditional probability based on outcome
    dem = count1(data["PlayTennis"], outcome)
    # Store this in our learning model for outlook
    outlook_learning[(outlook_item, outcome)] = num/dem

temperature_items = ["Hot", "Mild", "Cool"]
temperature_learning = {}
```

```
for temperature_item in temperature_items:
  for outcome in possible_outcomes:
    # This will count the number of times we have a particular temperature for a
    # specific outcome of Yes/No
    num = count2(data["Temperature"], temperature_item, data["PlayTennis"], outcome)
    # Count the number of times outcome appears in the model to
    # find the conditional probability based on outcome
    dem = count1(data["PlayTennis"], outcome)
    # Store this in our learning model for temperature
    temperature_learning[(temperature_item, outcome)] = num/dem

humidity_items = ["High", "Normal"]
humidity_learning = {}
```

```
for humidity_item in humidity_items:
  for outcome in possible_outcomes:
    # This will count the number of times we have a particular humidity for a
    # specific outcome of Yes/No
    num = count2(data["Humidity"], humidity_item, data["PlayTennis"], outcome)
    # Count the number of times outcome appears in the model to
    # find the conditional probability based on outcome
    dem = count1(data["PlayTennis"], outcome)
    # Store this in our learning model for humidity
    humidity_learning[(humidity_item, outcome)] = num/dem
```

```
wind_items = ["Strong", "Weak"]
wind_learning = {}

for wind_item in wind_items:
  for outcome in possible_outcomes:
    # This will count the number of times we have a particular wind type for a
    # specific outcome of Yes/No
    num = count2(data["Wind"], wind_item, data["PlayTennis"], outcome)
    # Count the number of times outcome appears in the model to
    # find the conditional probability based on outcome
    dem = count1(data["PlayTennis"], outcome)
    # Store this in our learning model for humidity
    wind_learning[(wind_item, outcome)] = num/dem

print("Outlook Learning:\n", outlook_learning)
print("Temperature Learning:\n", temperature_learning)
print("Humidity Learning:\n", humidity_learning)
print("Wind Learning:\n", wind_learning)
```

```
Outlook Learning:
 {('Sunny', 'Yes'): 0.2222222222222222, ('Sunny', 'No'): 0.6, ('Overcast', 'Yes'): 0.4444444444444444, ('Overcast', 'No'): 0.0, ('Rain', 'Yes')
: 0.3333333333333333, ('Rain', 'No'): 0.4}
Temperature Learning:
 {('Hot', 'Yes'): 0.2222222222222222, ('Hot', 'No'): 0.4, ('Mild', 'Yes'): 0.4444444444444444, ('Mild', 'No'): 0.4, ('Cool', 'Yes'): 0.33333333
33333333, ('Cool', 'No'): 0.2}
Humidity Learning:
 {('High', 'Yes'): 0.3333333333333333, ('High', 'No'): 0.8, ('Normal', 'Yes'): 0.6666666666666666, ('Normal', 'No'): 0.2}
Wind Learning:
 {('Strong', 'Yes'): 0.3333333333333333, ('Strong', 'No'): 0.6, ('Weak', 'Yes'): 0.6666666666666666, ('Weak', 'No'): 0.4}
```

```
# Sunny, Cool, High, Weak

# [P(Sunny|Yes)P(Cool|Yes)P(High|Yes)P(Weak|Yes)]P(Play=Yes)
predicted_1 = outlook_learning[("Sunny", "Yes")] * temperature_learning[("Cool", "Yes")] * humidity_learning[("High", "Yes")] * wind_learning[(
# [P(Sunny|No)P(Cool|No)P(High|No)P(Weak|No)]P(Play=No)
predicted_2 = outlook_learning[("Sunny", "No")] * temperature_learning[("Cool", "No")] * humidity_learning[("High", "No")] * wind_learning[("We

print("P(Yes|x'):", predicted_1)
print("P(No|x'): ", predicted_2)
if predicted_2 > predicted_1:
  print("Since P(No|x') > P(Yes|x'), resultant is No: ", [0])
else:
  print("Since P(Yes|x') > P(No|x'), resultant is Yes: ", [1])
print()

# Rain, Hot, High, Weak

# [P(Rain|Yes)P(Hot|Yes)P(High|Yes)P(Weak|Yes)]P(Play=Yes)
predicted_1 = outlook_learning[("Rain", "Yes")] * temperature_learning[("Hot", "Yes")] * humidity_learning[("High", "Yes")] * wind_learning[("W
# [P(Rain|No)P(Hot|No)P(High|No)P(Weak|No)]P(Play=No)
predicted_2 = outlook_learning[("Rain", "No")] * temperature_learning[("Hot", "No")] * humidity_learning[("High", "No")] * wind_learning[("Weak

print("P(Yes|x'):", predicted_1)
print("P(No|x'): ", predicted_2)
if predicted_2 > predicted_1:
  print("Since P(No|x') > P(Yes|x'), resultant is No: ", [0])
else:
  print("Since P(Yes|x') > P(No|x'), resultant is Yes: ", [1])
```

```
P(Yes|x'): 0.010582010582010581
P(No|x'):  0.013714285714285715
Since P(No|x') > P(Yes|x'), resultant is No:  [0]

P(Yes|x'): 0.010582010582010581
P(No|x'):  0.03291428571428573
Since P(No|x') > P(Yes|x'), resultant is No:  [0]
```

Observations:

1. First thing to observe is that when we use built in python libraries sklearn.naive_bayes to implement the Gaussian Naïve Bayes algorithm for classification then we get the answers as false and true respectively for the two parts.
2. But when we use custom ways to figure out the conditional probabilities for the two parts then we get both parts as false.
3. Another thing which I learnt through this question was the usage of zip function. I had learnt about this function but not used much. This question provided an appropriate use case of this function where in the feature values are extracted and zipped together.

2. Write a program for the decision tree for the below scenario with
   a) Gini as impurity index
   b) Information Gain as impurity index
   Write a program to classify a new sample X using decision tree with above 2 cases, when :

   - outlook = sunny, temperature = cool, humidity = high and windy = false
   - X = rain, hot, high, false

   a) Compare the your results with the in-build library of decision tree in python .
   b) Compare the results with the Bayesian Classifier.

Output:

Using Builtin Libraries

```
# From the encoded data we can decifer the following:
# Outlook:     Sunny-2, Overcast-0, Rain-1
# Temperature: Hot-1, Mild-2, Cool-0
# Humidity:    High-0, Normal-1
# Wind:        Weak-1, Strong-0
# Play:        No-0, Yes-1

# Using Gini

gini_predicted_1 = clf_gini.predict([(2, 0, 0, 1)]) # Sunny, Cool, High, Weak
gini_predicted_2 = clf_gini.predict([(1, 1, 0, 1)]) # Rain, Hot, High, Weak

print("Using gini as impurity index: ")
print(gini_predicted_1)
print(gini_predicted_2)
print()

# Usig Information gain

ig_predicted_1 = clf_ig.predict([(2, 0, 0, 1)]) # Sunny, Cool, High, Weak
ig_predicted_2 = clf_ig.predict([(1, 1, 0, 1)]) # Rain, Hot, High, Weak

print("Using information gain as impurity index: ")
print(ig_predicted_1)
print(ig_predicted_2)
```

```
Using gini as impurity index:
[0]
[1]

Using information gain as impurity index:
[0]
[1]
```

From the above output we can say that we can play tennis in the first case while we cannot in the second case.

Using custom Methods

```
predict(tree, {"Outlook": "Sunny", "Temerature": "Cool", "Humidity": "High", "Wind": "Weak"})
```

```
'No'
```

```
predict(tree, {"Outlook": "Rain", "Temerature": "Hot", "Humidity": "High", "Wind": "Weak"})
```

```
'Yes'
```

From the above output we can say that we can play tennis in the first case while we cannot in the second case.

Code:

Using built in Libraries

```
from sklearn import preprocessing

# Label encoding
outlook, temperature, humidity, wind, play = [], [], [], [], []
label_encoder = preprocessing.LabelEncoder()
cur=[]
for index, row in data.iterrows():
  cur.append(row["Outlook"])
  if(index == 13):
    outlook = cur
outlook_encoded = label_encoder.fit_transform(outlook)
cur=[]
for index, row in data.iterrows():
  cur.append(row["Temperature"])
  if(index == 13):
    temperature = cur
temperature_encoded = label_encoder.fit_transform(temperature)
cur=[]
for index, row in data.iterrows():
  cur.append(row["Humidity"])
  if(index == 13):
    humidity = cur
humidity_encoded = label_encoder.fit_transform(humidity)
cur=[]
for index, row in data.iterrows():
  cur.append(row["Wind"])
  if(index == 13):
    wind = cur
wind_encoded = label_encoder.fit_transform(wind)
cur=[]
for index, row in data.iterrows():
  cur.append(row["PlayTennis"])
  if(index == 13):
    play = cur
play_encoded = label_encoder.fit_transform(play)

print("Outlook:", outlook_encoded)
print("Temperature: ", temperature_encoded)
print("Humidity: ", humidity_encoded)
print("Wind: ", wind_encoded)
print("Play: ", play_encoded)
```

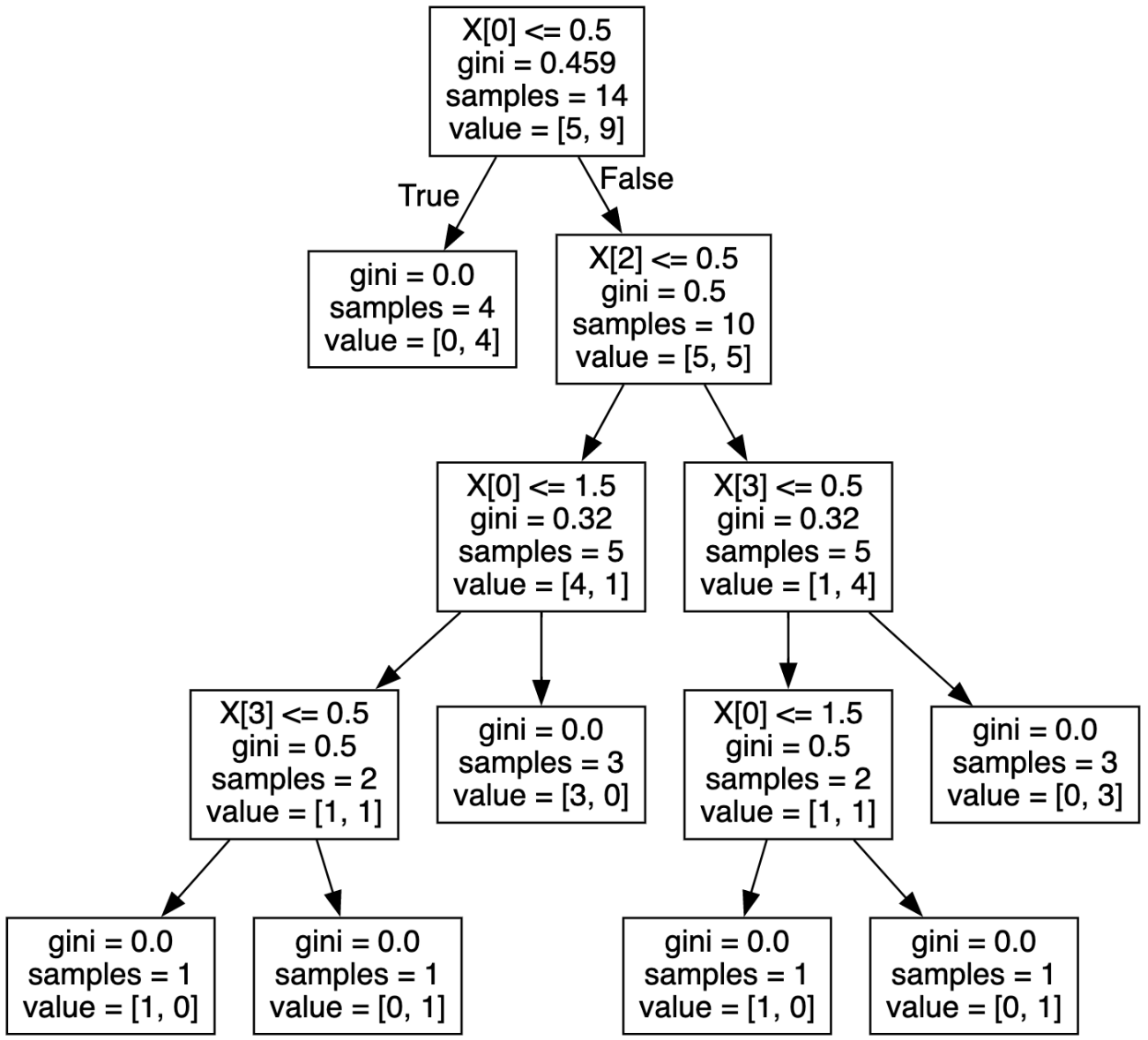
```
Outlook: [2 2 0 1 1 1 0 2 2 1 2 0 0 1]
Temperature:  [1 1 1 2 0 0 0 2 0 2 2 2 1 2]
Humidity:  [0 0 0 0 1 1 1 0 1 1 1 0 1 0]
Wind:  [1 0 1 1 1 0 0 1 1 1 0 0 1 0]
Play:  [0 0 1 1 1 0 1 0 1 1 1 1 1 0]
```

```
# zip() function returns a zip object, which is an iterator of tuples on which the play
# column depends. Thus, this will zip the deciding columns as a single unit
X = list(zip(outlook_encoded, temperature_encoded, humidity_encoded, wind_encoded))
Y = play_encoded

print("X: ", X)
print("Y: ", Y)
```

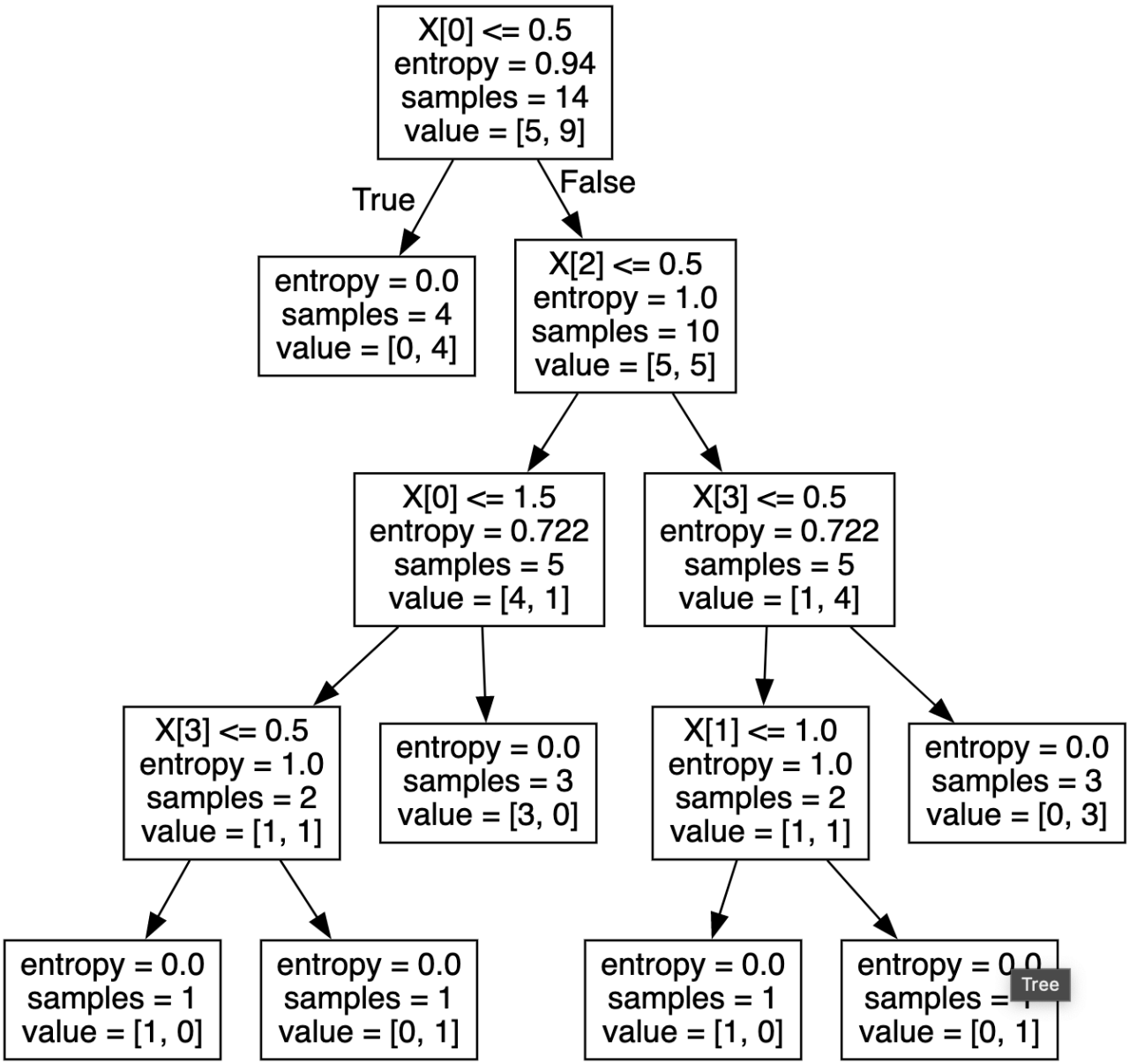
```
X:  [(2, 1, 0, 1), (2, 1, 0, 0), (0, 1, 0, 1), (1, 2, 0, 1), (1, 0, 1, 1), (1, 0, 1, 0), (0, 0, 1, 0), (2, 2, 0, 1), (2, 0, 1
, 0), (0, 1, 1, 1), (1, 2, 0, 0)]
Y:  [0 0 1 1 1 0 1 0 1 1 1 1 1 0]
```

```
# DecisionTreeClassifier is a class capable of performing multi-class classification on a dataset
# based on either cart(Gini) or ID3(Entropy/Information gain)
from sklearn import tree
clf_gini = tree.DecisionTreeClassifier(criterion = 'gini')
clf_gini = clf_gini.fit(X, Y)
clf_ig = tree.DecisionTreeClassifier(criterion = 'entropy')
clf_ig = clf_ig.fit(X, Y)
```

```
import graphviz
# Creating the graph using gini as impurity index
data_gini = tree.export_graphviz(clf_gini, out_file=None)
graph_gini = graphviz.Source(data_gini)

graph_gini
```




```
import graphviz
# Creating the graph using information gain as impurity index
data_ig = tree.export_graphviz(clf_ig, out_file=None)
graph_ig = graphviz.Source(data_ig)

graph_ig
```


X[0] <= 0.5
entropy = 0.94
samples = 14
value = [5, 9]
True
False
entropy = 0.0
samples = 4
value = [0, 4]
X[2] <= 0.5
entropy = 1.0
samples = 10
value = [5, 5]
X[0] <= 1.5
entropy = 0.722
samples = 5
value = [4, 1]
X[3] <= 0.5
entropy = 0.722
samples = 5
value = [1, 4]
X[3] <= 0.5
entropy = 1.0
samples = 2
value = [1, 1]
entropy = 0.0
samples = 3
value = [3, 0]
X[1] <= 1.0
entropy = 1.0
samples = 2
value = [1, 1]
entropy = 0.0
samples = 3
value = [0, 3]
entropy = 0.0
samples = 1
value = [1, 0]
entropy = 0.0
samples = 1
value = [0, 1]
entropy = 0.0
samples = 1
value = [1, 0]
entropy = 0.0
samples = 1
value = [0, 1]
Tree

```
# From the encoded data we can decifer the following:
# Outlook:     Sunny-2, Overcast-0, Rain-1
# Temperature: Hot-1, Mild-2, Cool-0
# Humidity:    High-0, Normal-1
# Wind:        Weak-1, Strong-0
# Play:        No-0, Yes-1

# Using Gini

gini_predicted_1 = clf_gini.predict([(2, 0, 0, 1)]) # Sunny, Cool, High, Weak
gini_predicted_2 = clf_gini.predict([(1, 1, 0, 1)]) # Rain, Hot, High, Weak

print("Using gini as impurity index: ")
print(gini_predicted_1)
print(gini_predicted_2)
print()

# Usig Information gain

ig_predicted_1 = clf_ig.predict([(2, 0, 0, 1)]) # Sunny, Cool, High, Weak
ig_predicted_2 = clf_ig.predict([(1, 1, 0, 1)]) # Rain, Hot, High, Weak

print("Using information gain as impurity index: ")
print(ig_predicted_1)
print(ig_predicted_2)
```

```
Using gini as impurity index:
[0]
[1]

Using information gain as impurity index:
[0]
[1]
```

Using Custom Methods

```
training_data = data
testing_data = data

# This function calculates the entropy of the whole dataset
def find_dataset_entropy(train_data, label, class_items):
    # Find the total number of rows
    rows = train_data.shape[0]
    # Resultant entropy to be returned
    entropy = 0

    # For each class variable (Yes/No), Calculates its respective entropy
    for item in class_items:
        # Number of times a particual vaiable appears in the respective column
        cur_item_count = train_data[train_data[label] == item].shape[0]
        # Entropy of the current item
        cur_item_entropy = - (cur_item_count/rows)*np.log2(cur_item_count/rows)
        # Add it to our entropy to be returned
        entropy += cur_item_entropy

    # Return the resultant entropy
    return entropy
```

```python
# Calculating entropy of a filtered dataset based on a specific feature
def calculate_feature_entropy(feature_data, label, class_items):
    # feature_data contains rows that have a specific value of a feature
    # class_count will basically contain the total number of raws of the
    # current dataframe
    class_count = feature_data.shape[0]
    entropy = 0

    for item in class_items:
        # Number of times a particular item appears
        cur_item_count = feature_data[feature_data[label] == item].shape[0]

        cur_item_entropy = 0
        if cur_item_count != 0: # To avoid divide by zero exception
            cur_item_entropy = - (cur_item_count/class_count) * np.log2(cur_item_count/class_count)

        # Add it to out entropy to be returned
        entropy += cur_item_entropy

    # Return the resultant entropy
    return entropy
```

```python
# This funciton would calculate information gain for a particular feature
def calculate_information_gain(feature_name, train_data, label, class_items):
    # This is a list of unique items pertaining to a specific feature
    items_list = train_data[feature_name].unique()
    rows = train_data.shape[0]
    feature_information = 0.0

    for item in items_list:
        # This will fetch all the rows from the train data where feature_name = item
        item_data = train_data[train_data[feature_name] == item]
        # Number of times a particular item appears in the train data
        item_count = item_data.shape[0]
        # Calculating the entropy of a specific item
        item_entropy = calculate_feature_entropy(item_data, label, class_items)
        item_probability = item_count/rows
        feature_information += item_probability * item_entropy

    # Entropy a feature = total entropy of the dataset - information of that feature
    return find_dataset_entropy(train_data, label, class_items) - feature_information

# Caculates the most informative feature ie the feature with highest information gain
def most_informative_feature(train_data, label, class_items):
    # This list contains the features on which our output would depend
    feature_list = train_data.columns.drop(label)
    # Initialize the variables
    max_info_gain = -1
    max_info_feature = None

    for feature in feature_list:
        # For the current feature calculate the information gain
        feature_info_gain = calculate_information_gain(feature, train_data, label, class_items)
        # If the maximum informatino gain until now is smaller than the current gain
        if max_info_gain < feature_info_gain:
            # Update the maximum information gain with the current gain
            max_info_gain = feature_info_gain
            # Update the resultant feature to be returned
            max_info_feature = feature

    # Return the feature with max information gain
    return max_info_feature
```

```
# Once we have found current root of a tree this method would help in adding
# a node to the tree
def generate_sub_tree(feature_name, train_data, label, class_items):
    # feature_name is the name of the feature that we want to add to the tree
    featere_count_dict = train_data[feature_name].value_counts(sort=False)
    tree = {}

    for feature, count in featere_count_dict.iteritems():
        # For each (feature, count) pair in our dictionary
        # feature_data is the dataset containing the value of the feature_name as the
        # current feaure
        feature_data = train_data[train_data[feature_name] == feature]

        # A flag to keep record if its assigned to a node
        # In other words to check if the current feature value is a pure class or not
        assigned = False
        for item in class_items:
            # This counts the number of times item class appears in our dataset for a
            # specific feature
            cur_class_count = feature_data[feature_data[label] == item].shape[0]

            # Now if the count of current class is same as the total count,
            # this means that it is a pure class thus that feature in out tree can be assigned
            # value as this item
            if cur_class_count == count:
                # Being pure class assign item as the value for this feature in our subtree
                tree[feature] = item
                # Remove the rows with this feature value
                train_data = train_data[train_data[feature_name] != feature]
                # Mark the flag variable
                assigned = True

        # If the node is not yet assigned so that feature in our dataset is still an impure class
        if not assigned:
            tree[feature] = "?"

    # return the subtree formed and train_data
    return tree, train_data
```

```
# Generation of tree: A recursive function to create decision tree using the above
# defined functinos
def generate_tree(root, prev_feature, train_data, label, class_items):
    # If there exists some dataset
    if train_data.shape[0] != 0:
        # Find the most informative feature
        max_info_feature = most_informative_feature(train_data, label, class_items)
        # Getting in the tree node and the updated dataset
        tree, train_data = generate_sub_tree(max_info_feature, train_data, label, class_items)
        next_root = None

        # Add the node with branches into the tree
        if prev_feature != None:
            # If its not none then add it to the intermediate node of our tree
            root[prev_feature] = dict()
            root[prev_feature][max_info_feature] = tree
            next_root = root[prev_feature][max_info_feature]
        else:
            # If the previous feature value is none that means we need to
            # add to the root of the tree
            root[max_info_feature] = tree
            next_root = root[max_info_feature]

        # Iterate on the tree node
        for node, branch in list(next_root.items()):
            # If a branch is ? then it means that it is expandable
            if branch == "?":
                # Done using the updated dataset
                feature_value_data = train_data[train_data[max_info_feature] == node]
                # Recursively calling this function with the updated dataset to create the
                # subtrees
                generate_tree(next_root, node, feature_value_data, label, class_items)
```

```
# This function is responsible for finding the unique classes of the label
# and starting the algorithm
def id3(training_data, label):
    # Create a copy of the training data(Dataset)
    train_data = training_data.copy()
    # Initializing the tree which will be updated
    tree = {}
    # Fetching the unique classes for a label which is PlayTennis in our case
    class_items = train_data[label].unique()
    # Generate the tree recursively
    generate_tree(tree, None, training_data, label, class_items)

    return tree

# Predicting from the tree generated
# We will recursively traverse the nested dictionary until any leaf node (= class) is found
# The key of the dictionary is a feature name
def predict(tree, instance):
    if not isinstance(tree, dict):
        return tree
    else:
        root_node = next(iter(tree))
        feature_value = instance[root_node]
        if feature_value in tree[root_node]:
            return predict(tree[root_node][feature_value], instance)
        else:
            return None

tree = id3(training_data, 'PlayTennis')
tree
```

```
{'Outlook': {'Sunny': {'Humidity': {'High': 'No', 'Normal': 'Yes'}},
  'Overcast': 'Yes',
  'Rain': {'Wind': {'Weak': 'Yes', 'Strong': 'No'}}}}
```

```
predict(tree, {"Outlook": "Sunny", "Temerature": "Cool", "Humidity": "High", "Wind": "Weak"})
```

```
'No'
```

```
predict(tree, {"Outlook": "Rain", "Temerature": "Hot", "Humidity": "High", "Wind": "Weak"})
```

```
'Yes'
```

Observations:

1. Using the decision tree, we are getting the result as false and true respectively in the two cases asked in the question using both built in libraries and custom methods.
2. This is same as that of Bayesian classifiers result using built in libraries where in we were getting false and true for the 2 cases respectively
   Some of the terms understood are as follows:
   - Entropy is the measure of randomness or impurity in the sample

     $$\text{Entropy}(S) = -p_+ \log_2 p_+ - p_- \log_2 p_-$$

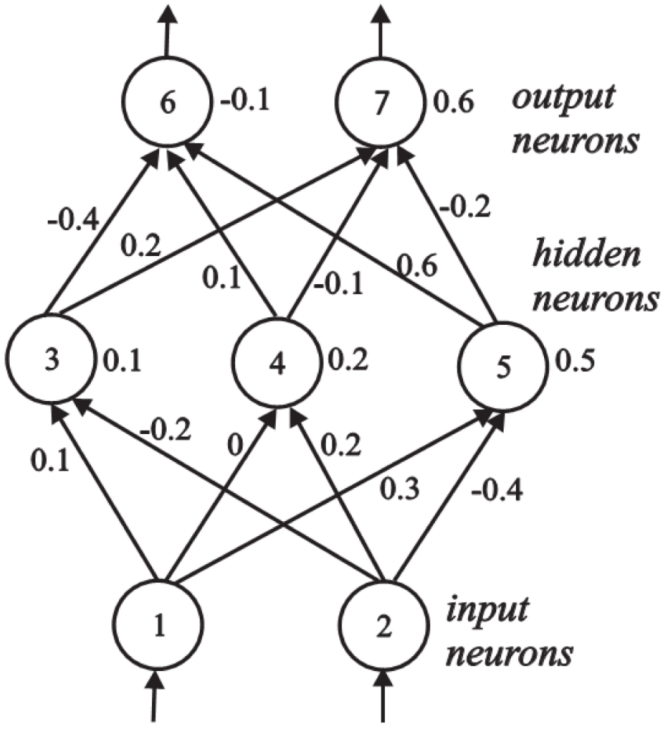
   - Information gain is the expected reduction in entropy due to partitioning of the sample S on a feature A

$$Gain(S,A) = Entropy(S) - \sum_{v \in values(A)} |S_v|/|S| \; Entropy(S_v)$$

3. Consider the same example discussed in the class (2 classes, 2 dim. input data) , The training set is :
   ex.1: 0.6 0.1 — class 1 (banana)
   ex.2: 0.2 0.3 — class 2 (orange)

   Mention the following in lab file :
   a) Network architecture
   b) How many inputs?
   c) How many hidden neurons?
   d) How many output neurons?
   e) What encoding of the outputs?
   f) Initial weights and learning rate

Write a code for Back Propagation Neural Network for the following Network architecture .
Let's learning rate (η) = 0.1 and the weights are set as in the figure below. Show what will be the final weight after 1st,2nd and 50 iteration.



a) Network architecture?
   Neural

b) How many inputs?
   The number of inputs is the same as the number of samples/variables. In this case the number of inputs are 2 which are neurons 1 and 2

c) How many hidden neurons?
There are three hidden neuron which are 3, 4, 5

d) How many output neurons?
There are 2 output neuron - 6, 7

e) What encoding of the outputs?
Encoding of the output is binary which is chosen over local encoding.

f) Initial weights and learning rate?
Learning Rate: 0.1
o1(weight of the neuron 1): 0.6
o2(weight of the neuron 2): 0.1

Output:

```
Initial weights

======= Hidden layer =======

Neuron  1
weights   [0.1, -0.2, 0.1]


Neuron  2
weights   [0, 0.2, 0.2]


Neuron  3
weights   [0.3, -0.4, 0.5]


======= Output layer =======

Neuron  1
weights   [-0.4, 0.1, 0.6, -0.1]


Neuron  2
weights   [0.2, -0.1, -0.2, 0.6]


Epoch=1, Learning Rate=0.100, Error=1.050
Epoch=2, Learning Rate=0.100, Error=1.049
Epoch=3, Learning Rate=0.100, Error=1.047
Epoch=4, Learning Rate=0.100, Error=1.046
Epoch=5, Learning Rate=0.100, Error=1.045
Epoch=6, Learning Rate=0.100, Error=1.044
Epoch=7, Learning Rate=0.100, Error=1.042
Epoch=8, Learning Rate=0.100, Error=1.041
Epoch=9, Learning Rate=0.100, Error=1.040
Epoch=10, Learning Rate=0.100, Error=1.039
```

```
Epoch=11, Learning Rate=0.100, Error=1.038
Epoch=12, Learning Rate=0.100, Error=1.037
Epoch=13, Learning Rate=0.100, Error=1.036
Epoch=14, Learning Rate=0.100, Error=1.035
Epoch=15, Learning Rate=0.100, Error=1.035
Epoch=16, Learning Rate=0.100, Error=1.034
Epoch=17, Learning Rate=0.100, Error=1.033
Epoch=18, Learning Rate=0.100, Error=1.032
Epoch=19, Learning Rate=0.100, Error=1.031
Epoch=20, Learning Rate=0.100, Error=1.031
Epoch=21, Learning Rate=0.100, Error=1.030
Epoch=22, Learning Rate=0.100, Error=1.030
Epoch=23, Learning Rate=0.100, Error=1.029
Epoch=24, Learning Rate=0.100, Error=1.028
Epoch=25, Learning Rate=0.100, Error=1.028
Epoch=26, Learning Rate=0.100, Error=1.027
Epoch=27, Learning Rate=0.100, Error=1.027
Epoch=28, Learning Rate=0.100, Error=1.026
Epoch=29, Learning Rate=0.100, Error=1.026
Epoch=30, Learning Rate=0.100, Error=1.025
Epoch=31, Learning Rate=0.100, Error=1.025
Epoch=32, Learning Rate=0.100, Error=1.024
Epoch=33, Learning Rate=0.100, Error=1.024
Epoch=34, Learning Rate=0.100, Error=1.024
Epoch=35, Learning Rate=0.100, Error=1.023
Epoch=36, Learning Rate=0.100, Error=1.023
Epoch=37, Learning Rate=0.100, Error=1.023
Epoch=38, Learning Rate=0.100, Error=1.022
Epoch=39, Learning Rate=0.100, Error=1.022
Epoch=40, Learning Rate=0.100, Error=1.022
Epoch=41, Learning Rate=0.100, Error=1.021
Epoch=42, Learning Rate=0.100, Error=1.021
Epoch=43, Learning Rate=0.100, Error=1.021
Epoch=44, Learning Rate=0.100, Error=1.021
Epoch=45, Learning Rate=0.100, Error=1.020
Epoch=46, Learning Rate=0.100, Error=1.020
Epoch=47, Learning Rate=0.100, Error=1.020
Epoch=48, Learning Rate=0.100, Error=1.020
Epoch=49, Learning Rate=0.100, Error=1.020
Epoch=50, Learning Rate=0.100, Error=1.019
```

Printing the first 50 epochs out of 5000

```
Final weights

======= Hidden layer =======

Neuron  1
weights   [5.564400873246104, -0.7625977573215473, -1.7753258577384965]
output   0.29103411074088203


Neuron  2
weights   [-3.8398637941009905, 0.4634998604537811, 1.078332868179256]
output   0.6103536078313122


Neuron  3
weights   [-5.290374922026391, 0.01401025379140653, 1.8800341793046895]
output   0.6953776652483354


======= Output layer =======

Neuron  1
weights   [-4.697721739008186, 2.6992366999341444, 4.101000904514553, -0.4640549631748329]
output   0.9350641841410314


Neuron  2
weights   [4.4939714352122895, -2.9757442179979146, -4.035897074124544, 0.6640809296490285]
output   0.06599150067169088
```

Code:

```
from math import exp

# Print the neurons
def print_arr(arr):
  i=1
  for item in arr:
    print("\nNeuron ", i)
    for key in item.keys():
      if key != 'delta':
        print(key, " ", item[key])
    print()
    i+=1

# Generating the dictionary to store the neuron
def gen(arr):
        res = []
        for item in arr:
                cur = {'weights': item}
                res.append(cur)
        return res

# This function does the initialization of the network
def create_network():
  network = []
  # Each dictionary would store the weight, output and delta of a specific neuron in a layer
  # Number of entries of weights list = number of incoming weights to that neuron + 1 (bias)
  # We have 2 links per hidden neuron thus 3 entries
  hidden_wts = [[0.1, -0.2, 0.1], [0, 0.2, 0.2], [0.3, -0.4, 0.5]]
  # We have 3 links per output neuron thus 4 entries
  ouput_wts = [[-0.4, 0.1, 0.6, -0.1], [0.2, -0.1, -0.2, 0.6]]
  hidden_layer, output_layer = gen(hidden_wts), gen(ouput_wts)
  # Appending the layers in the network
  network.append(hidden_layer)
  network.append(output_layer)
  # Inital printing of information
  print("Initial weights \n")
  print("======= Hidden layer =======")
  print_arr(hidden_layer)
  print("\n======= Output layer =======")
  print_arr(output_layer)
  print()
  # Return the network formed
  return network
```

```
# This would return the weighted sum of the links to a specific neuron
# added with the bias corresponding to that neuron which acts as the linear
# component of the inputs
def linear_component(weights, inputs):
  # Bias to a particualr neuron is the last entry of the list
  bias = weights[-1]
  sum, weighted_sum = 0, 0
  for i in range(len(weights)-1):
    # Add the respective weighted sum
    weighted_sum += weights[i] * inputs[i]
  sum = weighted_sum + bias
  # Returning the resultant sum
  return sum

# Signmoid function implementation which is the non linear
# component of the input
def non_linear_component(activation):
        return 1.0 / (1.0 + exp(-activation))
```

```
def forward_propagation(network, row):
        inputs = row
  # Iterate on the layers of the network
        for layer in network:
                new_inputs = []
    # For each layer, for each neuron
                for neuron in layer:
      # Figure out the resultant input to this neuron which is the
      # weighted sum of the links along with the bias
                        activation = linear_component(neuron['weights'], inputs)
      # Now the output of this neuron is the result of the activation
      # function which is then stored as new inputs for our next layer
                        neuron['output'] = non_linear_component(activation)
                        new_inputs.append(neuron['output'])
                inputs = new_inputs
  # The last list produced by the iteration is the result of the output
  # layer which is being returned from here
        return inputs

# In theory this is the partial derivative of the output for a specifc
# neuron with respect to the net
def transfer_derivative(output):
        return output * (1.0 - output)
```

```
# Once the output is generated by forward propagation and error is found
# we have to minimize that error
# This is achieved by backward propagation where the weights of the links are
# adjusted in such a manner that the resultant error is minimized
def backward_propagate_error(network, expected):
  # We start this process from the outer most layer
       for i in reversed(range(len(network))):
    # Extract the current layer of neurons
                current_layer = network[i]
                errors = []
    # If I am not at the last neuron
                if i != len(network)-1:
                        for j in range(len(current_layer)):
                                error = 0.0
        # Since the network is traversed in the backwards direction
        # this ensures that the output layer has delta values calculated first
        # which can be used in the hidden layer for calculating errors
                                for neuron in network[i + 1]:
          # j is the index of the neuron's weight in the next layer
          # which is being multiplied with the delta value of that specific neuron
          # from the next layer
                                        error += (neuron['weights'][j] * neuron['delta'])
                                errors.append(error)
                else:
                        for j in range(len(current_layer)):
                                neuron = current_layer[j]
                                errors.append(neuron['output'] - expected[j])
    # Once the errors are generated for the current layer neurons,
    # those are then multiplied by the transfer derivative function
    # which is a part of the formula obtained by the chain rule
                for j in range(len(current_layer)):
                        neuron = current_layer[j]
                        neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])
```

```
# This function is responsible for updating the weights with the error derivatives
# for a specific layer
def updation_of_weights(network, row, learning_rate):
        for i in range(len(network)):
    # Each row contains the output at the end of the row and remaining is input data
                inputs = row[:-1]
                if i != 0:
                        inputs = [neuron['output'] for neuron in network[i - 1]]
                for neuron in network[i]:
                        for j in range(len(inputs)):
        # As per the formula obtained by the chain rule,
        # New weight = old weight - learning rate * partial derivative of the total error
        # with respect to the current weight being considered
                                neuron['weights'][j] -= learning_rate * neuron['delta'] * inputs[j]
                        neuron['weights'][-1] -= learning_rate * neuron['delta']
```

```
# Now once the algoithm is in place we have to train the neural network with the dataset
# provided in the question
def neural_network_training(network, training_dataset, learning_rate, tot_epochs, tot_outputs):
        for epoch in range(tot_epochs):
    # For each of the iterations update the weights, and find the error
                error_sum = 0
                for row in training_dataset:
      # Output produced with the current input set
                        outputs = forward_propagation(network, row)
                        expected = [0 for i in range(tot_outputs)]
      # We have the expected output for this particular input which is stored at the end
      # of this row
                        expected[row[-1]] = 1
      # Error = (expected - obtained) ^ 2
      # Here we are obtaining the error for all the output neurons and adding them together
      # to find the total error corresponding to this iteration
                        error_sum += sum([(expected[i]-outputs[i])**2 for i in range(len(expected))])
      # Once total error is generated we update the weights to fit the data by the method of
      # back propagation
                        backward_propagate_error(network, expected)
      # Updation of weights
                        updation_of_weights(network, row, learning_rate)
                print('Epoch=%d, Learning Rate=%.3f, Error=%.3f' % (epoch+1, learning_rate, error_sum))
```

```
# Training dataset as per the question
dataset = [[0.6, 0.3, 1], [0.2, 0.3, 0]]
# Generating the number of distinct ouputs possible
n_outputs = len(set([row[-1] for row in dataset]))
# Network creation
network = create_network()
# Training the neural network
neural_network_training(network, dataset, 0.1, 5000, n_outputs)

print("\nFinal weights \n")
print("======= Hidden layer =======")
print_arr(network[0])
print("\n======= Output layer =======")
print_arr(network[1])
```

Observations:

1. This question helped in understanding the process of back propagation which I feel is a bit complex to understand in the beginning without an example due to extensive mathematical calculations.
2. In this I have trained the model for 5000 epochs thus resulting in error of about 0.018 and accuracy of about 98.2%.
3. We wanted the output to be 1 for the 1st neuron in the output layer and 0 for the 2nd neuron i.e. Class 1 as we are using binary encoding. After doing 5000 iterations our model a produces the

following result for the inputs 0.6 and 0.1 which is not present in our training data.