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: Kevin Li, UID: 405200619

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 [12]:
```
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!

1

The first split will classify all in node11 as male and node 12 as female yielding a misclassification rate of 1/4 The second split will classify all in node 21 as male and all in node22 as female, yielding a misclassification rate of 1/4 as well

Neither is superior

2

	A	B	C	D
1		Male	Female	Entropy
2	node11	300	100	0.81
3	node12	100	300	0.81
4	node21	400	200	0.92
5	node22	0	200	0

3

Original entropy: 1

Split 1:

1 - (400/800 *Entropy(node11)* + *400/800* \ Entropy(node12))

1 - 0.81

0.19

Split 2:

1 - (600/800 * Entropy(node21) + 200/800 * Entropy(node22))

1 - 0.69

0.31

4

The second split. This demonstrates how information gain is an inpurity measurement since even with the same misclassification rate, the second is preferred due to it making a completely "pure" node (i.e. node22 is ONLY female)

5

Split 1:

InfoGain = 0.19

SplitInfo = -(400/800 * $\log_2$(400/800) + 400/800 * $\log_2$(400/800))

GainRatio = 0.19

Split 2:

InfoGain = 0.31

SplitInfo = -(600/800 * $\log_2$(600/800) + 200/800 * $\log_2$(200/800))

GainRatio = 0.38

Either works, but using gain ratio might not be neccessary. This is because it tries to adjust for creating many branches/nodes due to a highly multivalued attribute. In our case, we don't have many nodes (only two nodes for either split). Gain ratio gives the same conclusion as information gain - this is becuase SplitInfo likes making fewer nodes and therefore likes things like large single nodes.

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 [13]:
```
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 data (21, 17)
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 [14]:
```
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 [15]:
```
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:

I like information gain since it got a higher test 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 [16]:
```
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[16]:

	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.

1

Points 1, 5, and 17 since they have non-zero alpha values

2

The normal for the hyperplane is [-1.34, -0.39]

In [17]:
```
svs = ds.iloc[[1, 5, 17]]
vector = svs.drop(columns=['y']).multiply(svs['y'], axis='rows').multiply([.50
84, .4625, .9709], axis='rows').sum()
vector
```

Out[17]:
```
x1   -1.338076
x2   -0.388998
dtype: float64
```

3

With these support vectors, we know the distance to the hyperplane from each point is equivalent. Thus, we can turn those into a system of equations and solve it.

Distance from point to line where line is defined by a, b, β and point is defined by x, y is the following:

$$d = \frac{|ax+by+\beta|}{\sqrt{(a^2+b^2)}}$$

We know the a and b values since they're the normal vector, and by using one point with class=1 and one point with class=-1, we get the following:

$$d = \frac{|-1.338076 \cdot .91 - 0.388998 \cdot .32 + \beta|}{\sqrt{(1.338076^2+0.388998^2)}}$$

$$d = \frac{|-1.338076 \cdot 2.05 - 0.388998 \cdot 1.54 + \beta|}{\sqrt{(1.338076^2+0.388998^2)}}$$

We set them equal and solve

$$0.717631|\beta - 1.34213| = 0.717631|\beta - 3.34211|$$

Thus, we get the bias term (given that the normal is [-1.34, -0.39])

$$\beta = 2.34212$$

In [18]:
```
for row in svs.iterrows():
    print('Point {}: ({}, {}), class={}'.format(row[0], *row[1]))
```

```
Point 1: (0.91, 0.32), class=1.0
Point 5: (0.41, 2.04), class=1.0
Point 17: (2.05, 1.54), class=-1.0
```

4

$f([2, -0.5]) = [2, -0.5, 1]^T[-1.34, -0.389, 2.34]$

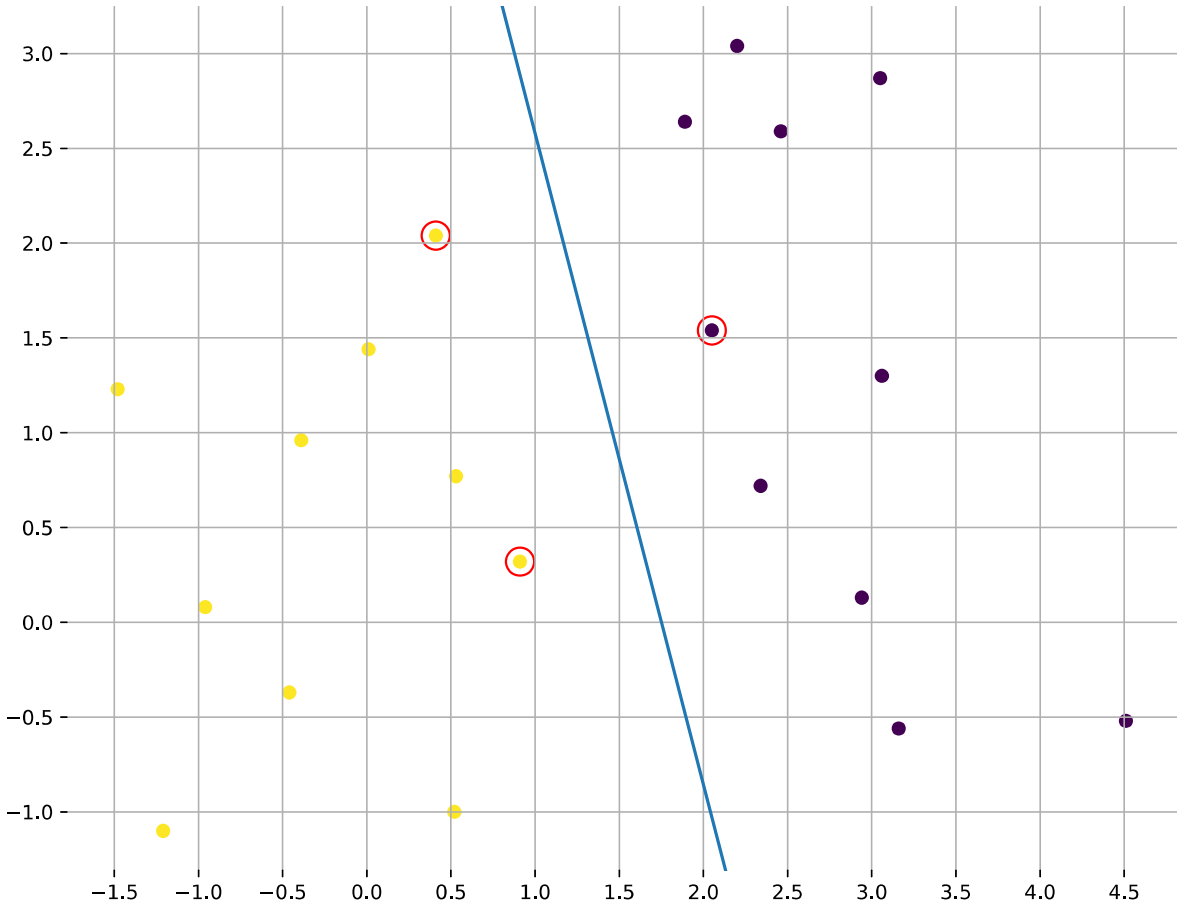
$f([2, -0.5]) = -0.1455$

The SVM predicts class=**-1**

In [19]:
```
# 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.iloc[[1, 5, 17]]
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
pt = (1.5, 0.861202)
ax.axline(pt, slope=-3.4398)
#=======================#
#   END YOUR CODE HERE   #
#=======================#
ax.scatter(support_vec['x1'], support_vec['x2'], marker='o', facecolor='none',
s=200, color='red')
sns.despine(ax=ax, left=True, bottom=True, offset=0)
plt.show()
```



5

The prediction looks right, and just barely as the numbers implied.

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 [20]:
```
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,) and testing data (274, 4) and (274,)
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 [21]:
```
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 very different. This is because the hard SVM couldn't give any slack and was unable to split the dataset (i.e. the dataset is not linearly separable); it seems it ended up using every single point as a support vector. It likely failed to converge on the optimization problem successfully and correctly.

Because of this, it failed to intelligently pick a bounding hyperplane that was relevant to the data i.e. it did not find a hyperplane that intelligently split the dataset.

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 [43]:
```
svm = SVM()
svm.load_data('./data/svm-train.csv', './data/svm-test.csv')
test_acc = 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
svm.train(kernel_name='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

It ended up doing slightly worse - this is probably because it overfit to the data as a polynomial kernel of is very similar to a linear kernel except that the higher the degree, the more closely it can fit the training data.

Since it was able to fit the training data better, it obviously chose to do that but ended it ended up slightly overfitting to the training data.

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 [46]:
```
svm = SVM()
svm.load_data('./data/svm-train.csv', './data/svm-test.csv')
test_acc = 0
#=======================#
# STRART YOUR CODE HERE  #
#=======================#
svm.train(kernel_name='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

1

The gaussian kernel ended up being the most successful kernel - gaussian RBF kernels are far more powerful than linear kernels and allow separation of non-linearly separable data but they tend to not overfit as grossly as high degree polynomial kernels.

2

Gaussian RBF. It generalizes extremely well to real-world data, is smooth and doesn't overfit, and has been popular for a long time for those reasons.

3

Question 3 was done as can be seen in the `svm.py` file

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

```python
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])
    info_gain -= sum(map(lambda x: x[1] / total_counts * compute_entropy(data.loc[data[att_name] == x[0]][y_name]), zip(vals, counts)))
    #========================#
    #   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 -= sum(map(lambda count: count/total_counts * np.log2(count/total_counts), counts))
    #=======================#
    #   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  #
            #=======================#
            if len(np.unique(data[self.y_name])) > 0:
                leaf_value = np.unique(data[self.y_name])[0]
            #=======================#
            #   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
```

```
112             parent_node_class = np.unique(data[self.y_name])
[np.argmax(np.unique(data[self.y_name],return_counts=True)[1])]
113 
114             #Select the feature which best splits the dataset
115             if measure == 'info_gain':
116                 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
117             elif measure == 'gain_ratio':
118                 item_values = [comput_gain_ratio(data, feature, self.y_name) for
feature in features] #Return the gain_ratio for the features in the dataset
119             else:
120                 raise ValueError("kernel not recognized")
121 
122             best_feature_index = np.argmax(item_values)
123             best_feature = features[best_feature_index]
124             print('best_feature is: ', best_feature)
125 
126             #Create the tree structure. The root gets the name of the feature
(best_feature)
127             tree = {best_feature:{}}
128 
129 
130         #Grow a branch under the root node for each possible value of the root node
feature
131 
132         for value in np.unique(data[best_feature]):
133             #Split the dataset along the value of the feature with the largest
information gain and therwith create sub_datasets
134             sub_data = data.where(data[best_feature] == value).dropna()
135 
136             #Remove the selected feature from the feature space
137             sub_data = sub_data.drop(best_feature, axis = 1)
138 
139             #Call the ID3 algorithm for each of those sub_datasets with the new
parameters --> Here the recursion comes in!
140             subtree = self.make_tree(sub_data, parent_node_class)
141 
142             #Add the sub tree, grown from the sub_dataset to the tree under the root
node
143             tree[best_feature][value] = subtree
144 
145         return tree
146 
147 
148     def test(self, y_name):
149         accuracy = self.classify(self.test_data, y_name)
150         return accuracy
151 
152     def classify(self, test_data, y_name):
153         #Create new query instances by simply removing the target feature column
from the test dataset and
154         #convert it to a dictionary
155         test_x = test_data.drop(y_name, axis=1)
156         test_y = test_data[y_name]
157 
158         n = test_data.shape[0]
159         predicted_y = np.zeros(n)
160 
161         #Calculate the prediction accuracy
```

```
162         for i in range(n):
163             predicted_y[i] = DecisionTree.predict(self.tree, test_x.iloc[i])
164 
165         output = np.zeros((n,2))
166         output[:,0] = test_y
167         output[:,1] = predicted_y
168         accuracy = compute_accuracy(predicted_y, test_y.values)
169         return accuracy
170 
171     def predict(tree, query):
172         # find the root attribute
173         default = -1
174         for root_name in list(tree.keys()):
175             try:
176                 subtree = tree[root_name][query[root_name]]
177             except:
178                 return default ## root_name does not appear in query attribute list
    (it is an error!)
179 
180             ##if subtree is still a dictionary, recursively test next attribute
181             if isinstance(subtree,dict):
182                 return DecisionTree.predict(subtree, query)
183             else:
184                 leaf = subtree
185                 return leaf
186 
187 
```

```python
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 @ 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 @ 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  #
#     #=======================#
#     #=======================#
#     #   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]

    dists = np.linalg.norm(X[:,None,:] - Y[None,:,:], axis=-1)
    kernel_matrix = np.exp(-dists**2/(2 * sigma**2))

    return kernel_matrix

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)
            #=======================#
```

```
            # START YOUR CODE HERE  #
            #=======================#
            predicted_y = self.kernel(X, self.w[:, None].T).T[0] + self.b
            #=======================#
            #   END YOUR CODE HERE   #
            #=======================#
            return predicted_y

        else: ## non-linear case
            n = X.shape[0]
            predicted_y = np.zeros(n)
            #=======================#
            # START YOUR CODE HERE  #
            #=======================#
            predicted_y = (self.a * self.sv_y * self.kernel(X, self.sv)).sum(axis=1)
+ 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
```