Name and ID

Mengqi Irina Wang 1278675/mwang17

HW05 Code

You will complete the following notebook, as described in the PDF for Homework 05 (included in the download with the starter code). You will submit:

1. This notebook file, along with your COLLABORATORS.txt file and the two tree images (PDFs generated using `graphviz` within the code), to the Gradescope link for code.
2. A PDF of this notebook and all of its output, once it is completed, to the Gradescope link for the PDF.

Please report any questions to the class Piazza page.

Import required libraries.

In [1]:

```
import numpy as np
import pandas as pd

import sklearn.tree
import graphviz

from sklearn.datasets import make_classification
from sklearn.feature_selection import SelectFromModel
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score

from sklearn import set_config
set_config(print_changed_only=False)

import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Decision Trees

You should start by computing the two heuristic values for the toy data described in the assignment handout. You should then load the two versions of the abalone data, compute the two heuristic values on features (for the simplified data), and then build decision trees for each set of data.

1 Compute both heuristics for toy data.

In [2]:

```
toy = np.array(
   [[1, 1, 0],
    [1, 1, 0],
    [0, 1, 0],
    [0, 0, 0],
    [0, 1, 1],
    [0, 0, 1],
    [0, 0, 1],
    [0, 0, 1]])
toy_features = np.array(['A', 'B'])

toy_X = toy[:, :2]
toy_y = toy[:, 2]

num_feat = len(toy_X[0])
```

(a) Compute the counting-based heuristic, and order the features by it.

In [3]:

```
feature_acc = np.zeros(num_feat)
total_data = len(toy_X)
```

In [4]:

```
for i in range(num_feat):
    toy_x = toy_X[:,i].reshape(-1, 1)
    model = sklearn.tree.DecisionTreeClassifier()
    model.fit(toy_x, toy_y)
    pred = model.predict(toy_x)
    feature_acc[i] = len(np.where(pred == toy_y)[0])
```

In [5]:

```
seq = np.argsort(-feature_acc)
for i in seq:
    print('%s: %i / %i' % (toy_features[i], feature_acc[i], total_data))
```

```
A: 6 / 8
B: 6 / 8
```

(b) Compute the information-theoretic heuristic, and order the features by it.

In [6]:

```
def calc_entropy(p,n):
    prob_p = p / (p+n)
    prob_n = n / (p+n)

    class1 = prob_p*(np.log2(prob_p)) if (prob_p != 0) else 0
    class2 = prob_n*(np.log2(prob_n)) if (prob_n != 0) else 0
    h = - (class1 + class2)

    return h
```

In [7]:

```
pre_pos = len(np.nonzero(toy_y)[0])
pre_neg = total_data - len(np.nonzero(toy_y)[0])

pre_entro = calc_entropy(pre_pos,pre_neg)
```

In [8]:

```
post_entro = np.zeros(num_feat)
```

In [9]:

```
def remainder_entropy():
    for i in range(num_feat):
        post_true = np.where(toy_X[:, i] == 1)
        post_false = np.where(toy_X[:, i] == 0)

        post_true_pos = len(np.where(toy_y[post_true] == 1)[0])
        post_true_neg = len(np.where(toy_y[post_true] == 0)[0])
        post_false_pos = len(np.where(toy_y[post_false] == 1)[0])
        post_false_neg = len(np.where(toy_y[post_false] == 0)[0])

        prob_post_true =  (post_true_pos + post_true_neg) / total_data
        prob_post_false =  (post_false_pos + post_false_neg) / total_data

        post_entro[i] = prob_post_true * calc_entropy(post_true_pos,post_true_neg) + prob_post_false * calc_ent


remainder_entropy()
```

In [10]:

```
info_gain = np.zeros(num_feat)
def calc_info_gain():
    for i in range(num_feat):
        info_gain[i] = pre_entro - post_entro[i]

calc_info_gain()
```

In [11]:

```
seq = np.argsort(-info_gain)
for i in seq:
    print('%s: %.3f' % (toy_features[i], info_gain[i]))
```

```
A: 0.311
B: 0.189
```

(c) Discussion of results.

The result shows that both features perform equally good in terms of correctness(counting-based heuristic), thus, using solely the counting-based heuristic, we won't be able to decide which feature is better. However, the infomation gain method successfully separate and order the two features -- feature A is better at reducing entropy, as showed through its high information gain value. Thus, if we built a tree using each of these two heuristics, knowing they will be equally good in terms of the counting-based heuristic, feature A will give us more information on the toy dataset than feature B.

2 Compute both heuristics for simplified abalone data.

In [12]:

```
simple_x_train = np.loadtxt('data_abalone/small_binary_x_train.csv', skiprows=1, delimiter=',')
simple_x_test = np.loadtxt('data_abalone/small_binary_x_test.csv', skiprows=1, delimiter=',')

three_class_y_train = np.loadtxt('data_abalone/3class_y_train.csv', skiprows=1, delimiter=',')
three_class_y_test = np.loadtxt('data_abalone/3class_y_test.csv', skiprows=1, delimiter=',')

assert simple_x_train.shape[0] == three_class_y_train.shape[0]
assert simple_x_test.shape[0] == three_class_y_test.shape[0]
```

In [13]:

```
simple_features = np.array(['is_male','length_mm','diam_mm','height_mm'])
```

In [14]:

```
simple_classes = np.array([0,1,2])
```

In [15]:

```
simple_classes_labels = np.array(["small","medium","large"])
```

In [16]:
```
num_feat = len(simple_features)
```

In [17]:
```
num_class = len(simple_classes)
```

(a) Compute the counting-based heuristic, and order the features by it.

In [18]:
```
feature_correct = np.zeros(num_feat)
total_data = len(simple_x_train)
```

In [19]:
```
for i in range(num_feat):
    selected_simple_x_train = simple_x_train[:, i].reshape(-1,1)
    model = sklearn.tree.DecisionTreeClassifier()
    model.fit(selected_simple_x_train, three_class_y_train)
    pred = model.predict(selected_simple_x_train)
    feature_correct[i] = len(np.where(pred == three_class_y_train)[0])
```

In [20]:
```
seq = np.argsort(-feature_correct)
for i in seq:
    print('%s: %i / %i' % (simple_features[i], feature_correct[i], total_data))
```

```
height_mm: 2316 / 3176
diam_mm: 2266 / 3176
length_mm: 2230 / 3176
is_male: 1864 / 3176
```

(b) Compute the information-theoretic heuristic, and order the features by it.

In [21]:
```
def calc_entropy(data, num_class):
    total_sum = np.sum(data)
    class_entro = np.zeros(num_class)

    for i in range(num_class):
        class_entro[i] = (data[i] / total_sum) * (np.log2(data[i] / total_sum)) if (data[i] != 0) else 0

    h = - (np.sum(class_entro))
    return h
```

In [22]:
```
# pre -class data
pre_class = np.zeros(num_class)

for i in simple_classes:
    pre_class[i] = len(np.where(three_class_y_train == i)[0])
```

In [23]:
```
pre_entropy = calc_entropy(pre_class, 3)
```

In [24]:
```
def single_remainder_entropy(X, y, E_val, num_class):
    E = np.zeros([len(E_val), num_class])
    p = np.zeros(len(E_val))
    post_entropy = np.zeros(len(E_val))
    for k in E_val:
        E_k = np.where(X == k)
        for i in range(num_class):
            E[k][i] = len(np.where(y[E_k] == i)[0])

        post_entropy[k] = calc_entropy(E[k], num_class)
        p[k] = len(E_k[0]) / len(X)

        info_gain = pre_entropy - (p[0] * post_entropy[0] + p[1] * post_entropy[1])
    return info_gain
```

In [25]:
```
e = np.array([0,1])
simple_info_gain = np.zeros(num_feat)
for k in range(num_feat):
    simple_info_gain[k] = single_remainder_entropy(simple_x_train[:,k], three_class_y_train, e, num_class)
```

In [26]:
```
seq = np.argsort(-simple_info_gain)
for i in seq:
    print('%s: %.3f' % (simple_features[i], simple_info_gain[i]))
```

```
height_mm: 0.173
diam_mm: 0.150
length_mm: 0.135
is_male: 0.025
```

3 Generate decision trees for full- and restricted-feature data

In [27]:

```
x_train = np.loadtxt('data_abalone/x_train.csv', skiprows=1, delimiter=',')
x_test = np.loadtxt('data_abalone/x_test.csv', skiprows=1, delimiter=',')

y_train = np.loadtxt('data_abalone/y_train.csv', skiprows=1, delimiter=',')
y_test = np.loadtxt('data_abalone/y_test.csv', skiprows=1, delimiter=',')

assert x_train.shape[0] == y_train.shape[0]
assert x_test.shape[0] == y_test.shape[0]
```

(a) Print accuracy values and generate tree images.

In [28]:
```
simple_clf = sklearn.tree.DecisionTreeClassifier(criterion='entropy')
```

In [29]:
```
# simple train
simple_clf.fit(simple_x_train, three_class_y_train)
acc_simple_train = simple_clf.score(simple_x_train, three_class_y_train)
acc_simple_test = simple_clf.score(simple_x_test, three_class_y_test)

print('accuracy score for simple train data is %.3f' % acc_simple_train)
print('accuracy score for simple test data is %.3f' % acc_simple_test)
```

```
accuracy score for simple train data is 0.733
accuracy score for simple test data is 0.722
```

In [30]:
```
simple_tree = sklearn.tree.export_graphviz(simple_clf, class_names =simple_classes_labels,  filled=True)
graph = graphviz.Source(simple_tree)
graph.render("simple tree graph")
```

Out[30]:
```
'simple tree graph.pdf'
```

In [31]:
```
general_clf = sklearn.tree.DecisionTreeClassifier(criterion='entropy')
```

In [32]:
```
# train
general_clf.fit(x_train, y_train)
acc_train = general_clf.score(x_train, y_train)
acc_test = general_clf.score(x_test, y_test)

print('accuracy score for general train data is %.3f' % acc_train)
print('accuracy score for general test data is %.3f' % acc_test)
```

```
accuracy score for general train data is 1.000
accuracy score for general test data is 0.178
```

In [33]:

```
tree = sklearn.tree.export_graphviz(general_clf,  filled=True)
graph = graphviz.Source(tree)
graph.render("tree graph")
```

Out[33]:

```
'tree graph.pdf'
```

(b) Discuss the results seen for the two trees

Discuss the results you have just seen. What do the various accuracy-score values tell you? How do the two trees that are produced differ? Looking at the outputs (leaves) of the simplified-data tree, what sorts of errors does that tree make?

First, while the accuracy score for training data and testing data are relatively the same for the three-class-classification, when using the decision tree on the full data set, it performs poorly on the testing data and clearly overfits the training data with an accuracy score of 1.

Correspondingly, we see the leaves of the tree for the full data set having the entropy of 0, while the leaves for the tree on smaller dataset have relatively higher entropies. This means there's a tradeoff between the acurracy score and adaptability/generalization of the model, just as seen in the models we encountered in the previous semesters. When the tree perfectly fits to the training data(especially with many classes or features), it's hard to apply it to other data since it becomes very specific.

On the other hand, for the smaller dataset, because we preprocessed the data, and the range of the outcomes(rings) is narrowed to 3 classes, the tree becomes a lot simpler. The leaves of the simplified-data tree often have high entropy. When we color-fill and label the tree, all classes are either small or medium, which means that no features are able to separate the “large” class. The problem could be due to our dataset or the given features — either the dataset is heavily skewed and there are thousands of small/medium data and only 27 large data, or that we don’t have enough features or the hard threshold at 0.5 doesn’t have the power to successfully separate the dataset. Either way, although the performance is relatively consistent over training and testing dataset, the errors manifests as relatively high entropy in the leaf nodes.

In [ ]: