```python
'''
Machine Language Programming assignment 2
see the assignment description in the README.md file
'''

import numpy as np
import pandas as pd

print("Gaussian Naive Bayes Algorithm")
print ("Machine Language Programming assignment 2")
print("Author: Larry Bilodeau")
print()
print()

# define a function to process the feature labels.
def read_and_truncate_file(file_path):
    feature_labels = []
    with open(file_path, 'r') as file:
        for line in file:
            feature_label = line.split(':')[0]
            feature_labels.append(feature_label)
    return feature_labels

# Load the raw data
rawdata = pd.read_csv('..\datasets\spambase.data', header=None)
print("Data set: spambase.data")
print("rawdata shape", rawdata.shape)
# load the feature labels for the rawdata
feature_labels = read_and_truncate_file('..\\datasets\\feature_labels.txt')

#
# step 1: split the data into training and testing sets
#
# Define the labels
labels = rawdata.iloc[:, -1]

# Split the data into training and testing sets
# Select 2300 random rows with 40% labels 1 and 60% labels 0
test_data = rawdata.sample(n=2300, weights=labels.map({0: 0.6, 1: 0.4}), random_state=42)
print("test data shape", test_data.shape)

# Define the labels for test_data
test_targets = test_data.iloc[:, -1]
test_data = test_data.iloc[:, :-1]

# Remove the selected test_data rows from rawdata to get training_data
training_data = rawdata.drop(test_data.index)
# Define the labels for training_data
training_targets = training_data.iloc[:, -1]
# trim off the targets from the training data
training_data = training_data.iloc[:, :-1]
print("training data shape", training_data.shape)



#
# step 2: create probalistic models for each class
#

# Determine each of the hypothesis ratios for each hypotesis (H), based on the test data
# P(H|D) = P(D|H) * P(H) / P(D)
hypotheses, frequencies = np.unique(training_targets, return_counts=True)
hypothesis_ratios = frequencies / len(test_targets)

# Calculate the prior probabilities P(H)
# P(H) = P(D|H) * P(H) / P(D)

```

```
68  for h, f, r in zip(hypotheses, frequencies, hypothesis_ratios):
69      print()
70      print(f"{'Hypothesis':<15} {'Frequency':<15} {'Ratio':<15}")
71      print(f"{h:<15} {f:<15} {r:<15.2f}")
72      print()
73      print("Prior Probability of each hypothesis")
74      print("hyposis: spam,  P(h1):", hypothesis_ratios[1])
75      print("hyposis: not spam,  P(h0):", hypothesis_ratios[0])
76      print(" P(+|h1):", 1 - hypothesis_ratios[1], "P(-|h1):", hypothesis_ratios[1])
77      print(" P(+|h0):", 1 - hypothesis_ratios[0], "P(-|h0):", hypothesis_ratios[0])
78
79  # Filter rows for each unique value in training_targets
80  grouped_data = training_data.groupby(training_targets)
81
82  # Calculate the mean and standard deviation for each group
83
84  for clas, group in grouped_data:
85      mean = group.mean()
86      std_dev = group.std()
87      #print(f"\nGroup for target = {clas}")
88      #for col in range(len(mean)):
89      #    print(f"Column {col:<10} Mean: {mean[col]:<15.4f} \
90      # Standard Deviation: {std_dev[col]:<15.4f}")
91
92  # probalistic Model
93  # build the variance table from the means for each column and the column variance,
94  # where the variance is the square of the standard deviation
95  print()
96  variance_table_data = []
97  for clas, group in grouped_data:
98      mean = group.mean()
99      std_dev = group.std()
100     std_dev[std_dev == 0.0000] = 0.0001
101     variance = std_dev ** 2
102     variance[variance == 0.0000] = 0.0001
103     for i, feature_label in enumerate(feature_labels):
104         print(f"class: {clas:<10} feature: {feature_label:<30} mean: {mean[i]:<10.4f} \
105               std_dev: {std_dev[i]:<10.4f} variance: {variance[i]:<10.4f}")
106         variance_table_data.append({'Class': clas, 'Feature': feature_label, \
107                                     'Mean': mean[i], 'Standard Deviation': std_dev[i],\
108                                      'Variance': variance[i]})
109 variance_table = pd.DataFrame(variance_table_data)
110 variance_table_class0 = variance_table[variance_table['Class'] == 0]
111 variance_table_class1 = variance_table[variance_table['Class'] == 1]
112
113 #print(np.shape(variance_table)    )
114
115 print("\nVariance Table")
116 print(f"{'                                    class 0':37} \
117       {'                          class 1':<25}")
118 print(f"{'Feature':<30} {'Spam_mean':<10} {'Spam_variance':<15} {'Not Spam_mean':<13} \
119       {'Not Spam_variance':<16}")
120 row_size = training_data.shape[1] -1
121 for i in range(row_size):
122     print(f"{variance_table_class0.iloc[i, 1]:<30}", \
123           f"{variance_table_class0.iloc[i, 2]-1:<10.4f}", \
124           f"{variance_table_class0.iloc[i, 4]:<15.4f}", \
125           f"{variance_table_class1.iloc[i, 2]-1:<13.4f}", \
126           f"{variance_table_class1.iloc[i, 4]:<16.4f}")
127 print()
128
129 # step 3
130 # Gaussian Naive Bayes Algorithm `
131
132 #calculatethe gaussian probability density function
133
134 def gaussian_pdf(x, mean, variance):
```

```
135         if variance == 0:
136             variance = 0.0001
137         return (1 / np.sqrt(2 * np.pi * variance)) * \
138         np.exp(-((x - mean) ** 2) / (2 * variance))
139 
140     def class_probabilites(data, variance_table_class0, variance_table_class1,
    hypothesis_ratios):
141         # initialize the probabilities
142         probabilities = []
143         for i in range(len(data)):
144             # initialize the probabilities for each class
145             probabilities_class0 = 0
146             probabilities_class1 = 0
147             for j in range(len(data.iloc[i])):
148                 # calculate the probabilities for each class
149                 probabilities_class0 += np.log(gaussian_pdf(data.iloc[i, j], \
150                             variance_table_class0.iloc[j, 2], variance_table_class0.iloc[j, 4
                            ]))
151                 probabilities_class1 += np.log(gaussian_pdf(data.iloc[i, j], \
152                             variance_table_class1.iloc[j, 2], variance_table_class1.iloc[j, 4
                            ]))
153             # calculate the probabilities for each class
154             probabilities_class0 += np.log(hypothesis_ratios[0])
155             probabilities_class1 += np.log(hypothesis_ratios[1])
156             # append the probabilities for each class
157             probabilities.append([probabilities_class0, probabilities_class1])
158         return probabilities
159 
160     # calculate the class probabilities for the training data
161     probabilities = class_probabilites(training_data, variance_table_class0,
    variance_table_class1, \
162                                        hypothesis_ratios)
163     print()
164     print("Training_data class probabilities")
165     print("class 0", "      class 1")
166     for i in range(10):
167         print(f"{probabilities[i][0]:10.4f} {probabilities[i][1]:10.4f}")
168     print()
169 
170 
171     # calculate the class probabilities for the test data
172     probabilities = class_probabilites(test_data, variance_table_class0,
    variance_table_class1, \
173                                        hypothesis_ratios)
174     print("Test_data class probabilities")
175     [print("class 0", "            class 1")]
176     for i in range(10):
177         print(f"{probabilities[i][0]:10.4f} {' ' * 6} {probabilities[i][1]:10.4f}")
178     print()
179 
180     # calculate the accuracy, precision, and recall of the model on the test_data \
181     # and the test_targets
182     def accuracy_precision_recall(test_targets, probabilities):
183         # initialize the accuracy, precision, and recall
184         accuracy = 0
185         precision = 0
186         recall = 0
187         for i in range(len(test_targets)):
188             # determine the predicted class
189             predicted_class = 0 if probabilities[i][0] > probabilities[i][1] else 1
190             # determine the actual class
191             actual_class = test_targets.iloc[i]
192             # increment the accuracy, precision, and recall
193             accuracy += 1 if predicted_class == actual_class else 0
194             precision += 1 if predicted_class == 1 and actual_class == 1 else 0
195             recall += 1 if actual_class == 1 else 0
196         # calculate the accuracy, precision, and recall
```

```
    accuracy /= len(test_targets)
    precision /= recall
    recall /= len(test_targets)
    return accuracy, precision, recall

print()
print("Accuracy    Precision   Recall")
accuracy, precision, recall = accuracy_precision_recall(test_targets, probabilities)
print(f"{accuracy:.4f}     {precision:.4f}     {recall:.4f}")
print()

# generate a confusion matrix for the test data given the test_targets
def confusion_matrix(test_targets, probabilities):
    # initialize the confusion matrix
    confusion_matrix = np.zeros((2, 2))
    for i in range(len(test_targets)):
        # determine the predicted class
        predicted_class = 0 if probabilities[i][0] > probabilities[i][1] else 1
        # determine the actual class
        actual_class = test_targets.iloc[i]
        # increment the confusion matrix
        confusion_matrix[actual_class][predicted_class] += 1
    return confusion_matrix

print()
print("Confusion Matrix")
print(confusion_matrix(test_targets, probabilities))

```