q2_2

November 17, 2019

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import string
from sklearn.model_selection import KFold
from sklearn.metrics import classification_report
from sklearn.metrics import f1_score, precision_score, recall_score, accuracy_score, c
from statistics import mean
from collections import Counter
from copy import deepcopy
import operator
from math import log2
from sklearn.model_selection import train_test_split, KFold
import math
from sklearn import preprocessing
import random

%matplotlib inline
```

In [3]:
```
df = pd.read_csv('ensemble_data.csv')
df.head()
```

Out[3]:

	type	cap_shape	cap_surface	cap_color	bruises	odor	gill_attachment	\
0	p	x	s	n	t	p	f	
1	e	x	s	y	t	a	f	
2	e	b	s	w	t	l	f	
3	p	x	y	w	t	p	f	
4	e	x	s	g	f	n	f	

	gill_spacing	gill_size	gill_color	...	stalk_surface_below_ring	\
0	c	n	k	...	s	
1	c	b	k	...	s	
2	c	b	n	...	s	
3	c	n	n	...	s	
4	w	b	k	...	s	

	stalk_color_above_ring	stalk_color_below_ring	veil_type	veil_color	\

```
0                    w                      w         p          w
1                    w                      w         p          w
2                    w                      w         p          w
3                    w                      w         p          w
4                    w                      w         p          w

  ring_number ring_type spore_print_color population habitat
0           o         p                 k          s       u
1           o         p                 n          n       g
2           o         p                 n          n       m
3           o         p                 k          s       u
4           o         e                 n          a       g

[5 rows x 23 columns]
```

```
In [4]: Y = df.iloc[:, 0].values
        X = df
        X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, random_state=
```

```
In [5]: header = ['Label', 'Text', 'Length', 'Unigram', 'Bigram', 'Trigram']
        class Question:
            def __init__(self, column, value):
                self.column = column
                self.value = value

            def match(self, example):
                val = example[self.column]
                return self.value == val

            def __repr__(self):
                condition = "contains"
                return "Does %s %s %s?" % (
                    "Col" + str(self.column), condition, str(self.value))
```

```
In [6]: def class_counts(rows):
            counts = {}
            for row in rows:
                label = row[0]
                if label not in counts:
                    counts[label] = 0
                counts[label] += 1
            return counts
```

```
In [7]: class Leaf:
            def __init__(self, rows):
                self.predictions = class_counts(rows)
```

```
In [8]: class Decision_Node:
            def __init__(self,
```

```
                    question,
                    true_branch,
                    false_branch):
            self.question = question
            self.true_branch = true_branch
            self.false_branch = false_branch
```

In [9]:
```
def partition(rows, question):
    trueRows = []
    falseRows = []

    for r in rows:
        if question.match(r):
            trueRows.append(r)
        else:
            falseRows.append(r)

    return trueRows, falseRows
```

In [10]:
```
def formStump(rows, question):

    trueRows, falseRows = partition(rows, question)
    trueBranch = Leaf(trueRows)
    falseBranch = Leaf(falseRows)

    return Decision_Node(question, trueBranch, falseBranch)
```

In [11]:
```
def classifyRow(node, row):
    if isinstance(node, Leaf):
        if len(node.predictions) == 0:
            return {'p' : 1, 'e' : 0}
        return node.predictions

    if node.question.match(row):
        return classifyRow(node.true_branch, row)
    else:
        return classifyRow(node.false_branch, row)
```

In [12]:
```
def classify(root, rows):
    predictions = []
    for r in rows:
        predictions.append(max(classifyRow(root, r).items(), key=operator.itemgetter(
    return predictions
```

In [13]:
```
def getDataInIndex(data, index):
    l = []
    for i in range(len(data)):
        if i in index:
            l.append(data[i])
    return l
```

```
In [14]: def getActualLabels(act_data):
             act_labels = []
             for d in act_data:
                 act_labels.append(d[0])
             return act_labels

In [15]: def get_unique_vals(X_train, index):
             ans = []
             for i, r in X_train.iterrows():
         #         print(r)
                 ans.append(r[index])

             return set(ans)

In [16]: questions = []

         for i in range(1, X_train.shape[1]):
             unique_vals = get_unique_vals(X_train,int(i))
             for val in unique_vals:
                 questions.append(Question(i,val))

In [17]: len(questions)

Out[17]: 117

In [18]: def getListOfTrees(data, questions):
             rootList = []

             for q in questions:
                 rootList.append(formStump(data, q))

             return rootList

In [19]: def getIncorrectCount(root, data):
             count = 0
             ic_list = []

             i = 0
             for r in data:
                 label_pred = max(classifyRow(root, r).items(), key=operator.itemgetter(1))[0]
                 label_act = r[0]
                 if label_pred != label_act:
                     count += 1
                     ic_list.append(i)

                 i += 1

             return count, ic_list
```

```
In [20]: def getBestTree(rootList, data):
             bestRoot = None
             incorrectCount = math.inf
             incorrectList = []

             for root in rootList:
                 ic, ic_list = getIncorrectCount(root, data)

                 if ic < incorrectCount:
                     incorrectCount = ic
                     bestRoot = root
                     incorrectList = ic_list

             return root, incorrectList

In [21]: def appendWeights(data):
             n = len(data)
             ans = []

             for r in data:
                 r.append(1.0 / n)
                 ans.append(r)

             return ans

In [22]: def calcTotalError(data, ic_index):
             s = 0

             for i in ic_index:
                 s += data[i][-1]

             return s

In [23]: def getSignificance(te):
             if te == 0:
                 return 999999.0
             return 0.5 * math.log((1-te) / te)

In [24]: def updateWeights(db_train_weighted, incorrectList, siginficance):
             for i in range(len(db_train_weighted)):
                 if i in incorrectList:
                     db_train_weighted[i][-1]*= math.exp(siginficance)
                 else:
                     db_train_weighted[i][-1]*= math.exp(-siginficance)

In [25]: def update_normalise_col(col, db_train_weighted):
             sm = sum(col)

             for r in db_train_weighted:
```

```
            if sm == 0:
                r[-1] = 0
            else:
                r[-1] /= sm
```

In [26]:
```
def get_roullete_and_spin(db_train_weighted):
    n = len(db_train_weighted)

    db_ind = []
    for i in range(len(db_train_weighted)):
        d = db_train_weighted[i]
        n_ = int(math.floor(d[-1]*n))
        for j in range(n_):
            db_ind.append(i)

    #SPIN
    db = []

    if len(db_ind) == 0:
        return db

    for i in range(n):
        choice = db_ind[random.randint(0, len(db_ind) - 1)]
        db.append(db_train_weighted[choice])

    return db
```

In [27]:
```
def predict(rootList, significanceList, rows):

    ans = []

    for r in rows:
        outresult = {'p' : 0,
                'e' : 0}
        i = 0
        for root in rootList:
            pred = max(classifyRow(root, r).items(), key=operator.itemgetter(1))[0]
            outresult[pred] += significanceList[i]
            i += 1

        ans.append(max(outresult.items(), key=operator.itemgetter(1))[0])

    return ans
```

In [28]:
```
def Adaboost(no_of_iterations, data, questions):
    significanceList = []
    rootList = []
    db_train_weighted = appendWeights(data)
```

```
        for i in range(no_of_iterations):
            l = getListOfTrees(db_train_weighted, questions)
            bestTree, incorrectList = getBestTree(l, db_train_weighted)
            totalError = calcTotalError(db_train_weighted, incorrectList)
            significance = getSsignificance(totalError)

            rootList.append(bestTree)
            significanceList.append(significance)

            updateWeights(db_train_weighted, incorrectList, significance)

            last_col = [row[-1] for row in db_train_weighted]
            update_normalise_col(last_col, db_train_weighted)

            next_database = get_roullete_and_spin(db_train_weighted)

            if len(next_database) == 0:
                break

            db_train_weighted = next_database

        return rootList, significanceList
```

In [29]:
```
db_train = X_train.values.tolist()
db_test = X_test.values.tolist()
```

In [30]:
```
kfold = KFold(5, True, 1)
precision = []
recall = []
f_score = []
accuracy = []
i = 0

for trainInd,testInd in kfold.split(db_train):
    train_data = getDataInIndex(deepcopy(db_train), trainInd)
    test_data = getDataInIndex(deepcopy(db_train), testInd)

    rootl, significancel = Adaboost(3, train_data, questions)

    prediction = predict(rootl, significancel, test_data)

    actual = getActualLabels(test_data)
    predicted = prediction

    precision.append(precision_score(actual, predicted, average='macro'))
    recall.append(recall_score(actual, predicted, average='macro'))
```

```
            f_score.append(f1_score(actual, predicted, average='macro'))
            accuracy.append(accuracy_score(actual, predicted))

            print("Training...")

        print("Precision Score = " + str(mean(precision)))
        print("Recall Score = " + str(mean(recall)))
        print("F Score = " + str(mean(f_score)))
        print("Accuracy = " + str(mean(accuracy)))
```

```
Training...
Training...
Training...
Training...
Training...
Precision Score = 0.2601951797240481
Recall Score = 0.5
F Score = 0.3422160512783737
Accuracy = 0.5203903594480962
```