Name and ID

Sammy Muench

HW04 Code

You will complete the following notebook, as described in the PDF for Homework 04 (included in the download with the starter code). You will submit:

1. This notebook file (`hw04.ipynb`), `implementation.py` , and two files for both trees images, i.e., `full` , `full.pdf` , `simple` , and `simple.pdf` (PDFs and text files generated using `graphviz` within the code). HINT: `render()` , and it should be clear when to use it, i.e., #3). Compress all files mentioned and submit 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 [137…

```
import numpy as np
import pandas as pd

import sklearn.tree
import graphviz

from implementation import information_remainder, counting_heuristic, set_entropy

%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

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 [138…
```
feature_names = np.array(["A", "B"])
feature_len = 2
classes = [0, 1]

x_set = np.array([[1, 1], [1, 1], [0, 1], [0, 0],
        [0, 1], [0, 0], [0, 0], [0, 0]])
y_set = np.array([0, 0, 0, 0, 1, 1, 1, 1])
```

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

In [139…
```
# sort the feature names by their correct counts
sort_correct = [counting_heuristic(x_set, y_set, i, classes) for i in range(x_set.s
sort_names = ['B', 'A'] #TODO: FIX ME

# Print the sorted features along with their correct predictions count in the small
longest = max(len(name) for name in sort_names)
for name, correct in zip(sort_names, sort_correct):
    print("%*s: %d/%d" % (longest, name, correct, len(x_set)))
```

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

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

In [140…
```
#information_remainder()

gains_abalone = [[information_remainder(x_set, y_set,
                                        i, classes), feature] for i, feature

#sort the feature names by their gains
sort_gains_init = sorted(gains_abalone, key = lambda x: x[0], reverse=True)
sort_gains = [x[0] for x in sort_gains_init]
sort_names_by_gains = [x[1] for x in sort_gains_init]

longest = max(len(name) for name in sort_names_by_gains)
for name, gain in zip(sort_names_by_gains, sort_gains):
    print("%*s: %.3f" % (longest, name, gain))
```

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

(c) Discussion of results.

Splitting by A seems to give more information, despite them having the same counting heuristic. This makes sense because if A == 0, then y == 1 throughout the whole dataset, but for B, knowing its value cannot determine y regardless of whether it equals 1 or 0. Hence, A has more splitting power.

2 Compute both heuristics for simplified abalone data.

In [141…
```
# load the data into np arrays
# fix the empty lists below
# full-feature abalone data

def convert_to_arr_simple(l):

    ''' Converts the simple datasets to numpy arrays for faster computation
    '''
    return np.array([[elem[0], elem[1], elem[2], elem[3]] for elem in l])

def convert_to_arr_big(l):
    ''' Converts the big datasets to numpy arrays for faster computation
    '''
    return np.array([elem[0], elem[1], elem[2], elem[3], elem[4], elem[5], elem[6],

def convert_to_arr_y(l):
    ''' Converts the output datasets to numpy arrays for faster computation
    '''
    return np.fromiter((elem[0] for elem in l), dtype=float)

x_train = pd.read_csv('data_abalone/x_train.csv')
x_test = pd.read_csv('data_abalone/x_test.csv') #read_csv because this works best w
y_train = convert_to_arr_y(np.genfromtxt('data_abalone/y_train.csv', delimiter=',',
y_test = convert_to_arr_y(np.genfromtxt('data_abalone/y_test.csv', delimiter=',', n

# simplified version of the data (Restricted-feature)
simple_x_train = convert_to_arr_simple(np.genfromtxt('data_abalone/small_binary_x_t
simple_x_test = convert_to_arr_simple(np.genfromtxt('data_abalone/small_binary_x_te
simple_y_train = convert_to_arr_y(np.genfromtxt('data_abalone/3class_y_train.csv',
simple_y_test = convert_to_arr_y(np.genfromtxt('data_abalone/3class_y_test.csv', de

# get useful information
full_feature_names = list(np.genfromtxt('data_abalone/x_train.csv', delimiter=',',
simple_feature_names = list(np.genfromtxt('data_abalone/small_binary_x_test.csv', d
classes_abalone = np.array([0, 1, 2]) # unique set of class labels
class_names = np.array(['Small', 'Medium', 'Large']) # name of the classes
```

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

In [142…
```
correct_abalone = [[float(counting_heuristic(simple_x_train, simple_y_train, i, cla
                          for i, feature in enumerate(simple_feature_names)]

# sort the feature names by their correct counts
sorted_abalone_init = sorted(correct_abalone, key = lambda x: x[0], reverse=True)
sort_correct_abalone = [x[0] for x in sorted_abalone_init]
sort_names_abalone = [x[1] for x in sorted_abalone_init]

# Print the sorted features along with their correct predictions count in the small
longest = max(len(name) for name in sort_names_abalone)
for name, correct in zip(sort_names_abalone, sort_correct_abalone):
    print("%*s: %d/%d" % (longest, name, correct, len(simple_x_train)))
```

```
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 [143…
```
# information_remainder()

gains_abalone = [[information_remainder(simple_x_train, simple_y_train,
                                        i, classes_abalone), feature] for i,

# sort the feature names by their gains
sort_gains_abalone_init = sorted(gains_abalone, key = lambda x: x[0], reverse=True)
sort_gains_abalone = [x[0] for x in sort_gains_abalone_init]
sort_names_by_gains_abalone = [x[1] for x in sort_gains_abalone_init]

longest = max(len(name) for name in sort_names_by_gains_abalone)
for name, gain in zip(sort_names_by_gains_abalone, sort_gains_abalone):
    print("%*s: %.3f" % (longest, name, gain))
```

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

3) Generate decision trees (criterion='entropy', random_state=42) for full- and simple-feature data

(a) Train and eval on entire train and test sets. Print accuracy values and generate tree images.

Render the tree diagram, naming it "full." A text file and PDF should be created and saved (i.e., `full` and `full.pdf`) - include both in submission.

In [144…
```
clf = sklearn.tree.DecisionTreeClassifier(criterion='entropy', random_state=42)
clf.fit(x_train, y_train)

train_accuracy = clf.score(x_train, y_train)
test_accuracy = clf.score(x_test, y_test)
print(f"Accuracy (train): {train_accuracy:.3f}")
print(f"Accuracy  (test): {test_accuracy:.3f}")

# TODO generate tree image
dot_data = sklearn.tree.export_graphviz(clf, feature_names=full_feature_names)
graph = graphviz.Source(dot_data)
graph.render("full")
```

```
Accuracy (train): 1.000
Accuracy  (test): 0.204
```

Out[144]:
```
'full.pdf'
```

(b) Restricted-feature (aka simple) data.

Train and eval on simple train and test sets. Same as above, accept this time use the `simple` set. Render the tree diagram, naming it "simple." A text file and PDF should be created and saved (i.e., `simple` and `simple.pdf`) - include both in submission.

In [145…

```
clf.fit(simple_x_train, simple_y_train)

simple_train_accuracy = clf.score(simple_x_train, simple_y_train)
simple_test_accuracy = clf.score(simple_x_test, simple_y_test)
print(f"Accuracy (train): {simple_train_accuracy:.3f}")
print(f"Accuracy  (test): {simple_test_accuracy:.3f}")

# generate tree image
dot_data = sklearn.tree.export_graphviz(clf, feature_names=simple_feature_names, cl
graph = graphviz.Source(dot_data)
graph.render("simple")
```

```
Accuracy (train): 0.733
Accuracy  (test): 0.722
```

Out[145]:

```
'simple.pdf'
```

(c) Discuss the results seen for the two trees

On the restricted tree, it seems that the accuracy does much better than the bigger tree. This makes sense, as the bigger tree may be more prone to overfitting, since considering new features may cause the tree to grow exponentially.

The trees differ in obviously size, but also the leaves. The smaller tree has less variance, which caused it to have greater accuracy. However, there is no leaf that represents the Large class, which is not the case with the larger tree, as many examples are represented. However, the large tree is hard to interpret, as it is super wide in order to account for all of the possible combinations of features and possible leaves.