**Name: Mozeb Ahmed Khan**

**Roll No: 20F-0161**

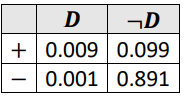
**Sec: BS (CS)-7A**

**Assignment: 03**

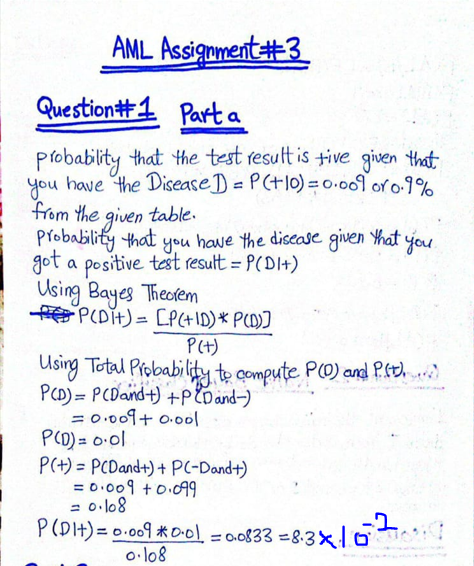
**Course: Applied Machine Learning**

**Question 1: Bayes Theorem**

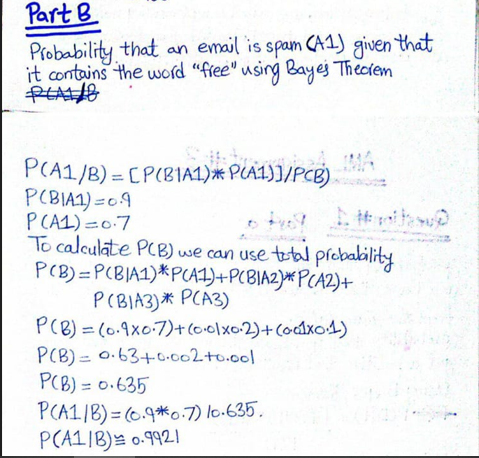
1. **A medical test for a disease D has outcomes + and −. The probabilities are:**



**What is the probability that the test will return a positive result for a sick person? If you go for a test and get a positive, what is the probability that you have the disease D?**

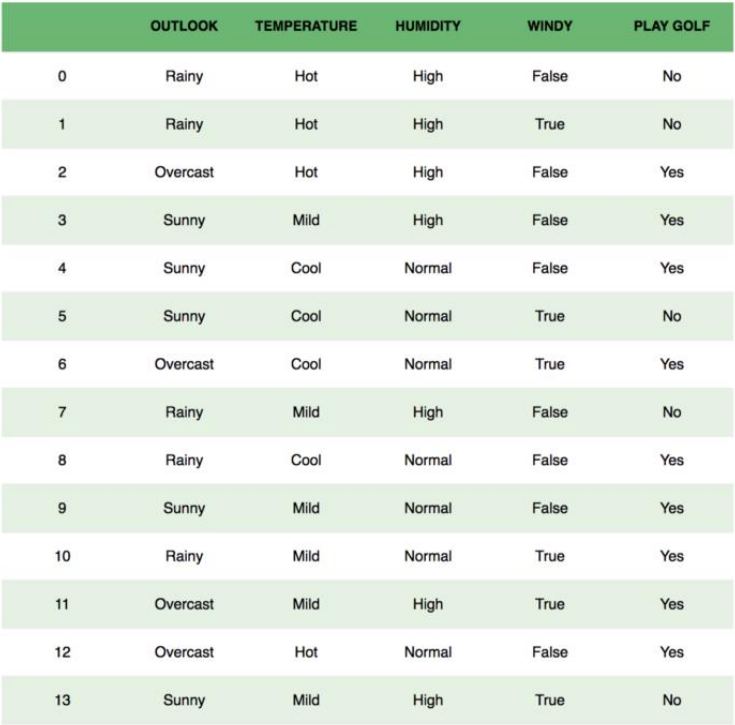


1. **Suppose you divide your email into three categories: A1 = spam, A2 =low priority, A3 =high priority. From previous experience you know that P(A1) = 0.7, P(A2) = 0.2, P(A3) = 0.1. Let B be the event that the email contains the word “free”. What is the probability that it is spam?**



**Question 2: Naïve Bayes Classifier**

**Implement the naïve bayes classifier on the following dataset. Your code should be flexible enough that may handle such types of data with different number of examples or the variable with different values.**



**Code:**

class NaiveBayesClassifier:

    def \_\_init\_\_(self):

        # Initialize dictionaries to store probabilities

        self.probabilities\_class = {}  # P(Play Golf = Yes), P(Play Golf = No)

        self.probabilities\_feature = {}  # P(Outlook | Play Golf), P(Temperature | Play Golf), ...

    def fit(self, data, labels):

        # Calculate class probabilities

        total\_samples = len(labels)

        unique\_classes, class\_counts = self.calculate\_counts(labels)

        for i, class\_label in enumerate(unique\_classes):

            # Calculate P(Play Golf = Yes) and P(Play Golf = No)

            self.probabilities\_class[class\_label] = class\_counts[i] / total\_samples

        # Calculate feature probabilities

        features = list(data.keys())

        for feature in features:

            # Find unique values for each feature

            unique\_values = self.get\_unique\_values(data[feature])

            self.probabilities\_feature[feature] = {}

            for class\_label in unique\_classes:

                self.probabilities\_feature[feature][class\_label] = {}

                for value in unique\_values:

                    # Calculate P(feature = value | Play Golf)

                    count = self.calculate\_feature\_count(data[feature], value, labels, class\_label)

                    total\_count = self.calculate\_total\_count(labels, class\_label)

                    self.probabilities\_feature[feature][class\_label][value] = count / total\_count

    def predict(self, sample):

        # Predict the class for a new sample

        predictions\_class = {}

        for class\_label, class\_prob in self.probabilities\_class.items():

            feature\_prob\_product = 1.0

            for feature, value in sample.items():

                # Multiply the probabilities for each feature value

                feature\_prob\_product \*= self.probabilities\_feature[feature][class\_label].get(value, 0.0)

            # Store the final probability for each class

            predictions\_class[class\_label] = class\_prob \* feature\_prob\_product

        # Return the class with the highest probability

        return max(predictions\_class, key=predictions\_class.get)

    def calculate\_counts(self, labels):

        # Calculate the count of each class in the dataset

        unique\_classes, class\_counts = [], []

        for label in labels:

            if label not in unique\_classes:

                unique\_classes.append(label)

                class\_counts.append(1)

            else:

                index = unique\_classes.index(label)

                class\_counts[index] += 1

        return unique\_classes, class\_counts

    def calculate\_feature\_count(self, feature\_values, value, labels, class\_label):

        # Calculate the count of a specific feature value for a given class

        count = 0

        for i in range(len(labels)):

            if feature\_values[i] == value and labels[i] == class\_label:

                count += 1

        return count

    def calculate\_total\_count(self, labels, class\_label):

        # Calculate the total count of a specific class in the dataset

        count = 0

        for label in labels:

            if label == class\_label:

                count += 1

        return count

    def get\_unique\_values(self, values):

        # Find unique values for a feature

        unique\_values = []

        for value in values:

            if value not in unique\_values:

                unique\_values.append(value)

        return unique\_values

# Example dataset

new\_data = {

    'Outlook': ['Rainy', 'Rainy', 'Overcast', 'Sunny', 'Sunny', 'Sunny', 'Overcast', 'Rainy', 'Rainy', 'Sunny', 'Rainy', 'Overcast', 'Overcast', 'Sunny'],

    'Temperature': ['Hot', 'Hot', 'Hot', 'Mild', 'Cool', 'Cool', 'Cool', 'Mild', 'Cool', 'Mild', 'Mild', 'Mild', 'Hot', 'Mild'],

    'Humidity': ['High', 'High', 'High', 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal', 'Normal', 'Normal', 'High', 'Normal', 'High'],

    'Windy': ['False', 'True', 'False', 'False', 'False', 'True', 'True', 'False', 'False', 'False', 'True', 'True', 'False', 'True'],

    'Play Golf': ['No', 'No', 'Yes', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No']

}

# Instantiate and train the Naive Bayes classifier

nb\_classifier = NaiveBayesClassifier()

nb\_classifier.fit(new\_data, new\_data['Play Golf'])

# Test the classifier with a new sample

new\_sample = {'Outlook': 'Rainy', 'Temperature': 'Cool', 'Humidity': 'High', 'Windy': 'True'}

prediction = nb\_classifier.predict(new\_sample)

# Display new sample and predicted class in tabular form

print("New Sample:")

print(f"{'Feature':<12} {'Value':<12}")

print("----------------------")

# Print feature-value pairs in tabular form

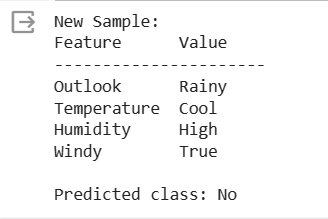
for feature, value in new\_sample.items():

    print(f"{feature:<12} {value:<12}")

# Display predicted class

print(f"\nPredicted class: {prediction}")

**Output:**



**Question 3: K-mean and K-medoid Clustering**

**Load the dataset available in the folder with the name “dataset” where there are four features A1, A2, A3, and A4. Your task is to implement the following,**

1. **K-mean clustering algorithm on the given dataset when the value for K is user-defined, that may be 2, 3, or 4, etc.**
2. **K-mediod clustering algorithm on the given dataset when the value for K is user-defined, that may be 2, 3, or 4, etc.**
3. **For visualization of the cluster, draw a scatter plot of the dataset and assign colors based on clusters computed through K-means and K-mediod methods.**

**Code:**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# Define the K-means clustering algorithm

class MyKMeans:

    def \_\_init\_\_(self, num\_clusters):

        # Initialize clusters

        self.num\_clusters = num\_clusters

        self.centroids = None

    def fit(self, input\_data):

        # Initialize centroids randomly

        self.centroids = input\_data[np.random.choice(input\_data.shape[0], self.num\_clusters, replace=False)]

        while True:

            # Assign each data point to the nearest centroid

            labels = self.assign\_labels(input\_data)

            # Update centroids based on the mean of assigned data points

            new\_centroids = np.array([input\_data[labels == i].mean(axis=0) for i in range(self.num\_clusters)])

            # Check for convergence

            if np.all(new\_centroids == self.centroids):

                break

            self.centroids = new\_centroids

        return labels

    def assign\_labels(self, input\_data):

        # Calculation of distances between data points and centroids

        distances = np.array([[np.linalg.norm(point - centroid) for centroid in self.centroids] for point in input\_data])

        # Assign labels by selecting the centroid index with the minimum distance for each data point

        labels = np.argmin(distances, axis=1)

        return labels

# Define the K-medoids clustering algorithm

class MyKMedoids:

    def \_\_init\_\_(self, num\_clusters):

        # Initialize clusters

        self.num\_clusters = num\_clusters

        self.medoids = None

    def fit(self, input\_data, max\_iter=100):

        num\_samples, num\_features = input\_data.shape

        # Initialize medoids randomly

        self.medoids = input\_data[np.random.choice(num\_samples, self.num\_clusters, replace=False)]

        for \_ in range(max\_iter):

            # Assign each data point to the nearest medoid

            labels, distances = self.calculate\_pairwise\_distances(input\_data)

            # Update medoids based on the total distance to other data points

            new\_medoids = np.array([input\_data[labels == i][np.argmin(distances[labels == i])] for i in range(self.num\_clusters)])

            # Check for convergence

            if np.all(new\_medoids == self.medoids):

                break

            self.medoids = new\_medoids

        return labels

    def calculate\_pairwise\_distances(self, input\_data):

        # Calculation of pairwise distances between data points and medoids

        distances = np.array([[np.linalg.norm(point - medoid) for medoid in self.medoids] for point in input\_data])

        # Find the index of the medoid with the minimum distance for each data point

        labels = np.argmin(distances, axis=1)

        # Find the minimum distances from each data point to its nearest medoid

        min\_distances = np.min(distances, axis=1)

        return labels, min\_distances

# Load the dataset

dataset = pd.read\_excel("dataset.xlsx").values

# Get user-defined value for K

num\_clusters\_value = int(input("Enter the number of clusters (K): "))

# Apply K-means clustering

kmeans = MyKMeans(num\_clusters=num\_clusters\_value)

kmeans\_labels = kmeans.fit(dataset)

# Apply K-medoids clustering

kmedoids = MyKMedoids(num\_clusters=num\_clusters\_value)

kmedoids\_labels = kmedoids.fit(dataset)

# Visualize the clusters with different colors

plt.scatter(dataset[:, 0], dataset[:, 1], c=kmeans\_labels, cmap='Set1', label='K-means')

plt.scatter(kmeans.centroids[:, 0], kmeans.centroids[:, 1], c='red', marker='X', label='K-means centroids')

plt.scatter(dataset[:, 0], dataset[:, 1], c=kmedoids\_labels, cmap='Set2', label='K-medoids')

plt.scatter(kmedoids.medoids[:, 0], kmedoids.medoids[:, 1], c='green', marker='X', label='K-medoids medoids')

plt.title(f'K-means and K-medoids Clustering (K={num\_clusters\_value})')

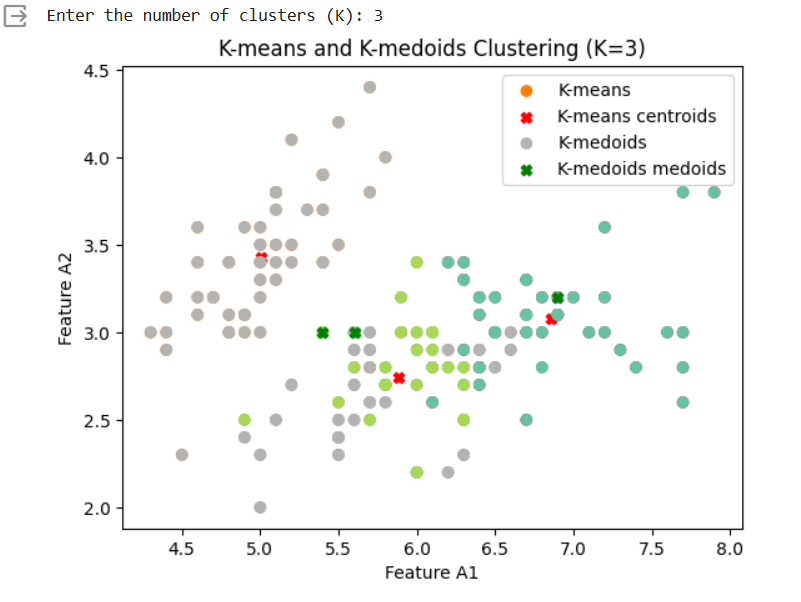
plt.xlabel('Feature A1')

plt.ylabel('Feature A2')

plt.legend()

plt.show()

**Output:**



**The End.**

**Thank You.**