CS145 Howework 2

Important Note: HW2 is due on **11:59 PM PT, Oct 30 (Friday, Week 4)**. Please submit through GradeScope.

Print Out Your Name and UID

Name: Rui Deng, UID: 205123245

Before You Start

You need to first create HW2 conda environment by the given `cs145hw2.yml` file, which provides the name and necessary packages for this tasks. If you have `conda` properly installed, you may create, activate or deactivate by the following commands:

```
conda env create -f cs145hw2.yml
conda activate hw1
conda deactivate
```

OR

```
conda env create --name NAMEOFYOURCHOICE -f cs145hw2.yml
conda activate NAMEOFYOURCHOICE
conda deactivate
```

To view the list of your environments, use the following command:

```
conda env list
```

More useful information about managing environments can be found here (https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html).

You may also quickly review the usage of basic Python and Numpy package, if needed in coding for matrix operations.

In this notebook, you must not delete any code cells in this notebook. If you change any code outside the blocks (such as some important hyperparameters) that you are allowed to edit (between STRART/END YOUR CODE HERE), you need to highlight these changes. You may add some additional cells to help explain your results and observations.

In [13]:

```
import numpy as np
import pandas as pd
import seaborn as sns
import sys
import random as rd
import matplotlib.pyplot as plt
%load_ext autoreload
%autoreload 2
```

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

If you can successfully run the code above, there will be no problem for environment setting.

1. Decision trees

This workbook will walk you through a decision tree.

1.1 Attribute selection measures

For classification models, misclassification rate is usually used as the final performance measurement. However, for classification trees, when selecting which attribute to split, measurements people often use includes information gain, gain ratio, and Gini index. Let's investigate these different measurements through the following problem.

Note: below shows how to calculate the misclassification rate of a classification tree with N total data points, K classes of the value we want to predict, and M leaf nodes.

In a node m, $m = 1, \ldots, M$, let's denote the number of data points using N_m, and the number of data points in class k as N_{mk}, so the class prediction under majority vote is $j = argmax_k N_{mk}$. The misclassification rate of this node m is $R_m = 1 - \frac{N_{mj}}{N_m}$. The total misclassification rate of the tree will be $R = \frac{\sum_{m=1}^{M} R_m * N_m}{N}$

Questions

Note: this question is a pure "question answer" problem. You don't need to do any coding.

Suppose our dataset includes a total of 800 people with 400 males and 400 females, and our goal is to do gender classification. Consider two different possible attributes we can split on in a decision tree model. Split on the first attribute results in a node11 with 300 male and 100 female, and a node12 with 100 male and 300 female. Split on the second attribute results in in a node21 with 400 male and 200 female, and a node22 with 200 female only.

1. Which split do you prefer when the measurement is misclassifcation rate and why?
2. What is the entropy in each of these four node?
3. What is the information gain of each of the two splits?
4. Which split do you prefer if the measurement is information gain. Do you see why it is an uncertainty or impurity measurement?
5. What is the gain ratio (normalized information gain) of each of the two splits? Which split do you prefer under this measurement. Do you get the same conclusion as information gain?

Your answer here:

Note: you can use several code cells to help you compute the results and answer the questions. Again you don't need to do any coding.

Please type your answer here!

answer 1

By the definition of misclassification rate, we can see that it can be calculated as the number of misclassified samples divided by the total sample number. Thus, misclassification for the split on the first attribute is: (100 females are misclassified on node11, 100 males are misclassified on node 12)

$$\frac{(100+100)}{800} = 0.25$$

Misclassification for the split on the second attribute: (200 females are misclassified on node21)

$$\frac{200}{800} = 0.25$$

Thus, there is no different between two splits if we only consider the misclassification rate.

answer 2

We know that:

$$Entropy = Info(D) = -\sum_{i=1}^{m} p_i log_2(p_i)$$

So entropy for node11 is:

$$Info(node11) = -\frac{3}{4}log(\frac{3}{4}) - \frac{1}{4}log(\frac{1}{4}) = 0.8112781$$

Similarly,

$$Info(node12) = -\frac{3}{4}log(\frac{3}{4}) - \frac{1}{4}log(\frac{1}{4}) = 0.8112781$$

$$Info(node21) = -\frac{2}{3}log(\frac{2}{3}) - \frac{1}{3}log(\frac{1}{3}) = 0.9182958$$

$$Info(node22) = -1log(1) = 0$$

answer 3

We first calculate the entropy of dataset:

$$Info(root) = -\frac{1}{2}log(\frac{1}{2}) - \frac{1}{2}log(\frac{1}{2}) = 1$$

Then, the information gain for the first split is:

$$Gain(first) = 1 - (\frac{400}{800} * 0.8112781 + \frac{400}{800} * 0.8112781) = 0.187129$$

Similarly, the information gain for the second split is:

$$Gain(second) = 1 - (\frac{600}{800} * 0.0.9182958 + \frac{200}{800} * 0) = 0.31127813$$

answer 4

We should choose the split on the second attribute, since it gives us the most information gain.

The entropy is a measure of uncertainty, since when we have 400 females and 400 males in the root node, we have an entropy of 1, saying that we are totally unsure about the classification; conversely, when we have only 200 females in the node22, then the entropy is 0, meaning that there is no uncertainty or impurity in this node, and we are confident that this node should be classified to the female class.

answer 5

By the formula, the split info of the first split is:

$$SplitInfo(first) = -\frac{1}{2}log(\frac{1}{2}) - \frac{1}{2}log(\frac{1}{2}) = 1$$

And the gain ratio of the split split is therefore:

$$GainRatio(first) = \frac{Gain(first)}{SplitInfo(first)} = 0.18729$$

Similarly:

$$SplitInfo(second) = -\frac{3}{4}log(\frac{3}{4}) - \frac{1}{4}log(\frac{1}{4}) = 0.8112781$$

$$GainRatio(second) = \frac{Gain(second)}{SplitInfo(second)} = 0.3836885$$

Still, we will choose the second attribute to split, since it gives us the most gain ratio, and this result is the same as information gain.

We know that information gain is biased towards attribute with multiple values, but in this case, we only have 2 values for each attribute, so the gain ratio does not give us a different final choice of attribute.

1.2 Coding decision trees

In this section, we are going to use the decision tree model to predict the the animal `type` class of the `zoo` dataset. The dataset has been preprocessed and splited into `decision-tree-train.csv` and `decision-tree-test.csv` for you.

In [14]:

```
from hw2code.decision_tree import DecisionTree
mytree = DecisionTree()
mytree.load_data('./data/decision-tree-train.csv','./data/decision-tree-test.csv')
# As a sanity check, we print out the size of the training data (80, 17) and testing
print('Training data shape: ', mytree.train_data.shape)
print('Testing data shape:', mytree.test_data.shape)
```

```
Training data shape:  (80, 17)
Testing data shape: (21, 17)
```

1.2.1 Infomation gain

Complete the `make_tree` and `compute_info_gain` function in `decision_tree.py` .

Train you model using `info_gain` measure to classify `type` and print the test accuracy.

In [25]:

```
mytree = DecisionTree()
mytree.load_data('./data/decision-tree-train.csv','./data/decision-tree-test.csv')
test_acc= 0
#=========================#
# STRART YOUR CODE HERE  #
#=========================#
mytree.train('type', 'info_gain')
test_acc = mytree.test('type')
#=========================#
#   END YOUR CODE HERE   #
#=========================#
print('Test accuracy is: ', test_acc)
```

```
best_feature is:  legs
best_feature is:  fins
best_feature is:  toothed
best_feature is:  eggs
best_feature is:  hair
best_feature is:  hair
best_feature is:  toothed
best_feature is:  aquatic
Test accuracy is:  0.8571428571428571
```

1.2.2 Gain ratio

Complete the `compute_gain_ratio` function in `decision_tree.py` .

Train you model using `gain_ratio` measure to classify `type` and print the test accuracy.

In [27]:

```
mytree = DecisionTree()
mytree.load_data('./data/decision-tree-train.csv','./data/decision-tree-test.csv')
test_acc = 0
#=========================#
# STRART YOUR CODE HERE  #
#=========================#
mytree.train('type', 'gain_ratio')
test_acc = mytree.test('type')
#=========================#
#   END YOUR CODE HERE   #
#=========================#
print('Test accuracy is: ', test_acc)
```

```
best_feature is:  feathers
best_feature is:  backbone
best_feature is:  airborne
best_feature is:  predator
best_feature is:  milk
best_feature is:  fins
best_feature is:  legs
Test accuracy is:  0.8095238095238095
```

Question

Which measure do you like the most and why?

Your answer here:

We know that information gain is biased towards attributes with a large number of values, whereas the gain ratio tends to prefer unbalanced splits in which one partition is much smaller than others. Then, we need to examine our training set.

In [37]:

```
mytree = DecisionTree()
mytree.load_data('./data/decision-tree-train.csv','./data/decision-tree-test.csv')
features = mytree.train_data.columns.values
for feature in features:
    vals, counts = np.unique(mytree.train_data[feature], return_counts=True)
    print("%s: " % feature, counts)
```

```
hair:  [43 37]
feathers:  [64 16]
eggs:  [37 43]
milk:  [44 36]
airborne:  [62 18]
aquatic:  [50 30]
predator:  [31 49]
toothed:  [30 50]
backbone:  [13 67]
breathes:  [17 63]
venomous:  [74  6]
fins:  [66 14]
legs:  [17 20 33  8  2]
tail:  [20 60]
domistic:  [67 13]
catsize:  [43 37]
type:  [36 16  2 10  3  6  7]
```

We can see that there are only two attributes with multiple number of values other than two, but there are lots of attributes, such as venomous, tail, legs, and demoistic, that has unbalanced splits. Therefore, in this case, using information gain gives us a better accuracy.

2. SVM

This workbook will walk you through a SVM.

2.1 Support vectors and decision boundary

Note: for this question you can work entirely in the Jupyter Notebook, no need to edit any .py files.

Consider classifying the following 20 data points in the 2-d plane with class label y

In [38]:

```
ds = pd.read_csv('data/svm-2d-data.csv')
ds.head()
# This command above will print out the first five data points
# in the dataset with column names as "x1", "x2" and "y"
# You may use command "ds" to show the entire dataset, which contains 20 data points
```

Out[38]:

	x1	x2	y
0	0.52	-1.00	1
1	0.91	0.32	1
2	-1.48	1.23	1
3	0.01	1.44	1
4	-0.46	-0.37	1

Suppose by solving the dual form of the quadratic programming of svm, we can derive the α_i's for each data point as follows: Among $j = 0, 1, \cdots, 19$ (note that the index starts from 0), α_1 = 0.5084, α_5 = 0.4625, α_{17} = 0.9709, and α_j = 0 for all other j.

Questions

1. Which vectors in the training points are support vectors?
2. What is the normal vector of the hyperplane w?
3. What is the bias b?
4. With the parameters w and b, we can now use our SVM to do predictions. What is predicted label of $x_{new} = (2, -0.5)$? Write out your $f(x_{new})$.
5. A plot of the data points has been generated for you. Please change the `support_vec` variable such that only the support vectors are indicated by red circles. Please also fill in the code to draw the decision boundary. Does your prediction of part 4 seems right visually on the plot?

Your answer here

Note: you can use several code cells to help you compute the results and answer the questions. Again you don't need to edit any .py files.

Please type your answer here!

answer 1

The support vectors are data points with non-zero α values. In this case, data point 1, 5, 17 are the only three support vectors.

answer 2

The solution of SVM gives us the normal vector as:

$$w = \sum a_i y_i x_i$$

Thus, we have:

In [46]:

```
w = 0.5084 * ds.iloc[1][2] * ds.iloc[1][:2].to_numpy() + 0.4625 * ds.iloc[5][2] * ds
0.9709 * ds.iloc[17][2] * ds.iloc[17][:2].to_numpy()
print(w)
```

```
[-1.338076 -0.388998]
```

So the normal vector is $w = [-1.338076, -0.388998]$.

answer 3

The solution of SVM gives us the bias term as:

$$b = y_k - w^T x_k$$

for any x_k with non-zero α_k term. Thus, we have:

In [71]:

```
b = ds.iloc[1][2] - w.dot(ds.iloc[1][:2].to_numpy())
b += ds.iloc[5][2] - w.dot(ds.iloc[5][:2].to_numpy())
b += ds.iloc[17][2] - w.dot(ds.iloc[17][:2].to_numpy())
b /= 3
print(b)
```

```
2.342136106666666
```

Here, we take the average of three possible bias terms, and the bias is therefore 2.3421361.

answer 4

We know that

$$f(x_{new}) = w^T x_k + b$$

So we have:

In [60]:

```
x = np.array([2, -0.5])
f = w.dot(x) + b
print(f)
```

```
-0.1395168933333335
```

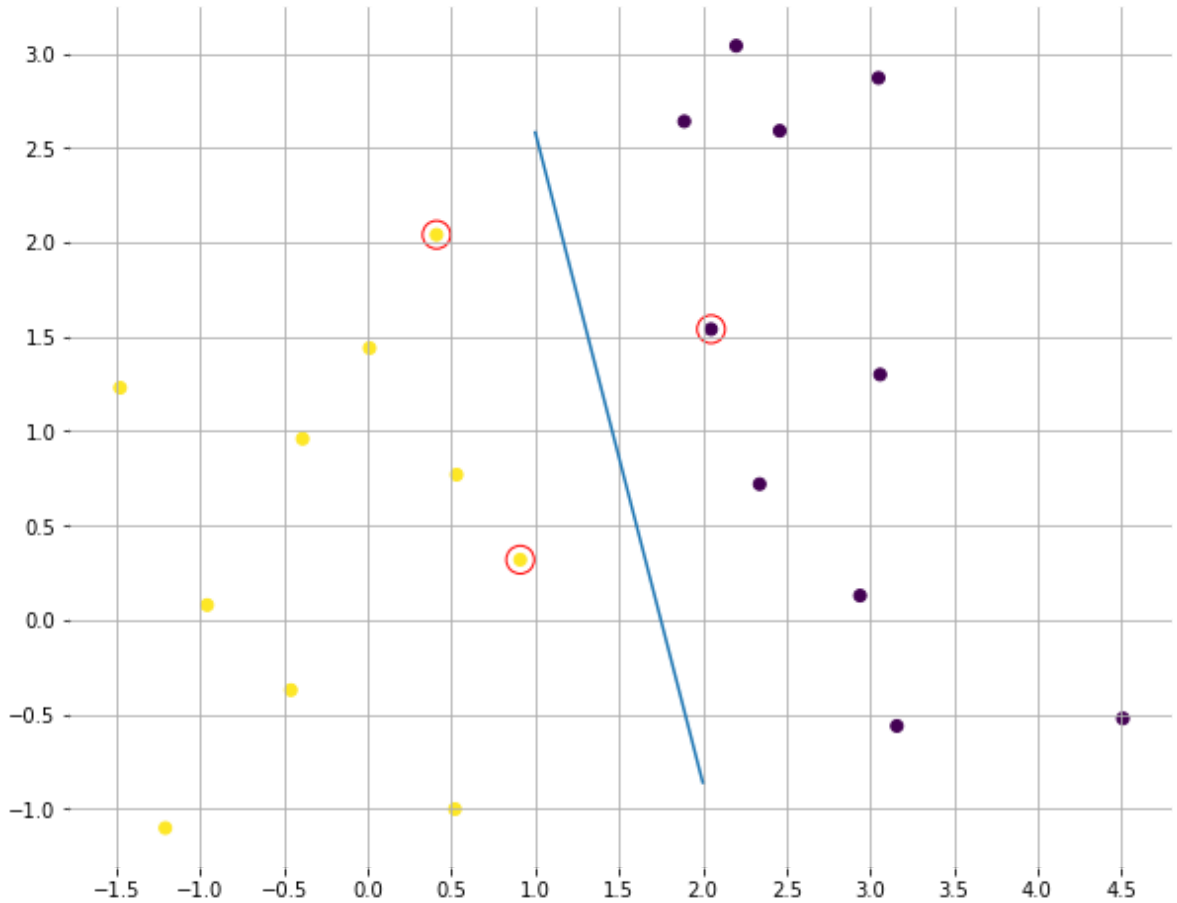
Here, we have $f(x_{new}) = -0.139516893$ is less than 0, so we predict that it belongs to y = -1.

In [73]:

```
# answer 5
x1_range = np.arange(-2, 5, 0.5)
x2_range = np.arange(-2, 4., 0.5)

fig, ax = plt.subplots(figsize=(10, 8))
ax = fig.gca()
ax.set_xticks(x1_range)
ax.set_yticks(x2_range)
ax.grid()
ax.scatter(ds['x1'], ds['x2'], c=ds['y'])

support_vec = ds
#========================#
# STRART YOUR CODE HERE  #
#========================#
support_vec = ds.iloc[[1, 5, 17], :]
xs = np.linspace(1, 2, 100)
ys = - w[0] / w[1] * xs - b / w[1]
ax.plot(xs, ys)
#========================#
#   END YOUR CODE HERE   #
#========================#
ax.scatter(support_vec['x1'], support_vec['x2'], marker='o', facecolor='none', s=200
sns.despine(ax=ax, left=True, bottom=True, offset=0)
plt.show()
```



2.2 Coding SVM

In this section, we are going to use SVM for classifying the `y` value of 4-dimensional data points. The dataset has been preprocessed and splited into `svm-train.csv` and `svm-test.csv` for you.

For this question we are going to use the `cvxopt` package to help us solve the optimization problem of SVM. You will see it in the .py files, but you don't need to any coding with it. For this question, you only need to implement the right kernel function, and your kernel matrix `K` in `svm.py` line 135 will be pluged in the cvxopt optimization problem solver.

For more information about `cvxopt` please refer to http://cvxopt.org/ (http://cvxopt.org/)

In [76]:

```
from hw2code.svm import SVM
svm = SVM()
svm.load_data('./data/svm-train.csv', './data/svm-test.csv')
# As a sanity check, we print out the size of the training data (1098, 4) and (1098,
print('Training data shape: ', svm.train_x.shape, svm.train_y.shape)
print('Testing data shape:', svm.test_x.shape, svm.test_y.shape)
```

```
Training data shape:  (1098, 4) (1098,)
Testing data shape: (274, 4) (274,)
```

2.2.1 Linear kernel

Complete the `SVM.predict` and `linear_kernel` function in `svm.py` . Train a hard margin SVM and a soft margin SVM with linear kernel. Print the test accuracy for both cases.

In [151]:

```
svm_hard = SVM()
svm_hard.load_data('./data/svm-train.csv', './data/svm-test.csv')
hard_test_acc = 0
#=========================#
# STRART YOUR CODE HERE  #
#=========================#
svm_hard.train()
hard_test_acc = svm_hard.test()
#=========================#
#   END YOUR CODE HERE   #
#=========================#

svm_soft = SVM()
svm_soft.load_data('./data/svm-train.csv', './data/svm-test.csv')
soft_test_acc = 0
#=========================#
# STRART YOUR CODE HERE  #
#=========================#
svm_soft.train(C=1)
soft_test_acc = svm_soft.test()
#=========================#
#   END YOUR CODE HERE   #
#=========================#
print('Hard margin test accuracy is: ', hard_test_acc)
print('Soft margin test accuracy is: ', soft_test_acc)
```

```
1098 support vectors out of 1098 points
38 support vectors out of 1098 points
Hard margin test accuracy is:  0.5547445255474452
Soft margin test accuracy is:  0.9890510948905109
```

Questions

Are these two results similar? Why or why not?

Your Answer

The results are not similar. For hard margin SVM, it necessitates that all the data points are of certain margin to the decision boundary. However, in higher dimensions, our data cannot be separated neatly by a hyperplane. In such case, hard margin SVM uses all the points as support vectors, so it requires all the points are outside of the margin and thus it leaves little margin in between. Consequently, the small margin does not provide us with a well-separated decision boundary, and the test accuracy for hard SVM is far below the soft SVM case.

On the contrary, the soft SVM allows some extent of misclassifications as well as allowing some points inside the margin. As a result, we have only 38 support vectors, leading to a larger margin and less overfitting than the hard SVM case. The larger margin gives rise to a more robust decision boundary and leads to an almost perfect test accuracy.

2.2.2 Polynomial kernel

Complete the `polynomial_kernel` function in `svm.py` . Train a soft margin SVM with degree 3 polynomial kernel and parameter `C = 100` for the regularization term. Print the test accuracy.

In [153]:

```
svm = SVM()
svm.load_data('./data/svm-train.csv', './data/svm-test.csv')
test_acc = 0
#========================#
# STRART YOUR CODE HERE  #
#========================#
svm.train('polynomial_kernel', C=100)
test_acc = svm.test()
#========================#
#   END YOUR CODE HERE   #
#========================#
print('Test accuracy is: ', test_acc)
```

```
19 support vectors out of 1098 points
Test accuracy is:  0.927007299270073
```

Questions

Is the result better than linear kernel? Why or why not?

Your Answer Test accuracy with polynomial degree = 2: 1.0

Test accuracy with polynomial degree = 3: 0.927007299270073

Test accuracy with polynomial degree = 5: 0.9781021897810219

We can see that if we use the polynomial of degree 2, then we have the perfect accuracy. That is to say, using polynomial of degree 3 may cause an overfit to the training data that reduces the accuracy on the testing set.

For polynomials of degree 3 and above, none of them outperform the soft SVM with linear kernel. This is possibly because that the high-dimensional polynomial transformation of the data points still cannot give us a boundary that separate them clearly. Maybe it is worse than the linear case since the distribution of mapped data are more complex and thus more difficult to separate by a hyperplane.

Please write down your answers and/or observations here

2.2.3 Gaussian kernel

Complete the `gaussian_kernel` function using the `gaussian_kernel_point` in `svm.py` . Train a soft margin SVM with Gaussian kernel and parameter `C = 100` for the regularization term. Print the test accuracy.

In [118]:

```
svm = SVM()
svm.load_data('./data/svm-train.csv', './data/svm-test.csv')
test_acc = 0
#========================#
# STRART YOUR CODE HERE  #
#========================#
svm.train('gaussian_kernel', C=100)
test_acc = svm.test()
#========================#
#   END YOUR CODE HERE   #
#========================#
print('Test accuracy is: ', test_acc)
```

```
35 support vectors out of 1098 points
Test accuracy is:  1.0
```

Questions

1. Is the result better than linear kernel and polynomial kernel? Why or why not?
2. Which one of these four models do you like the most and why?
3. (Bonus question, optional) Can you come up with a vectorized implementation of `gaussian_kernel` without calling `gaussian_kernel_point` ? Fill that in svm.py.

Your Answer

Please write down your answers and/or observations here

answer 1

The result is indeed better than both the linear kernel and the polynomial kernel. As introduced in lecture, the Gaussian kernel gives us a mapping which is infinite in the dimension of x. That is to say, we can treat the Guassian kernel as a polynomial kernel with infinite dimensions. As a result, the Gaussian kernel can be generalized to different kinds of distributions, and it gives us the bell-shaped surface centered at each support vector.

answer 2

I prefer the soft SVM with Gaussian kernel, since it can extend x to infinite dimensions and normalize it with its distribution. That is to say, Gaussian kernel can be well adopted to different kinds of distributions. On the contrary, linear kernel are only suitable for linear-separable data, and the polynomial kernel has limited power to expand the feature space of x.

answer 3

See the vectorized implementation in svm.py.

In [149]:

```
svm = SVM()
svm.load_data('./data/svm-train.csv', './data/svm-test.csv')
test_acc = 0
#=========================#
# STRART YOUR CODE HERE  #
#=========================#
svm.train('gaussian_kernel', C=100)
test_acc = svm.test()
#=========================#
#   END YOUR CODE HERE   #
#=========================#
print('Test accuracy is: ', test_acc)
```

```
35 support vectors out of 1098 points
Test accuracy is:  1.0
```

End of Homework 2 :)

After you've finished the homework, please print out the entire `ipynb` notebook and two `py` files into one PDF file. Make sure you include the output of code cells and answers for questions. Prepare submit it to GradeScope. Also this time remember assign the pages to the questions on GradeScope

In [ ]:

```

```

```
import pandas as pd
import numpy as np
from pprint import pprint
import sys

# Reads the data from CSV files, each attribute column can be obtained via its
 name, e.g., y = data['y']
def getDataframe(filePath):
    data = pd.read_csv(filePath)
    return data

# predicted_y and y are the predicted and actual y values respectively as numpy
 arrays
# function prints the accuracy
def compute_accuracy(predicted_y, y):
    acc = 100.0
    acc = np.sum(predicted_y == y)/predicted_y.shape[0]
    return acc

#Compute entropy according to y distribution
def compute_entropy(y):
    entropy = 0.0
    elements,counts = np.unique(y, return_counts = True)
    n = y.shape[0]

    for i in range(len(elements)):
        prob = counts[i]/n
        if prob!= 0:
            entropy -= prob * np.log2(prob)
    return entropy

#att_name: attribute name; y_name: the target attribute name for classification
def compute_info_gain(data, att_name, y_name):
    info_gain = 0.0

    #Calculate the values and the corresponding counts for the select attribute
    vals, counts = np.unique(data[att_name], return_counts=True)
    total_counts = np.sum(counts)

    #Calculate the conditional entropy
    #=======================#
    # STRART YOUR CODE HERE  #
    #=======================#
    info_gain = compute_entropy(data[y_name])
    for v, count in zip(vals, counts):
        info_gain -= count/total_counts * compute_entropy(data[data[att_name]
         == v][y_name])
    #=======================#
    #   END YOUR CODE HERE   #
    #=======================#
```

```
    return info_gain


def comput_gain_ratio(data, att_name, y_name):
    gain_ratio = 0.0
    #Calculate the values and the corresponding counts for the select attribute
    vals, counts = np.unique(data[att_name], return_counts=True)
    total_counts = np.sum(counts)

    #Calculate the information for the selected attribute
    att_info = 0.0
    #========================#
    # STRART YOUR CODE HERE  #
    #========================#
    att_info = compute_entropy(data[att_name])
    #========================#
    #   END YOUR CODE HERE   #
    #========================#
    gain_ratio = 0.0 if np.abs(att_info) < 1e-9 else min(1,
     compute_info_gain(data, att_name, y_name) / att_info)
    return gain_ratio

# Class of the decision tree model based on the ID3 algorithm
class DecisionTree(object):
    def __init__(self):
        self.train_data = pd.DataFrame()
        self.test_data = pd.DataFrame()

    def load_data(self, train_file, test_file):
        self.train_data = getDataframe(train_file)
        self.test_data = getDataframe(test_file)

    def train(self, y_name, measure, parent_node_class= None):
        self.y_name = y_name
        self.measure = measure
        self.tree = self.make_tree(self.train_data, parent_node_class)

    def make_tree(self, train_data, parent_node_class = None):
        data = train_data
        features = data.drop(self.y_name, axis = 1).columns.values
        measure =  self.measure
        #Stopping condition 1: If all target_values have the same value, return
         this value
        if len(np.unique(data[self.y_name])) <= 1:
            leaf_value = -1
            #========================#
            # STRART YOUR CODE HERE  #
            #========================#
            leaf_value = np.unique(data[self.y_name])
            #========================#
            #   END YOUR CODE HERE   #
```

```
    #========================#
    return leaf_value

#Stopping condition 2: If the dataset is empty, return the
 parent_node_class
elif len(data)== 0:
    return parent_node_class

#Stopping condition 3: If the feature space is empty, return the
 majority class
elif len(features) == 0:
    return np.unique(data[self.y_name])
     [np.argmax(np.unique(data[y_name],return_counts=True)[1])]

# Not a leaf node, create an internal node
else:
    #Set the default value for this node --> The mode target feature
     value of the current node
    parent_node_class = np.unique(data[self.y_name])
     [np.argmax(np.unique(data[self.y_name],return_counts=True)[1])]

    #Select the feature which best splits the dataset
    if measure == 'info_gain':
        item_values = [compute_info_gain(data, feature, self.y_name)
         for feature in features] #Return the information gain values
         for the features in the dataset
    elif measure == 'gain_ratio':
        item_values = [comput_gain_ratio(data, feature, self.y_name)
         for feature in features] #Return the gain_ratio for the
         features in the dataset
    else:
        raise ValueError("kernel not recognized")

    best_feature_index = np.argmax(item_values)
    best_feature = features[best_feature_index]
    print('best_feature is: ', best_feature)

    #Create the tree structure. The root gets the name of the feature
     (best_feature)
    tree = {best_feature:{}}


#Grow a branch under the root node for each possible value of the root
 node feature

for value in np.unique(data[best_feature]):
    #Split the dataset along the value of the feature with the largest
     information gain and therwith create sub_datasets
    sub_data = data.where(data[best_feature] == value).dropna()

    #Remove the selected feature from the feature space
```

```
            sub_data = sub_data.drop(best_feature, axis = 1)

            #Call the ID3 algorithm for each of those sub_datasets with the new
             parameters --> Here the recursion comes in!
            subtree = self.make_tree(sub_data, parent_node_class)

            #Add the sub tree, grown from the sub_dataset to the tree under the
             root node
            tree[best_feature][value] = subtree

        return tree


    def test(self, y_name):
        accuracy = self.classify(self.test_data, y_name)
        return accuracy

    def classify(self, test_data, y_name):
        #Create new query instances by simply removing the target feature
         column from the test dataset and
        #convert it to a dictionary
        test_x = test_data.drop(y_name, axis=1)
        test_y = test_data[y_name]

        n = test_data.shape[0]
        predicted_y = np.zeros(n)

        #Calculate the prediction accuracy
        for i in range(n):
            predicted_y[i] = DecisionTree.predict(self.tree, test_x.iloc[i])

        output = np.zeros((n,2))
        output[:,0] = test_y
        output[:,1] = predicted_y
        accuracy = compute_accuracy(predicted_y, test_y.values)
        return accuracy

    def predict(tree, query):
        # find the root attribute
        default = -1
        for root_name in list(tree.keys()):
            try:
                subtree = tree[root_name][query[root_name]]
            except:
                return default ## root_name does not appear in query attribute
                 list (it is an error!)

            ##if subtree is still a dictionary, recursively test next attribute
            if isinstance(subtree, dict):
                return DecisionTree.predict(subtree, query)
            else:
```

```
leaf = subtree
return leaf
```

```
import numpy as np
from numpy import linalg
import cvxopt
import cvxopt.solvers
import sys
import pandas as pd
cvxopt.solvers.options['show_progress'] = False

# Reads the data from CSV files, converts it into Dataframe and returns x and y
 dataframes
def getDataframe(filePath):
    dataframe = pd.read_csv(filePath)
    y = dataframe['y']
    x = dataframe.drop('y', axis=1)
    y = y*2 -1.0
    return x.to_numpy(), y.to_numpy()

def compute_accuracy(predicted_y, y):
    acc = 100.0
    acc = np.sum(predicted_y == y)/predicted_y.shape[0]
    return acc

def gaussian_kernel_point(x, y, sigma=5.0):
    return np.exp(-linalg.norm(x-y)**2 / (2 * (sigma ** 2)))

def linear_kernel(X, Y=None):
    Y = X if Y is None else Y
    m = X.shape[0]
    n = Y.shape[0]
    assert X.shape[1] == Y.shape[1]
    kernel_matrix = np.zeros((m, n))
    #=========================#
    # STRART YOUR CODE HERE  #
    #=========================#

    kernel_matrix = X.dot(Y.T)

    #=========================#
    #   END YOUR CODE HERE   #
    #=========================#
    return kernel_matrix

def polynomial_kernel(X, Y=None, degree=3):
    Y = X if Y is None else Y
    m = X.shape[0]
    n = Y.shape[0]
    assert X.shape[1] == Y.shape[1]
    kernel_matrix = np.zeros((m, n))
    #=========================#
    # STRART YOUR CODE HERE  #
    #=========================#
```

```
    kernel_matrix = (X.dot(Y.T) + 1) ** degree

    #=======================#
    #   END YOUR CODE HERE   #
    #=======================#
    return kernel_matrix

# def gaussian_kernel(X, Y=None, sigma=5.0):
#     Y = X if Y is None else Y
#     m = X.shape[0]
#     n = Y.shape[0]
#     assert X.shape[1] == Y.shape[1]
#     kernel_matrix = np.zeros((m, n))
#     #=======================#
#     # STRART YOUR CODE HERE  #
#     #=======================#
#     for i in range(m):
#         for j in range(n):
#             kernel_matrix[i][j] = gaussian_kernel_point(X[i], Y[j])
#     #=======================#
#     #   END YOUR CODE HERE   #
#     #=======================#
#     return kernel_matrix


# Bonus question: vectorized implementation of Gaussian kernel
# If you decide to do the bonus question, comment the gaussian_kernel function
 above,
# then implement and uncomment this one.
def gaussian_kernel(X, Y=None, sigma=5.0):
    Y = X if Y is None else Y
    assert X.shape[1] == Y.shape[1]
    x_norm = np.expand_dims(X.dot(X.T).diagonal(), axis=-1)
    y_norm = np.expand_dims(Y.dot(Y.T).diagonal(), axis=-1)

    x_norm_mat = x_norm.dot(np.ones((Y.shape[0], 1), dtype=np.float64).T)
    y_norm_mat = np.ones((X.shape[0], 1), dtype=np.float64).dot(y_norm.T)
    k = x_norm_mat + y_norm_mat - 2 * X.dot(Y.T)
    k /= - 2 * sigma ** 2
    return np.exp(k)


class SVM(object):
    def __init__(self):
        self.train_x = pd.DataFrame()
        self.train_y = pd.DataFrame()
        self.test_x = pd.DataFrame()
        self.test_y = pd.DataFrame()
        self.kernel_name = None
        self.kernel = None
```

```
def load_data(self, train_file, test_file):
    self.train_x, self.train_y = getDataframe(train_file)
    self.test_x, self.test_y = getDataframe(test_file)


def train(self, kernel_name='linear_kernel', C=None):
    self.kernel_name = kernel_name
    if(kernel_name == 'linear_kernel'):
        self.kernel = linear_kernel
    elif(kernel_name == 'polynomial_kernel'):
        self.kernel = polynomial_kernel
    elif(kernel_name == 'gaussian_kernel'):
        self.kernel = gaussian_kernel
    else:
        raise ValueError("kernel not recognized")

    self.C = C
    if self.C is not None:
        self.C = float(self.C)

    self.fit(self.train_x, self.train_y)

# predict labels for test dataset
def predict(self, X):
    if self.w is not None: ## linear case
        n = X.shape[0]
        predicted_y = np.zeros(n)
        #========================#
        # STRART YOUR CODE HERE  #
        #========================#

        predicted_y = X.dot(self.w.T) + self.b

        #========================#
        #   END YOUR CODE HERE   #
        #========================#
        return predicted_y

    else: ## non-linear case
        n = X.shape[0]
        predicted_y = np.zeros(n)
        #========================#
        # STRART YOUR CODE HERE  #
        #========================#
        for i in range(n):
            prod = np.expand_dims(self.a * self.sv_y, axis=-1)
            x = np.expand_dims(X[i], axis = -1)
            predicted_y[i] = np.sum(prod * self.kernel(self.sv, x.T)) +
             self.b
        #========================#
```

```
        #   END YOUR CODE HERE   #
        #========================#
        return predicted_y

#================================================#
# Please DON'T change any code below this line!  #
#================================================#
def fit(self, X, y):
    n_samples, n_features = X.shape
    # Kernel matrix
    K = self.kernel(X)

    # dealing with dual form quadratic optimization
    P = cvxopt.matrix(np.outer(y,y) * K)
    q = cvxopt.matrix(np.ones(n_samples) * -1)
    A = cvxopt.matrix(y, (1,n_samples),'d')
    b = cvxopt.matrix(0.0)

    if self.C is None:
        G = cvxopt.matrix(np.diag(np.ones(n_samples) * -1))
        h = cvxopt.matrix(np.zeros(n_samples))
    else:
        tmp1 = np.diag(np.ones(n_samples) * -1)
        tmp2 = np.identity(n_samples)
        G = cvxopt.matrix(np.vstack((tmp1, tmp2)))
        tmp1 = np.zeros(n_samples)
        tmp2 = np.ones(n_samples) * self.C
        h = cvxopt.matrix(np.hstack((tmp1, tmp2)))

    # solve QP problem
    solution = cvxopt.solvers.qp(P, q, G, h, A, b)
    # Lagrange multipliers
    a = np.ravel(solution['x'])

    # Support vectors have non zero lagrange multipliers
    sv = a > 1e-5
    ind = np.arange(len(a))[sv]
    self.a = a[sv]
    self.sv = X[sv]
    self.sv_y = y[sv]

    print("%d support vectors out of %d points" % (len(self.a), n_samples))

    # Intercept via average calculating b over support vectors
    self.b = 0
    for n in range(len(self.a)):
        self.b += self.sv_y[n]
        self.b -= np.sum(self.a * self.sv_y * K[ind[n],sv])
    self.b /= len(self.a)

    # Weight vector
```

```
    if self.kernel_name == 'linear_kernel':
        self.w = np.zeros(n_features)
        for n in range(len(self.a)):
            self.w += self.a[n] * self.sv_y[n] * self.sv[n]
    else:
        self.w = None


def test(self):
    accuracy = self.classify(self.test_x, self.test_y)
    return accuracy

def classify(self, X, y):
    predicted_y = np.sign(self.predict(X))
    accuracy = compute_accuracy(predicted_y, y)
    return accuracy
```