‘Bharatiya Vidya Bhavan’s
Sardar Patel Institute of Technology
Bhavan’s Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India
(Autonomous College Affiliated to University of Mumbai)

BE-ETRX **UID:**2019110039 **Sub-Minor ML**

NAME: Devansh Palliyath

Exp 3

Aim: Implementing KNN algorithm on the given dataset.

Code:

```
import numpy as np
from numpy import linalg as lin
from cs771 import genSyntheticData as gsd
from cs771 import plotData as pd
from matplotlib import pyplot as plt
import time as t
class Node:
    # A node stores its own depth (root = depth 0), its decision stump,
its parent and child information
    # Leaf nodes also store a constant label that is assigned to every
data point that reaches that leaf
    def __init__( self, depth = 0, stump = (0,0), parent = None ):
        self.depth = depth
        self.stump = stump
        self.parent = parent
        self.left = None
        self.right = None
        self.isLeaf = True
        self.label = 0

    def predict( self, data ):
        # If I am a leaf I can predict rightaway
        # May change this constant leaf action to something more
interesting and powerful
        if self.isLeaf:
```

```
            return self.label
        # Else I have to ask one of my children to do the job
        else:
            if data[self.stump[0]] > self.stump[1]:
                return self.right.predict( data )
            else:
                return self.left.predict( data )

    # Get the Gini coefficient of a node with nPos positive points and 
nNeg negative points
    def getGini( self, nPos, nNeg ):
        nTot = nPos + nNeg
        # Find the proportion of the positives and negatives in that node
        pPos = nPos/nTot
        pNeg = nNeg/nTot
        # The gini index is always a real number between 0 and 0.5
        # A perfectly pure node has gini index = 0
        # The smaller the gini index the purer the node
        gini = 1 - (pPos**2 + pNeg**2)
        return gini
        def getStump( self, X, y ):
        # How many data points do I have at this node?
        n = y.size
        bestObjective = float('inf')

        # For each of the features in the data
        for i in range( X.shape[1] ):
            # Do not use the same feature as used by the parent node
            if self.parent is not None and i == self.parent.stump[0]:
                continue
            # Find out all values at which we can threshold that feature
            candidateThresholds = np.sort( X[:, i] )
            idx = np.argsort( X[:, i] )
            # The cumulative sum trick used here will work only if labels 
are binary
            ySorted = y[idx]
            yCum = np.cumsum( ySorted )
            yCumRev = np.cumsum( ySorted[::-1] )[::-1]
            # For each possible threshold (except the ones at the extreme)
            for j in range( 1, candidateThresholds.size-1 ):
```

```
                # Give 0.5 weight to balance and 1 weight to purity of the
two nodes
                candidateObjective = 0.0 * 0.0 \
                                    + 1.0 * self.getGini( (yCum[j] +
j+1)/2, (j+1 - yCum[j])/2 ) \
                                    + 1.0 * self.getGini( (yCumRev[j+1] +
n-j-1)/2, (n-j-1 - yCumRev[j+1])/2 )
                if candidateObjective < bestObjective:
                    bestObjective = candidateObjective
                    bestFeat = i
                    bestThresh = candidateThresholds[j]

        # Can try LwP decision stump as well
        bestThresh = (np.mean(X[y > 0, bestFeat]) + np.mean(X[y < 0,
bestFeat]))/2
        return (bestFeat, bestThresh)
         def train( self, X, y, maxLeafSize, maxDepth ):
        # If too few data points are present, or else if this node is too
deep in the tree, make this a leaf
        if y.size < maxLeafSize or self.depth >= maxDepth:
            self.isLeaf = True
            self.label = np.mean( y )
        else:
            # This node will be split and hence it is not a leaf
            self.isLeaf = False
            # Get the best possible decision stump
            self.stump = self.getStump( X, y )
            self.left = Node( depth = self.depth + 1, parent = self )
            self.right = Node( depth = self.depth + 1, parent = self )
            # Find which points go to my left child and which go to my
right child
            discriminant = X[:, self.stump[0]] - self.stump[1]
            # Train my two children recursively
            self.left.train( X[discriminant <= 0, :], y[discriminant <=
0], maxLeafSize, maxDepth )
            self.right.train( X[discriminant > 0, :], y[discriminant > 0],
maxLeafSize, maxDepth )
d = 2
n = 50
r = 2
```

```
tmp1 = gsd.genSphericalData( d, n, [-5, -7], r )
tmp2 = gsd.genSphericalData( d, n, [5, 0], r )
tmp3 = gsd.genSphericalData( d, n, [-5, 7], r )
XPos = np.vstack( (tmp1, tmp2, tmp3) )
yPos = np.ones( (3*n,) )

tmp1 = gsd.genSphericalData( d, n, [5, -7], r )
tmp2 = gsd.genSphericalData( d, n, [-5, 0], r )
tmp3 = gsd.genSphericalData( d, n, [5, 7], r )
XNeg = np.vstack( (tmp1, tmp2, tmp3) )
yNeg = -np.ones( (3*n,) )

X = np.vstack( (XPos, XNeg) )
y = np.concatenate( (yPos, yNeg) )

DT = Tree( maxLeafSize = 5, maxDepth = 4 )
DT.train( X, y )

def drawTreeSplits( node, fig, xlim, ylim ):
    if not node.isLeaf:
        plt.figure( fig.number )
        # Is this a vertical split or a horizontal one?
        if node.stump[0] == 0:
            plt.plot( [node.stump[1], node.stump[1]], ylim, color = 'c',
linestyle = '--' )
            drawTreeSplits( node.left, fig, [xlim[0], node.stump[1]], ylim
)
            drawTreeSplits( node.right, fig, [node.stump[1], xlim[1]],
ylim )
        elif node.stump[0] == 1:
            plt.plot( xlim, [node.stump[1], node.stump[1]], color = 'c',
linestyle = '--' )
            drawTreeSplits( node.left, fig, xlim, [ylim[0], node.stump[1]]
)
            drawTreeSplits( node.right, fig, xlim, [node.stump[1],
ylim[1]] )

def kNNClass( xt, yt ):
    diff = X - np.array( [xt, yt] )
```

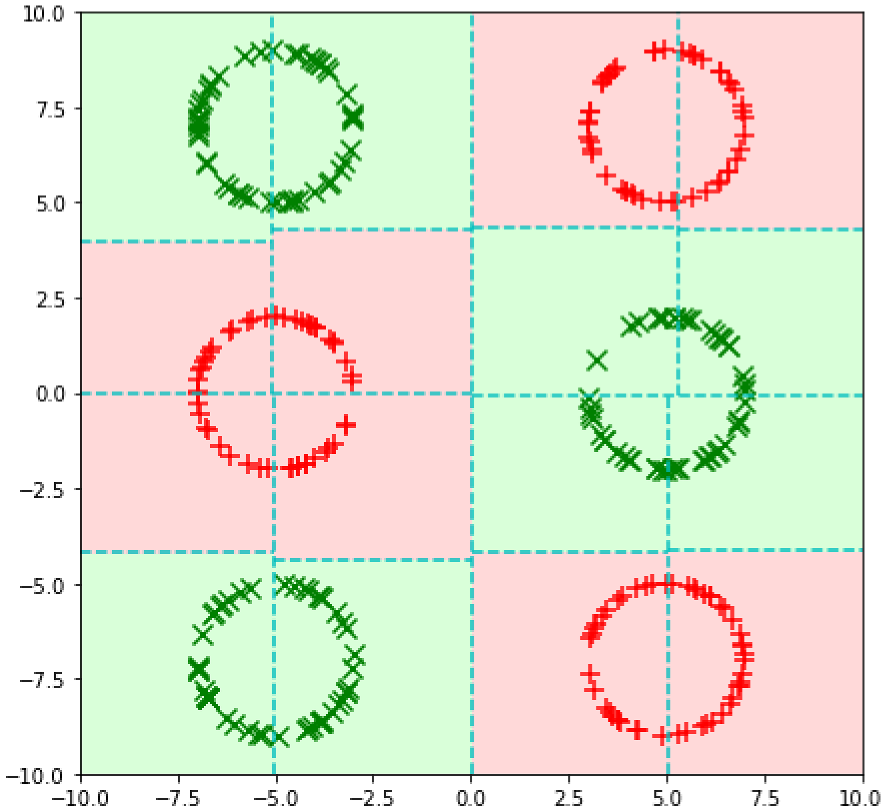
```
    dist = lin.norm( diff, axis = 1 )
    idx = np.argsort( dist )
    yhat = 0
    wsum = 0
    for i in range( k ):
        yhat = yhat + y[idx[i]]
    return yhat/k

fig = pd.getFigure()
tic = t.process_time()
pd.shade2D( DT.predict, fig, mode = 'point', xlim = 10, ylim = 10, nBins =
500 )
toc = t.process_time()
print( "It took " + str(toc - tic) + " seconds to complete the shading
with a DT")
drawTreeSplits(DT.root, fig, xlim = [-10, 10],  ylim = [-10, 10])
pd.plot2D( XNeg, fig, color = 'r', marker = '+' )
pd.plot2D( XPos, fig, color = 'g', marker = 'x' )

k = 1
fig2 = pd.getFigure()
tic = t.process_time()
pd.shade2D( kNNClass, fig2, mode = 'point', xlim = 10, ylim = 10, nBins =
500 )
toc = t.process_time()
print( "It took " + str(toc - tic) + " seconds to complete the shading
with kNN")
pd.plot2D( XNeg, fig2, color = 'r', marker = '+' )
pd.plot2D( XPos, fig2, color = 'g', marker = 'x' )
print( "DTs get faster compared to kNN as number of training points
increases - set n = 500 and see" )
```

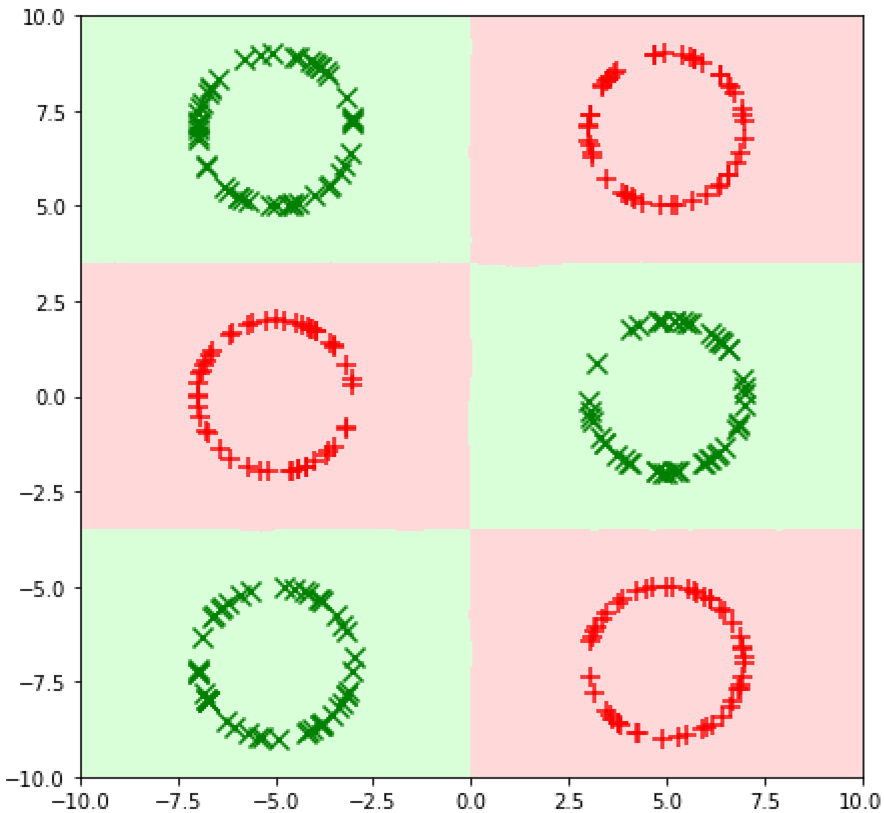
Output:

It took 0.8856712210000026 seconds to complete the shading with a DT

It took 8.132892257000002 seconds to complete the shading with kNN

DTs get faster compared to kNN as number of training points increases - set n = 500 and see






```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from google.colab import drive
drive.mount('/content/drive')
df = pd.read_csv('/content/drive/MyDrive/data/breast-cancer.csv')
columns=df.columns
columns=columns[1:-1]
print(columns)
```

```
Index(['diagnosis', 'radius_mean', 'texture_mean', 'perimeter_mean',
       'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean',
       'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean',
       'radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se',
       'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se',
       'fractal_dimension_se', 'radius_worst', 'texture_worst',
       'perimeter_worst', 'area_worst', 'smoothness_worst',
       'compactness_worst', 'concavity_worst', 'concave points_worst',
       'symmetry_worst'],
      dtype='object')
```

```
df['diagnosis'].replace(['B', 'M'], [0, 1], inplace=True)
```

```python
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
```

```python
datavar = pd.DataFrame(df, columns=['id','radius_mean', 'texture_mean', 'perimeter_mean',
       'area_mean', 'smoothness_mean', 'compactness_mean', 'concavity_mean',
       'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean',
       'radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se',
       'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se',
       'fractal_dimension_se', 'radius_worst', 'texture_worst',
       'perimeter_worst', 'area_worst', 'smoothness_worst',
       'compactness_worst', 'concavity_worst', 'concave points_worst',
       'symmetry_worst', 'fractal_dimension_worst'])

target=df['diagnosis']
```

```python
X = datavar
y = target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1) # 80% training and 20% test
clf = DecisionTreeClassifier()
clf = clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.9385964912280702
```

```python
from sklearn.tree import DecisionTreeRegressor

# create a regressor object
regressor = DecisionTreeRegressor(random_state = 0)

# fit the regressor with X and Y data
regressor.fit(X, y)
```

```
DecisionTreeRegressor(random_state=0)
```

```python
y_pred = regressor.predict(X_test)
```

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

```
Accuracy: 1.0
```

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
df_classifier = DecisionTreeClassifier(random_state=999)

params_DT = {'criterion': ['gini', 'entropy'],
             'max_depth': [1, 2, 3, 4, 5, 6, 7, 8],
             'min_samples_split': [2, 3]}

gs_DT = GridSearchCV(estimator=df_classifier,
                     param_grid=params_DT,
                     cv=15,
                     verbose=1,
                     scoring='accuracy')

gs_DT.fit(datavar, target);
```

```
Fitting 15 folds for each of 32 candidates, totalling 480 fits
```

```
gs_DT.best_score_
```

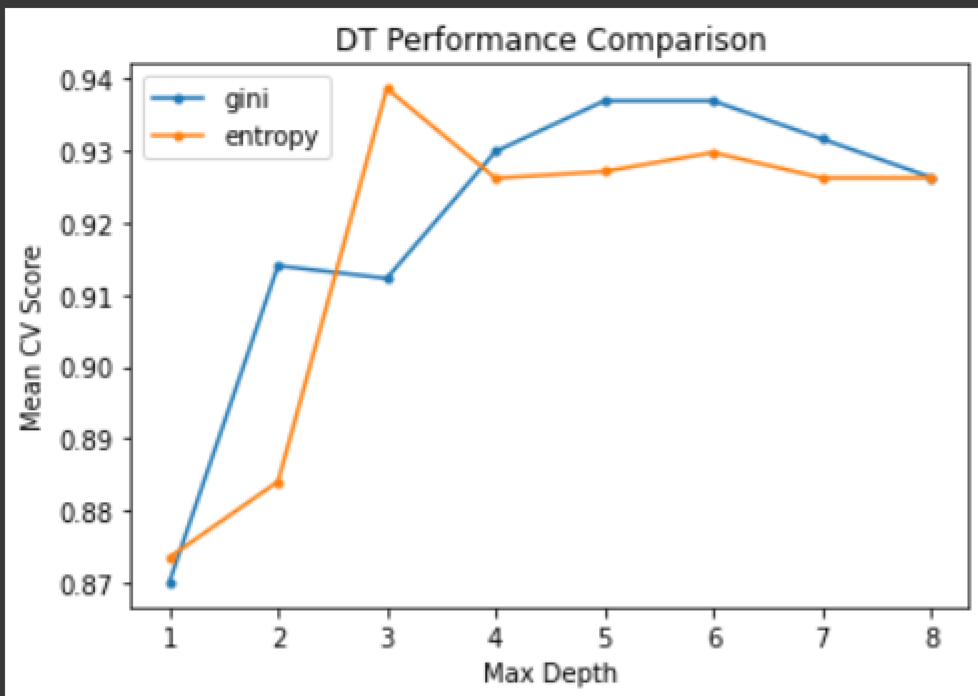
```
0.9385490753911807
```

```
results_DT = pd.DataFrame(gs_DT.cv_results_['params'])
results_DT['test_score'] = gs_DT.cv_results_['mean_test_score']
results_DT.columns
```

```
Index(['criterion', 'max_depth', 'min_samples_split', 'test_score'], dtype='object')
```

```
for i in ['gini', 'entropy']:
    temp = results_DT[results_DT['criterion'] == i]
    temp_average = temp.groupby('max_depth').agg({'test_score': 'mean'})
    plt.plot(temp_average, marker = '.', label = i)


plt.legend()
plt.xlabel('Max Depth')
plt.ylabel("Mean CV Score")
plt.title("DT Performance Comparison")
plt.show()
```

```
plt.ylabel("Mean CV Score")
plt.title("DT Performance Comparison")
plt.show()
```




```
clf = DecisionTreeClassifier(criterion='entropy', splitter='best',
max_depth=6, min_samples_split=3, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_features=None, random_state=None,
max_leaf_nodes=None, min_impurity_decrease=0.0, class_weight=None,
ccp_alpha=0.0)
clf = clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

Conclusion:

- A decision tree is a type of supervised machine learning used to categorize or make predictions based on how a previous set of questions were answered. The model is a form of supervised learning, meaning that the model is trained and tested on a set of data that contains the desired categorization.
- The Gini Impurity favors bigger partitions (distributions) and is simple to implement, whereas information gains favor smaller partitions (distributions) with a variety of diverse values, necessitating a data and splitting criterion experiment.
- Gini index favors larger partitions (distributions) and is very easy to implement whereas information gain supports smaller partitions (distributions) with various distinct values, i.e there is a need to perform an experiment with data and splitting criterion.
- We successfully plotted the gini and the entropy on the mean cv score vs max depth of the decision tree.
- We used KNN algorithm to tackle the null values.