Report Name: Decision tree and K-Nearest Neighbour algorithm

Name:Kaspia Tabbassom ID:UG02-47-18-046
Course Code:CSE-0408
Department of Computer Science and Engineering
State University of Bangladesh (SUB)
Dhaka, Bangladesh
email address :kaspiaishita@gmail.com

***Abstract*—Main theme of your assignment or academic projects.**
***Index Terms*—The word mostly used in your report.**

I. Introduction for Decision tree algorithm

A decision tree is a support tool with a tree-like structure that models probable outcomes, cost of resources, utilities, and possible consequences. Decision trees provide a way to present algorithms with conditional control statements. They include branches that represent decision-making steps that can lead to a favorable result.

II. Types of Decisions

There are two main types of decision trees that are based on the target variable, i.e., categorical variable decision trees and continuous variable decision trees.

A. Categorical variable decision tree

A categorical variable decision tree includes categorical target variables that are divided into categories. For example, the categories can be yes or no. The categories mean that every stage of the decision process falls into one category, and there are no in-betweens.

B. Continuous variable decision tree

A continuous variable decision tree is a decision tree with a continuous target variable. For example, the income of an individual whose income is unknown can be predicted based on available information such as their occupation, age, and other continuous variables.

III. 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.

IV. 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. The complete process can be better understood using the below algorithm:

o Step-1: Begin the tree with the root node, says S, which contains the complete dataset.
o Step-2: Find the best attribute in the dataset using Attribute Selection Measure (ASM).
o Step-3: Divide the S into subsets that contains possible values for the best attributes.
o Step-4: Generate the decision tree node, which contains the best attribute.
o Step-5: Recursively make new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf node.

V. Advantages of the Decision Tree

o It is simple to understand as it follows the same process which a human follow while making any decision in real-life.
o It can be very useful for solving decision-related problems.
o It helps to think about all the possible outcomes for a problem.
o There is less requirement of data cleaning compared to other algorithms.

VI. Disadvantages of the Decision Tree

o The decision tree contains lots of layers, which makes it complex.
o It may have an overfitting issue, which can be resolved

using the Random Forest algorithm.
o For more class labels, the computational complexity of the decision tree may increase.

VII. Code for decision tree

```
import numpy as np
from itertools import groupby
import math
import collections
from copy import deepcopy
import pickle

class TreeNode:
def __init__(self,split,col_index):self.col_id=col_indexself.split_value=splitself.parent=Noneself.left=Noneself.right=None
class Tree():

def __init__(self):self.treemodel=Nonedeftrain(self,trainData):Attributes/LastColumnisclassself.create_tree(trainData)defcreateTree(self,trainData):createthetreeself.tre
returns the best split on the data instance along
with the splitted dataset and column index
def getBestSplit(data):
set the max information gain
maxInfoGain = -float('inf')

convert to array
dataArray = np.asarray(data)

to extract rows and columns
dimension = np.shape(dataArray)

iterate through the matrix
for col in range(dimension[1]-1):
dataArray = sorted(dataArray, key=lambda x: x[col])
for row in range(dimension[0]-1):
val1=dataArray[row][col]
val2=dataArray[row+1][col]
expectedSplit = (float(val1)+float(val2))/2.0 infoGain,l,r=
calcInfoGain(data,col,expectedSplit)
if(infoGain¿maxInfoGain):
maxInfoGain=infoGain
best= (col,expectedSplit,l,r)
return best
```

This method is used to calculate the gain and returns the left and right data as per the split

```
def calcInfoGain(data,col,split):
totalLen = len(data)
infoGain = entropy(data)
```

$left_data, right_data = getDataSplit(data, split, col)$

$infoGain = infoGain - (len(left_data)/totalLen * entropy(left_data))$
$infoGain = infoGain - (len(right_data)/totalLen * entropy(right_data))$

$return infoGain, left_data, right_data$

$def getDataSplit(data, split, col) :$
$l_data = []$
$r_data = []$

$for val in data :$
$if(val[col] < split) :$
$l_data.append(val)$
$else :$
$r_data.append(val)$

$return l_data, r_data$

$calculates the entropy of the dataset provided$
$def entropy(data) :$
$totalLen = len(data)$
$entropy = 0$
$group_by_class = groupby(data, lambda x : x[5])$
$for key, group in group_by_class :$
$grp_len = len(list(group))$
$entropy += -(grp_len/totalLen) * math.log((grp_len/totalLen), 2)$
$return entropy$

$this method builds the decision tree recursively until the leaf nodes are re$
$def build_tree(data, parent_data) :$
$code to find out if the class variable is all one$
$value$
$count = 0;$
$group_by_class = groupby(data, lambda x : x[5])$

finds out if all the instances have the same class or not

```
for key,group in group_by_class :
count = count + 1;
```

if same class for all instances then return the leaf node class value

```
if(count==1):
return data[0][5];

elif(len(data)==0):
```

this counts all the column class variable row values and finds most common in it

```
return collections.Counter(np.asarray(data[:,5]))
.most_common(1)[0][0]

else:
bestsplit= getBestSplit(data)
node = TreeNode(bestsplit[1],bestsplit[0])
node.left= build_tree(bestsplit[2], data)
node.right = build_tree(bestsplit[3], data)
return node
```

this method is used to classify the test set with the model

created
def classify(tree, row):
if type(tree)==str:
return tree
if row[tree.col$_id$] <= $tree.split_value$:
$returnclassify(tree.left, row)$
$else$:
$returnclassify(tree.right, row)$

this method saves the decision tree model using pickle package
def saveTree(tree):
decisionTree= deepcopy(tree)
pickle.dump(decisionTree,open('model.pkl','wb'))

this method creates a confusion matrix and finds accuracy for test dataset
def build$_confusion_matrix(tree, data)$:
$confusion_mat = [[0 for row in range(4)] for col in range(4)]$

total$_len = len(data)$
$num_correct_instances = 0;$
$num_incorrect_instances = 0;$

map required to build the confusion matrix
map='B':0,'G':1,'M':2, 'N':3

for row in data:
actual$_class = row[5]$
$predicted_class = classify(tree, row)$
$if(actual_class == predicted_class)$:
$num_correct_instances = num_correct_instances + 1$
$confusion_mat[map.get(actual_class)][map.get(actual_class)] = confusion_mat[map.get(actual_class)][map.get(actual_class)] + 1$
$else$:
$num_incorrect_instances = num_incorrect_instances + 1$
$confusion_mat[map.get(actual_class)][map.get(predicted_class)] = confusion_mat[map.get(actual_class)][map.get(predicted_class)] + 1$

$print("Accuracy of the model : ", (num_correct_instances/total_len) * 100, "print("Correct instances", num_correct_instances)$
$print("Incorrect instances", num_incorrect_instances)$

print$_map = 0 :' B', 1 :' G', 2 :' M', 3 :' N'$
$print("Confusion Matrix : ")$

print(" B G M N")

ind=0;
printing matrix
for row in confusion$_mat$:
$print(print_map.get(ind), "", row)$
$ind += 1$

VIII. Conclusion

Decision trees assist analysts in evaluating upcoming choices. The tree creates a visual representation of all possible outcomes, rewards and follow-up decisions in one document. Each subsequent decision resulting from the original choice is also depicted on the tree, so you can see the overall effect of any one decision. As you go through the tree and make choices, you will see a specific path from one node to another and the impact a decision made now could have down the road.

IX. Introduction

o K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on Supervised Learning technique.
o K-NN algorithm assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories.
o K-NN algorithm stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using K- NN algorithm.

X. How does K-NN work?

The K-NN working can be explained on the basis of the below algorithm:
o Step-1: Select the number K of the neighbors
o Step-2: Calculate the Euclidean distance of K number of neighbors
o Step-3: Take the K nearest neighbors as per the calculated Euclidean distance.
o Step-4: Among these k neighbors, count the number of the data points in each category.
o Step-5: Assign the new data points to that category for which the number of the neighbor is maximum
. o Step-6: Our model is ready.

XI. Advantages of KNN Algorithm:

o It is simple to implement.
o It is robust to the noisy training data
o It can be more effective if the training data is large.

XII. Disadvantages of KNN Algorithm:

o Always needs to determine the value of K which may be complex some time.
o The computation cost is high because of calculating the distance between the data points for all the training samples.

XIII. Code For KNN

!/usr/bin/env python
coding: utf-8

In[14]:

Import necessary modules
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model$_s election import train_t est_s plit$
$from sklearn.datasets import load_i ris$

Loading data
irisData = load$_i ris()$

Create feature and target arrays
X = irisData.data
y = irisData.target

Split into training and test set
$X_t rain, X_t est, y_t rain, y_t est = train_t est_s plit($
$X, y, test_s ize = 0.2, random_s tate = 42)$

knn = KNeighborsClassifier(n$_n eighbors = 7)$

knn.fit($X_t rain, y_t rain$)

Predict on dataset which model has not seen before
print(knn.predict($X_t est$))

In[13]:

Import necessary modules
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model$_s election import train_t est_s plit$
$from sklearn.datasets import load_i ris$

Loading data
irisData = load$_i ris()$

Create feature and target arrays
X = irisData.data
y = irisData.target

Split into training and test set
$X_t rain, X_t est, y_t rain, y_t est = train_t est_s plit($
$X, y, test_s ize = 0.2, random_s tate = 42)$

knn = KNeighborsClassifier(n$_n eighbors = 7)$

knn.fit($X_t rain, y_t rain$)

Calculate the accuracy of the model
print(knn.score($X_t est, y_t est$))

In[12]:

Import necessary modules
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model$_s election import train_t est_s plit$
$from sklearn.datasets import load_i ris$
$import numpy as np$
$import matplotlib.pyplot as plt$

irisData = load$_i ris()$

Create feature and target arrays
X = irisData.data
y = irisData.target

Split into training and test set
$X_t rain, X_t est, y_t rain, y_t est = train_t est_s plit($
$X, y, test_s ize = 0.2, random_s tate = 42)$

neighbors = np.arange(1, 9)
train$_a ccuracy = np.empty(len(neighbors))$
$test_a ccuracy = np.empty(len(neighbors))$

Loop over K values
for i, k in enumerate(neighbors):
knn = KNeighborsClassifier(n$_n eighbors = k)$
$knn.fit(X_t rain, y_t rain)$

Compute training and test data accuracy
train$_a ccuracy[i] = knn.score(X_t rain, y_t rain)$
$test_a ccuracy[i] = knn.score(X_t est, y_t est)$

Generate plot
plt.plot(neighbors, test$_a ccuracy, label = '$
$Testing dataset Accuracy')$
$plt.plot(neighbors, train_a ccuracy, label = '$
$Training dataset Accuracy')$

plt.legend()
plt.xlabel('n$_n eighbors')$
$plt.ylabel('Accuracy')$
$plt.show()$

In[ ]:

XIV. Conclusion

The K Nearest Neighbors algorithm doesn't require any additional training when new data becomes available. Rather it determines the K closest points according to some distance metric (the samples must reside in memory). Then, it looks at the target label for each of the neighbors and places the new found data point into the same category as the majority. Given that KNN computes distance, it's imperative that we scale our data. In addition, since KNN disregards the underlying features, it's our responsibility to filter out any features that are deemed irrelevant.

ACKNOWLEDGMENT

I would like to thank my honourable**Khan Md. Hasib Sir** for his time, generosity and critical insights into this project.