Applications of ML HW 2

By Srilalith (nsrilalith)

# Part 2:

## Code:

from sklearn import tree

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, confusion\_matrix

import pandas as pd

import pydotplus

def writegraphtofile(clf, featurelabels, filename):

dot\_data = tree.export\_graphviz(clf, feature\_names=featurelabels, out\_file=None)

graph = pydotplus.graph\_from\_dot\_data(dot\_data)

graph.write\_png(filename)

data = pd.read\_excel("AlienMushrooms.xlsx")

clf = tree.DecisionTreeClassifier(criterion="entropy")

feature\_labels = data.columns[:-1]

features = data[feature\_labels]

target\_label = data.columns[-1]

target = data[target\_label]

clf.fit(features, target)

writegraphtofile(clf, feature\_labels, "Assignment\_2" + "tree\_pic.png")

predictions = clf.predict(features)

# Calculate metrics

accuracy = accuracy\_score(target, predictions)

precision = precision\_score(target, predictions, average='macro') # Use 'macro' for multi-class classification

recall = recall\_score(target, predictions, average='macro')

f1 = f1\_score(target, predictions, average='macro')

conf\_matrix = confusion\_matrix(target, predictions)

# Print the metrics

print(f"Accuracy: {accuracy:.4f}")

print(f"Precision: {precision:.4f}")

print(f"Recall: {recall:.4f}")

print(f"F1 Score: {f1:.4f}")

print("Confusion Matrix:")

print(conf\_matrix)

## Console Output:

/usr/local/bin/python3.11 /Users/srilalithnampally/Classes/AppML\_Assignments/Assignment\_2/Q2.py

Accuracy: 0.9167

Precision: 0.8992

Recall: 0.8992

F1 Score: 0.8992

Confusion Matrix:

[[ 6 1]

[ 1 16]]

Process finished with exit code 0

## Image of Tree:

A diagram of a algorithm

Description automatically generated

## Discussion on Part 1 Tree vs Part 2 Tree:

The tree structure produced from manual calculations is identical to that generated by the software. However, there appears to be a labeling reversal in the code-generated tree, where 'Frilly=0' is labeled as True and 'Frilly=1' as False, which might make the manually calculated tree seem like a mirror image of the software-generated one. Upon examining the tree's leaves, it's evident that nodes with zero entropy are reached at the same decision level in both trees. Additionally, the manual calculations accurately reflect the count of samples filtered at each decision point.

# Part 3:

## Code:

import numpy as np

from sklearn import tree

from sklearn.preprocessing import OneHotEncoder

from sklearn.preprocessing import OrdinalEncoder

from sklearn.impute import SimpleImputer

from sklearn.pipeline import Pipeline

from sklearn.compose import ColumnTransformer

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

import pandas as pd

import pydotplus

def loadData(filename):

# Load data

data = pd.read\_excel(filename)

return data

def writegraphtofile(clf, featurelabels, filename):

dot\_data = tree.export\_graphviz(clf, feature\_names=featurelabels, out\_file=None)

graph = pydotplus.graph\_from\_dot\_data(dot\_data)

graph.write\_png(filename)

def preprocessData(data):

mix\_col = []

for col in data.columns:

unique\_types = data[col].apply(type).unique()

if len(unique\_types) > 1:

mix\_col.append(col)

data[mix\_col] = data[mix\_col].astype(str)

data.replace("?", np.nan, inplace=True)

# Separate features and target

X = data.iloc[:, :-1] # All columns except the last one

y = data.iloc[:, -1] # The last column

X = X.drop(['encounter\_id', 'patient\_nbr'], axis=1)

# Identify categorical columns (modify this list based on your dataset)

categorical\_features = ["race", "gender", "age", "weight", "admission\_type\_id", "discharge\_disposition\_id",

"admission\_source\_id", "payer\_code", "medical\_specialty", "diag\_1", "diag\_2", "diag\_3",

'max\_glu\_serum', 'A1Cresult', 'change', 'diabetesMed'] # Example column names

# Identify Ordinal columns (for medications)

ordinal\_features\_medications = ['metformin', 'repaglinide', 'nateglinide', 'chlorpropamide', 'glimepiride',

'acetohexamide', 'glipizide', 'glyburide', 'tolbutamide', 'pioglitazone',

'rosiglitazone', 'acarbose', 'miglitol', 'troglitazone', 'tolazamide', 'examide',

'citoglipton', 'insulin', 'glyburide-metformin', 'glipizide-metformin',

'glimepiride-pioglitazone', 'metformin-rosiglitazone', 'metformin-pioglitazone', ]

medication\_order = [['No', 'Steady', 'Up', 'Down']]

medication\_order\_for\_all\_columns = medication\_order \* len(ordinal\_features\_medications)

# Imputers for categorical and ordinal features

categorical\_imputer = SimpleImputer(strategy='constant', fill\_value='unknown')

ordinal\_imputer = SimpleImputer(strategy='most\_frequent')

# Identify numerical columns (assuming all other columns are numerical if not categorical or ordinal)

numerical\_features = [col for col in X.columns if col not in categorical\_features + ordinal\_features\_medications]

# Imputers

numerical\_imputer = SimpleImputer(strategy='mean') # Impute numerical columns with their mean

# Encoder

one\_hot\_encoder = OneHotEncoder()

ordinal\_encoder = OrdinalEncoder(categories=medication\_order\_for\_all\_columns)

# Update ColumnTransformer to include numerical imputation

preprocessor = ColumnTransformer(

transformers=[

('cat', Pipeline([('imputer', categorical\_imputer), ('encoder', one\_hot\_encoder)]), categorical\_features),

('ord\_meds', Pipeline([('imputer', ordinal\_imputer),

('encoder', ordinal\_encoder)]),

ordinal\_features\_medications),

('num', numerical\_imputer, numerical\_features)

],

remainder='passthrough' # Ensure no column is left unprocessed

)

# Check for NaN values post-imputation

print("Finished Pre-Processing, Starting Transformation")

# Apply transformations

X\_encoded = preprocessor.fit\_transform(X)

# Extracting new feature names after preprocessing

feature\_names = []

# Getting feature names for categorical features

for cat\_feature, one\_hot in zip(categorical\_features,

preprocessor.named\_transformers\_['cat']['encoder'].categories\_):

feature\_names.extend([f"{cat\_feature}\_{category}" for category in one\_hot])

# Adding ordinal feature names as is

feature\_names.extend(ordinal\_features\_medications)

# Adding numerical feature names as is

feature\_names.extend(numerical\_features)

print("Finished Transformation of dataset")

return X\_encoded, y, feature\_names

def MultiClassPrediction(X\_encoded, y):

# Split data for the multiclass classification model

X\_train\_multi, X\_test\_multi, y\_train\_multi, y\_test\_multi = train\_test\_split(X\_encoded, y, test\_size=0.2,

random\_state=42)

# Initialize and train the multiclass classifier

clf\_multiclass = tree.DecisionTreeClassifier(criterion="entropy", max\_depth=4)

clf\_multiclass.fit(X\_train\_multi, y\_train\_multi)

# Predictions for multiclass classification

predictions\_multi = clf\_multiclass.predict(X\_test\_multi)

predictions\_multi\_train = clf\_multiclass.predict(X\_train\_multi)

# Evaluate the multiclass model

accuracy\_multi = accuracy\_score(y\_test\_multi, predictions\_multi)

print(f"Multiclass Classification Accuracy with Test Set: {accuracy\_multi:.4f}")

accuracy\_multi\_train = accuracy\_score(y\_train\_multi, predictions\_multi\_train)

print(f"Multiclass Classification Accuracy with Train Set: {accuracy\_multi\_train:.4f}")

# Plotting the multiclass classification tree

return clf\_multiclass

def BinaryClassification(X\_encoded, y):

# Binary target preprocessing

y\_binary = y.replace({'NO': 0, '<30': 1, '>30': 1})

# Split data for the binary classification model

X\_train\_bin, X\_test\_bin, y\_train\_bin, y\_test\_bin = train\_test\_split(X\_encoded, y\_binary, test\_size=0.2,

random\_state=42)

# Initialize and train the binary classifier

clf\_binary = tree.DecisionTreeClassifier(criterion="entropy", max\_depth=4)

clf\_binary.fit(X\_train\_bin, y\_train\_bin)

# Predictions for binary classification

predictions\_bin\_test = clf\_binary.predict(X\_test\_bin)

predictions\_bin\_train = clf\_binary.predict(X\_train\_bin)

# Evaluate the binary model

accuracy\_bin\_test = accuracy\_score(y\_test\_bin, predictions\_bin\_test)

print(f"Binary Classification Accuracy with Test Set: {accuracy\_bin\_test:.4f}")

accuracy\_bin\_train = accuracy\_score(y\_train\_bin, predictions\_bin\_train)

print(f"Binary Classification Accuracy with Train Set: {accuracy\_bin\_train:.4f}")

# Plotting the binary classification tree

return clf\_binary

if \_\_name\_\_ == "\_\_main\_\_":

file\_location = "diabetic\_data.xlsx"

dataframe = loadData(file\_location) # created a dataframe

feature\_labels = dataframe.columns[:-1].tolist()

X, Y, feature\_names = preprocessData(dataframe) # Preprocessing the dataframe, and obtaining (features, target)

print("Dataset Is Ready \n")

print("Training Dataset")

clf\_binary = BinaryClassification(X, Y)

writegraphtofile(clf\_binary, feature\_names, "Assignment\_2" + "tree\_pic\_binary.png")

clf\_multiclass = MultiClassPrediction(X, Y)

writegraphtofile(clf\_multiclass, feature\_names, "Assignment\_2" + "tree\_pic\_multiclass.png")

## Console Output:

/usr/local/bin/python3.11 /Users/srilalithnampally/Classes/AppML\_Assignments/Assignment\_2/Q3.py

Finished Pre-Processing, Starting Transformation

Finished Transformation of dataset

Dataset Is Ready

Training Dataset

/Users/srilalithnampally/Classes/AppML\_Assignments/Assignment\_2/Q3.py:130: FutureWarning: Downcasting behavior in `replace` is deprecated and will be removed in a future version. To retain the old behavior, explicitly call `result.infer\_objects(copy=False)`. To opt-in to the future behavior, set `pd.set\_option('future.no\_silent\_downcasting', True)`

y\_binary = y.replace({'NO': 0, '<30': 1, '>30': 1})

Binary Classification Accuracy with Test Set: 0.6209

Binary Classification Accuracy with Train Set: 0.6234

Multiclass Classification Accuracy with Test Set: 0.5748

Multiclass Classification Accuracy with Train Set: 0.5735

Process finished with exit code 0

## Images Generated:

### Binary Classification:

A diagram of a number of numbers

Description automatically generated with medium confidence

### Multiclass Classification:

A diagram of a data flow

Description automatically generated

## Discussion on the differences between the Trees:

* Entropy: The multi-class tree has higher entropy values, indicating more complexity and less purity at each node compared to the binary tree.
* Leaf Purity: The binary tree has leaves with zero entropy, showing perfect class separation, while the multi-class tree does not, suggesting some classes remain mixed at the leaves.
* Class Counts in Leaves: The binary tree's leaves show counts for two classes, whereas the multi-class tree's leaves have counts for multiple classes, reflecting the complexity of separating more classes.
* Tree Structure: The binary tree might be simpler with fewer splits to achieve class separation, while the multi-class tree may require more splits to handle additional classes.
* Decision Boundaries: The binary tree suggests clearer decision boundaries with leaves often fully belonging to one class, whereas the multi-class tree indicates less distinct boundaries due to the presence of multiple classes.