Project - Old bailey decisions

Decision tree implementation

```
import numpy as np
import pandas as pd
import math
import random
import matplotlib.pyplot as plt
%matplotlib inline
```

```
# These are the datasets with the misc attributes included, worked much better for perceptron
TRAINING_PATH = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-deci
TESTING_PATH = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decisi
EVAL_PATH = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decisions
EVAL_IDS = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decisions/

# These are my updated misc.. let's see how they compare
TRAINING_PATH = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-deci
TESTING_PATH = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decisi
EVAL_PATH = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decisions
EVAL_IDS = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decisions/


def load_ids(file_path):
  with open(file_path) as f:
    raw_data = [int(line.split()[0]) for line in f]
  # print(raw_data)
  return raw_data
```

```
Major challenge here will be converting data structures I have to data format
that I used for my hw implementation

Also, won't use glove or tfidf b/c these are real-valued, this seems tough
to convert to decision tree...would need to bin each. Wait, also applies to BOW.
Just gonne use misc features. I could convert features to work with dec-tree...
seems like kind of a pain rn tho

'or BOW, I would just need to adjust the decision tree algorithm to accomodate
different values + change data structure to be a full matrix



This is the data class I used for hw1, let's alter it to work with my current data
setup


.ass Data:
def __init__(self,path=None):
  self.path = path
  self.data = {}
```

```
  self.data = {}
  self.length = 0
  self.classes = None
  self.num_features = 0
  if self.path != None:
    self.data = self.load_data(self.path)
    self.length = len(self.data)
    label_arr = get_labels(self.data)
    self.classes = np.unique(label_arr)

def load_data(self,path):
  # pandas load csv
  raw_data = pd.read_csv(path)
  data = raw_data.to_numpy()
  self.num_features = data.shape[1]-1
  # print(raw_data)
  # print(data)
  # Will store as dictionary with label and features items with index as key
  data_dict = {}
  for index,line in enumerate(data):
    features = []
    for i,feat in enumerate(line[1:]):
      if feat != 0:
        features.append(str(i))
    if line[0] == 1:
      label = "+1"
    else:
      label = "-1"
    data_dict[index] = {
        'label': label,
        'features': features
    }
  return data_dict

def add_data(self,path_to_add):
  with open(path_to_add) as f:
    raw_data = [line.split() for line in f]
  new_index = self.length
  for index,line in enumerate(raw_data):
    features = []
    for feat in line[1:]:
      features.append(feat[:-2])
    self.data[index + new_index] = {
        'label': line[0],
        'features': features
    }
  # Update length, and classes
  self.length = len(self.data)
  label_arr = get_labels(self.data)
  self.classes = np.unique(label_arr)


ef get_labels(data):
# Returns list of labels given a data dict input
labels = []
for item in data.items():
```

```
  labels.append(item[1]['label'])
return labels

ef count_labels(data,classes):
# receives data and classes and returns label count
label_arr = get_labels(data)
label_counts = {}
for label in classes:
  label_counts[label] = label_arr.count(label)
return label_counts

ef split_on_attr(attr,data):
# returns two subsets of input data split on attribute
if attr == None:
  return data
has_attr = {}
no_attr = {}
for item in data.items():
  if attr in item[1]['features']:
    has_attr[item[0]] = item[1]
  else:
    no_attr[item[0]] = item[1]
return has_attr,no_attr

ef get_common_label(data):
labels = get_labels(data)
return max(set(labels), key = labels.count)



impurity measures and information gain related functions

Computing Gini
ef compute_gini(count_arr):
a,b = count_arr
total = a+b
if total == 0:
  return 0
frac_a = a/total
frac_b = b/total
arr = [frac_a,frac_b]
gini = 0
for i in range(len(arr)):
  gini += arr[i]*(1-arr[i])
return gini

Computing entropy
ef compute_entropy(count_arr):
# input is array of respective counts
# Given two integers representing number of respective labels,
# returns entropy
if len(count_arr) == 0:
  return 0
a, b = count_arr
if a==0 or b==0:
  return 0
```

```
total = a+b
frac_a = a/total
frac_b = b/total
return -frac_a*math.log2(frac_a) - frac_b*math.log2(frac_b)


Lets have this information gain function combine everything
f information_gain(data,attr_split,classes,purity_func,verbose=0):
# Compute entropy of entire data input
full_labels = list(count_labels(data,classes).values())
full_purity = purity_func(full_labels)
# Total instance count
S = len(data)
# print(data)
# Split data across indicated attribute
has_attr, no_attr = split_on_attr(attr_split,data)
# print(has_attr,no_attr)
# return attribute counts for each split
label_counts = [count_labels(has_attr,classes), count_labels(no_attr,classes)]
if verbose == 2: print("label counts of split on {}: {}".format(attr_split,label_counts))
sub_purity = 0
total_count = 0
for label_count in label_counts:
  counts = list(label_count.values())
  total_count += sum(counts)
  sub_purity += (sum(counts)/(S))*purity_func(counts)
if S != total_count:
  raise Exception(print("totals don't match up"))
if verbose == 1 or verbose == 2: print("Information gain splitting on {}: {}".format(attr_spli
return full_purity - sub_purity


Returns best attribute along with information gain given data and attributes to consider
f get_best_attr(data,attributes,classes,purity_func=compute_entropy):
max_information_gain = 0
best_attr = None
for attr in attributes:
  if information_gain(data,str(attr),classes,purity_func) > max_information_gain:
    max_information_gain = information_gain(data,str(attr),classes,purity_func)
    best_attr = attr
if best_attr == None:
  # just pick one
  best_attr = attributes[0]

return best_attr,max_information_gain



Classes for Tree

ass Node:
def __init__(self,label=None):
  self.level = None
  self.attribute = None
  self.information_gain = None
```

```
  self.label = label
  self.left = None # reference to left child node
  self.right = None # reference to right child node



ass DecisionTreeClassifier:
# pass in data object along with full list of attributes at first
def __init__(self,data,attrs,purity_measure=compute_entropy):
  self.data = data.data
  self.most_common_label = str(get_common_label(self.data))
  self.attrs = attrs
  self.classes = data.classes
  self.num_classes = len(data.classes)
  self.tree = None
  self.tree_depth = 0
  self.purity_measure = purity_measure

def build_tree(self,depth_limit=float('inf')):
  self.tree = self.train_tree(self.data,self.attrs,0,depth_limit)
  return self.tree

def train_tree(self,data,attrs,level,depth_limit):
  new_attrs = attrs.copy()
  new_data = data.copy()
  current_level = level # way of tracking depth
  # Using ID3 algorithm
  # Base case, when we have all of one label
  if 0 in list(count_labels(new_data,self.classes).values()):
    labels = count_labels(new_data,self.classes)
    max_label = max(labels,key=labels.get)
    # Return a single node tree with the correct label
    node = Node(max_label)
    node.level = current_level
    return node
  # If we are at our max depth, return a Node with the most common label
  if current_level == depth_limit:
    common_label_node = Node(str(get_common_label(data)))
    common_label_node.level = current_level
    return common_label_node
  # recursive case
  else:
    # iterate to next level and store if it's bigger than current max
    next_level = current_level + 1
    if next_level > self.tree_depth:
      self.tree_depth = next_level

    # Make root node
    root = Node()
    A,_information_gain = get_best_attr(new_data,new_attrs,self.classes,self.purity_measure) #
    # make A the root node
    root.attribute = A
    root.information_gain = _information_gain
    root.level = next_level
    # Split on the attribute
```

```
    [split_1,split_0] = split_on_attr(str(A),new_data)
    splits = [split_0,split_1] # Put 0 value first the 1 value second
    # Remove attribute from list
    new_attrs.remove(A)
    if A in new_attrs: print("ALERT")
    for i in range(2): # Possible values are a 0 and a 1
      if len(splits[i]) == 0:  # If the set is empty
        common_label_node = Node(str(get_common_label(data)))
        common_label_node.level = next_level
        if i == 0:
        # Add to left of tree
          root.left = common_label_node
        # Add to the right of tree
        if i == 1:
          root.right = common_label_node
      # Add branch for A taking value v
      if i == 0:
        # Add to left of tree
        root.left = self.train_tree(splits[i],new_attrs,next_level,depth_limit)
      if i == 1:
        # Add to right of tree
        root.right = self.train_tree(splits[i],new_attrs,next_level,depth_limit)
    return root

def get_prediction(self,instance):
  # Feed in data instance features and return label
  example = instance.copy()
  current_node = self.tree
  # traverse along the tree
  traversing = True
  while traversing:
    if current_node.label:
      # print("label",current_node.label)
      return current_node.label
    else:
      # print("splitting attr",current_node.attribute)
      splitting_Attr = str(current_node.attribute)
      if splitting_Attr in example:
        # Go right, b/c it has it
        # print("right")
        current_node = current_node.right
      else:
        # Go left, b/c it doesn't have it
        # print("left")
        current_node = current_node.left

def get_predict_accuracy(self,data):
  myData = data.data.copy()
  correct_labels = 0
  total_examples = len(myData)
  for i in myData.items():
    label = i[1]['label']
    features = i[1]['features']
    index = i[0]
    predicted_label = self.get_prediction(features)
    # print(index,label,predicted_label,features)
```

```
    # print(index,label,predicted_label,features)
    if label == predicted_label:
      correct_labels += 1
    # print(correct_labels)
  return correct_labels / total_examples

# returns predicitons in numpy array
def get_predictions(self,data):
  myData = data.data.copy()
  correct_labels = 0
  total_examples = len(myData)
  predictions = []
  for i in myData.items():
    label = i[1]['label']
    features = i[1]['features']
    index = i[0]
    predicted_label = int(self.get_prediction(features))
    # print(index,label,predicted_label,features)
    if predicted_label == -1:
      predicted_label = 0
    predictions.append(predicted_label)
  return np.array(predictions)


def get_predict_error(self,data):
  return 1 - self.get_predict_accuracy(data)

def get_depth(self):
  return self.tree_depth
```

```
# Testing data implementation
myData = Data(TRAINING_PATH)
has_attr, no_attr = split_on_attr('5',myData.data)
has_attr_labels = get_labels(has_attr)
print(has_attr_labels)
print(count_labels(has_attr,myData.classes))
print(count_labels(no_attr,myData.classes))
# print(count_labels(myData.data,myData.classes))
print(myData.data[0])
print(get_common_label(myData.data))
```

```
['+1', '+1', '-1', '-1', '+1', '+1', '+1', '-1', '+1', '+1', '+1', '+1', '+1', '+1', '+1'
{'+1': 7225, '-1': 2362}
{'+1': 1465, '-1': 6448}
{'label': '-1', 'features': ['4', '6', '11', '26', '78']}
-1
```

```
# Ok - looks like it works ok - lets build some trees ...

# Load test set

myData = Data(TRAINING_PATH)
```

```
myData = Data(TRAINING_PATH)
myTestData = Data(TESTING_PATH)

num_features = myData.num_features
attributes = [i for i in range(1,num_features)]
decision_tree = DecisionTreeClassifier(myData,attributes)
tree = decision_tree.build_tree(5)

# Decision tree results
print("training accuracy: ",decision_tree.get_predict_accuracy(myData))
print("training error: ",decision_tree.get_predict_error(myData))
print("test accuracy: ",decision_tree.get_predict_accuracy(myTestData))
print("test error: ", decision_tree.get_predict_error(myTestData))
print("root attribuite:",decision_tree.tree.attribute)
print("root information gain:", decision_tree.tree.information_gain)
print("max depth: ",decision_tree.get_depth())
decision_tree.get_predictions(myTestData)
```

```
training accuracy:  0.7881714285714285
training error:  0.21182857142857148
test accuracy:  0.7977777777777778
test error:  0.20222222222222222
root attribuite: 5
root information gain: 0.24616352406776065
max depth:  5
array([0, 1, 0, ..., 1, 0, 1])
```

```
# Ok, 5 seems decent
# Let's output

# Run on eval and return submission file in csv w labels columns: "example_id" "label"
evalData = Data(EVAL_PATH)
eval_ids = np.reshape(np.array(load_ids(EVAL_IDS),dtype=np.int32),(evalData.length,1))
predictions = decision_tree.get_predictions(evalData)
predictions = np.reshape(predictions,(evalData.length,1))
eval_out = np.hstack((eval_ids,predictions))
# print(eval_out)
eval_df = pd.DataFrame(data = eval_out,index = None,columns=['example_id','label'])
save_to_path = '/content/drive/My Drive/Colab Notebooks/Machine Learning 2020/old-bailey-decis
eval_df.to_csv(path_or_buf=save_to_path,index=False)
```

```
# training accuracy:  0.8021142857142857
# training error:  0.19788571428571433
# test accuracy:  0.7928888888888889
# test error:  0.20711111111111113
# root attribuite: 5
# root information gain: 0.24616352406776065
# max depth:  50

# training accuracy:  0.7906857142857143
# training error:  0.20931428571428567
# test accuracy:  0.7973333333333333
# test error:  0.20266666666666666
# root attribuite: 5
# root information gain: 0.24616352406776065
# max depth:  10
```

```
# max depth:  10
```