CYBR 486 - Lab #6 – Decision Trees

This lab will take a look at 2 primary things. In previous labs you've been asked to display information about the data types in a data frame from a dataset, as well as determining if there are any null or missing values. The dataset for this lab still won't have any nulls but you should notice that all columns are strings even the target label column is all strings. We know that string values by themselves don't work when training and generating predictions with these models. Given that, while there will be some very familiar steps to this lab, the 2 new things you'll be doing is encoding string columns into usable numeric values, and then using the encoded data frame to create a few decision trees, ending with evaluating them like we've done previously.

Imports and Documentation

```
import pandas as pd

from sklearn.preprocessing import LabelEncoder, OrdinalEncoder

# https://scikit-learn.org/stable/modules/preprocessing.html#encoding-categorical-features

from sklearn.metrics import accuracy_score, precision_score, recall_score, classification_report

# https://scikit-learn.org/stable/modules/model_evaluation.html

from sklearn.model_selection import train_test_split

# https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
```

```
from sklearn.tree import DecisionTreeClassifier, export_graphviz

# https://scikit-learn.org/dev/modules/generated/sklearn.tree.DecisionTreeClassifier.html

import graphviz
```

I left the imports for precision and precision and recall score as I believe there are ways you can make them work for evaluation of a decision tree, but the code example I'll give uses classification_report() instead, as it gives a summary of multiple evaluation metrics in a handy formatted table. The other new imports are, as per usual, the new model for DecisionTreeClassifier(), and some imports for graphviz, a library we'll be using to visually display the decision tree at the end of the lab.

Tasks for this lab:

1) View the summary of the dataset

2) Preprocess the categorical variables in this dataset and encode them into numeric representation.

3) split the dataset into 80% training and 20% testing sets

4) Use the training dataset to build two decision tree models, one based on 'gini index' criterion and other based on 'entropy' criterion.

5) evaluate the performance of two models using the test dataset measuring the accuracy, recall and precision

6) visualize the decision-trees models with graphviz

2.) Starting with step 2 as we've done step 1 many times at this point, and it should be self-explanatory. We will be doing some dataset preprocessing.

Start by looking at the data types of the feature columns. As I mentioned at the top of the lab, all the feature columns contain String objects, and we need to encode them into numeric values for the model to train. Before we start, we need to talk about what types of encoding we'll be using.

I'll briefly touch on one-hot encoding though we won't be using it for this lab, it is a common form of encoding for dataset preprocessing.

https://scikit-learn.org/dev/modules/generated/sklearn.preprocessing.OneHotEncoder.html

One-hot encoding involves taking a feature column of categorical data, (like a feature column where all the entries are 'red', 'blue', or 'green'), and creates new sort-of sub-columns. If the original column was "colors" with the values mentioned above, then the result would be the removal of the "colors" column, and inserting where it was, 3 new columns with titles "colors_red", "colors_blue", and "colors_green".

This works fairly well when the values in the feature columns aren't considered "ordered", in that you can't, at least don't by default, associate some hierarchical order between the possible options in that column. If you look at the car_evaluation dataset we're using for this lab however, you'll notice that the features have entries like "low, mid, high" or "2, 4, more". We would consider columns like this to have a form of order to them (even if the data isn't sorted that way) which brings us to the 2nd type of encoding you'll commonly see,

Ordinal Encoding – With this we can specify a specific order to the possible values which will hopefully help the model make correct decisions about patterns in the dataset. What ordinal encoding does is take as an argument, an ordered list of options and assign a value starting at 0.0, and going up by 1 as it encounters Strings in the column. It is basically substituting ordered numerical values for the String values.

Label Encoding – I'll cover this one more in depth when we get to it. But this one works just like ordinal, except it doesn't care about any specific order and just substitutes increasing numerical values (0.0, 1.0, 2.0, etc.) as it comes across new unique entries in the label or target column. We only want to use this for the y subset, or the target/label subset.

In order to let the ordinal encoding library work, we need to manually define the possible values and the order they should be in for each feature.

```
buying_price_order = ['low', 'med', 'high', 'vhigh']
```

Above is an example of the custom ordering. You generally want to stick to small → big, or big → small. So long as it's consistent across all feature columns with increasing or decreasing order.

Using the pattern above, create ordered lists for the other 5 feature columns

```
ordinal_encoder = OrdinalEncoder(categories=[buying_price_order, maintenance_cost_order,

number_of_doors_order, number_of_persons_order, lug_boot_order, safety_order])

car_X_encoded = ordinal_encoder.fit_transform(car_X)

print(car_X_encoded)

# notice how the dataframe isn't a dataframe anymore, its just a multi-dimensional array/list, we need

to turn it back into a dataframe
```

Next, we create the encoder object, and pass it the ordered lists in the order the columns appear in the dataset. Next, we create the encoded X set using fit_transform(), but you'll notice if you print the output, its no longer in the format of a data frame, so we need to turn it back into one.

```
encoded_column_list = ["buying_price_encoded", "maintenance_cost_encoded",

"number_of_doors_encoded", "number_of_persons_encoded", "lug_boot_encoded",

"safety_encoded"]

car_X_encoded = pd.DataFrame(car_X_encoded, columns=encoded_column_list)

print(car_X_encoded)
```

This reattaches the column names to our encoded feature data.

Next, we'll use a label encoder to similarly encode the target column as well.

```
# we don't really need to order the categories of the target column so we can just let the library do

the work for us

label_encoder = LabelEncoder()



# we call fit_transform() on the column from car_y because at this point its still a pandas dataframe

object, and fit_transform() wants a 1-d array, even though it only has 1 column

car_y = label_encoder.fit_transform(car_y['decision'])

print(car_y)
```

```
# encoding the target subset also outputs it as a 1-d array so we need to turn it back into a

dataframe just like the X set

car_y_encoded = pd.DataFrame(car_y, columns=['decision_encoded'])

print(car_y_encoded)
```

There are some applications where you can leave the encoded sets as is without turning them back into pandas data frames, but it's a skill that you should be fairly familiar with.

```
C:\Users\Bhavy\PycharmProjects\lab44\venv\Scripts\python.exe C:/Users/Bhavy/PycharmProjects/lab44/main.py
    buying_price_encoded  maintenance_cost_encoded  number_of_doors_encoded  number_of_persons_encoded  lug_boot_encoded  safety_encoded
0                    3.0                       3.0                      0.0                        0.0               0.0             0.0
1                    3.0                       3.0                      0.0                        0.0               0.0             1.0
2                    3.0                       3.0                      0.0                        0.0               0.0             2.0
3                    3.0                       3.0                      0.0                        0.0               1.0             0.0
4                    3.0                       3.0                      0.0                        0.0               1.0             1.0
5                    3.0                       3.0                      0.0                        0.0               1.0             2.0
6                    3.0                       3.0                      0.0                        0.0               2.0             0.0
7                    3.0                       3.0                      0.0                        0.0               2.0             1.0
8                    3.0                       3.0                      0.0                        0.0               2.0             2.0
9                    3.0                       3.0                      0.0                        1.0               0.0             0.0
10                   3.0                       3.0                      0.0                        1.0               0.0             1.0
11                   3.0                       3.0                      0.0                        1.0               0.0             2.0
12                   3.0                       3.0                      0.0                        1.0               1.0             0.0
13                   3.0                       3.0                      0.0                        1.0               1.0             1.0
14                   3.0                       3.0                      0.0                        1.0               1.0             2.0
15                   3.0                       3.0                      0.0                        1.0               2.0             0.0
16                   3.0                       3.0                      0.0                        1.0               2.0             1.0
17                   3.0                       3.0                      0.0                        1.0               2.0             2.0
18                   3.0                       3.0                      0.0                        2.0               0.0             0.0
19                   3.0                       3.0                      0.0                        2.0               0.0             1.0
20                   3.0                       3.0                      0.0                        2.0               0.0             2.0
21                   3.0                       3.0                      0.0                        2.0               1.0             0.0
22                   3.0                       3.0                      0.0                        2.0               1.0             1.0
23                   3.0                       3.0                      0.0                        2.0               1.0             2.0
24                   3.0                       3.0                      0.0                        2.0               2.0             0.0
25                   3.0                       3.0                      0.0                        2.0               2.0             1.0
26                   3.0                       3.0                      0.0                        2.0               2.0             2.0
27                   3.0                       3.0                      1.0                        0.0               0.0             0.0
28                   3.0                       3.0                      1.0                        0.0               0.0             1.0
29                   3.0                       3.0                      1.0                        0.0               0.0             2.0
30                   3.0                       3.0                      1.0                        0.0               1.0             0.0
31                   3.0                       3.0                      1.0                        0.0               1.0             1.0
```

```
      decision_encoded
0                    2
1                    2
2                    2
3                    2
4                    2
5                    2
6                    2
7                    2
8                    2
9                    2
10                   2
11                   2
12                   2
13                   2
14                   2
15                   2
16                   2
17                   2
18                   2
19                   2
20                   2
21                   2
```

3.) I'll leave splitting the set into training and test sets as you should be pretty familiar with it at this point and it works the same, except you're doing it not with the original X and y parts but their encoded counterparts.

```
C:\Users\Bhavy\PycharmProjects\lab44\venv\Scripts\python.exe C:/Users/Bhavy/PycharmProjects/lab44/3.py
Training feature set size: (1382, 6)
Testing feature set size: (346, 6)
Training target set size: (1382, 1)
Testing target set size: (346, 1)

Process finished with exit code 0
```

4.) Build, train, and generate predictions for 2 decisions trees.

```
# QUESTION 4a.)  CREATE AND TRAIN A DECISION TREE BASED ON 'ENTROPY' AS THE

CRITERION

maxDepth = 6

classifier_tree_entropy = DecisionTreeClassifier(criterion='entropy', max_depth=maxDepth)

classifier_tree_entropy.fit(X_train, y_train)

predictions = classifier_tree_entropy.predict(X_test)
```

As with the other models, its pretty simple to create and train the decision tree classifier model. The first one will be using entropy to make decisions as was covered in the lecture slides. The other argument we use for this is a max_depth. At the end of the lab, we'll use a tool to display a diagram of the decision tree generated by the model, and without setting a max depth…. The trees can get… a bit out of control.

The other model we'll make is a "gini index" decision tree. I won't get too in depth about how the Gini Index works, but it's measuring the impurity of a dataset, by calculating the probability of getting the prediction for a randomly selected sample from the dataset incorrect. It is created identically to how the entropy one is above.

Using above as a reference, create and train the “gini index” decision tree.

Basically, everything is the same but the criterion changes to: criterion=”gini”

```
C:\Users\Bhavy\PycharmProjects\lab44\venv\Scripts\python.exe C:/Users/Bhavy/PycharmProjects/lab44/4.py
Decision Tree using 'Entropy':
Accuracy: 0.9132947976878613
Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.73      0.81        83
           1       0.45      0.82      0.58        11
           2       0.96      0.98      0.97       235
           3       0.79      0.88      0.83        17

    accuracy                           0.91       346
   macro avg       0.78      0.85      0.80       346
weighted avg       0.93      0.91      0.92       346

Decision Tree using 'Gini':
Accuracy: 0.9393063583815029
Classification Report:
              precision    recall  f1-score   support

           0       0.96      0.81      0.88        83
           1       0.45      0.82      0.58        11
           2       0.99      1.00      0.99       235
           3       0.79      0.88      0.83        17

    accuracy                           0.94       346
   macro avg       0.80      0.88      0.82       346
weighted avg       0.95      0.94      0.94       346


Process finished with exit code 0
```

5.) Evaluate the trees

Evaluation works pretty similarly to how we did the previous lab with accuracy_score.

Find both the accuracy score individually, and generate the classification report using the methods imported at the start of the lab. Classification_report() works the same way accuracy_score() does with arguments.

```
C:\Users\Bhavy\PycharmProjects\lab44\venv\Scripts\python.exe C:/Users/Bhavy/PycharmProjects/lab44/4.py
Decision Tree using 'Entropy':
Accuracy: 0.9132947976878613
Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.73      0.81        83
           1       0.45      0.82      0.58        11
           2       0.96      0.98      0.97       235
           3       0.79      0.88      0.83        17

    accuracy                           0.91       346
   macro avg       0.78      0.85      0.80       346
weighted avg       0.93      0.91      0.92       346

Decision Tree using 'Gini':
Accuracy: 0.9393063583815029
Classification Report:
              precision    recall  f1-score   support

           0       0.96      0.81      0.88        83
           1       0.45      0.82      0.58        11
           2       0.99      1.00      0.99       235
           3       0.79      0.88      0.83        17

    accuracy                           0.94       346
   macro avg       0.80      0.88      0.82       346
weighted avg       0.95      0.94      0.94       346


Process finished with exit code 0
```

6.) Use Graphviz to display one of your decision trees.

Finally we're going to use one of our trained decision tree models and plug that into a tool called Graphviz to display a diagram of the tree generated during training.

Tool Link: https://dreampuf.github.io/GraphvizOnline/

We'll create .dot file in our code, then copy the contents of that file into the link above to generate the tree.

```
classNames = ["unacc", "acc", "good", "vgood"]

dot_data = export_graphviz(classifier_tree_entropy,

                           out_file="tree.dot",

                           feature_names=car_X.columns.to_list(),
```

```
                class_names=classNames,

                filled=True,

                rounded=True,

                special_characters=True)



graph = graphviz.Source(dot_data)
```

I should note that there are ways to get this library to work in your ide but it was more finnicky than I'd like so I found an alternative with the online tool and that works just fine. Run this code, changing the decision tree model to whatever variable you used for the trained model, it'll create a .dot file called "tree.dot", copy the text contents of that file and paste them into the left side of the tool link. This will generate the diagram on the right, you can then download it as a .png and upload to canvas.

I encourage you to at this point to play around with the max_depth value that we set when creating and training the tree models. This isn't required for the lab but it's interesting to see the effects on the trees as depths change.

Entropy tree.pdf attached as answer, all code can be found in zip.

Take a screenshot of each major step of the lab and either attach to this document and reupload, or attach them to another docx or pdf file and upload that.

Finally answer this question: What do you think would happen, given the fact that there are 6 feature columns, if the maximum depth of the tree was 3? How about if the maximum depth was 12? How does depth affect how the model learns to build the tree?

A maximum depth of **3** would likely cause **underfitting**, as the tree would be too shallow to fully capture the relationships between the 6 feature columns, resulting in a model that is too simple and may miss important patterns in the data. Conversely, a maximum depth of **12** would likely cause **overfitting**, where the tree becomes too complex, fitting the training data too closely, including noise, and performing poorly on unseen data. The tree depth impacts the **bias-variance tradeoff**: shallow trees tend to have **higher bias** and **lower variance**, making them less prone to overfitting but potentially missing important patterns, while deeper trees have **lower bias** and **higher variance**, increasing the risk of overfitting and poor generalization. Finding the right depth is key to balancing model complexity and accuracy.