# **Naive Bayes Implementation**

**Bayes Theorem**

* Based on prior knowledge of conditions that may be related to an event, Bayes theorem describes the probability of the event
* conditional probability can be found this way
* Assume we have a Hypothesis(*H*) and evidence(*E*),   
  According to Bayes theorem, the relationship between the probability of Hypothesis before getting the evidence represented as *P(H)* and the probability of the hypothesis after getting the evidence represented as *P(H|E)* is:

*P(H|E) = P(E|H)\*P(H)/P(E)*

* **Prior probability** = *P(H)* is the probability before getting the evidence   
  **Posterior probability** = *P(H|E)* is the probability after getting evidence
* In general,

*P(class|data) = (P(data|class) \* P(class)) / P(data)*

**Bayes Theorem Example**

Assume we have to find the probability of the randomly picked card to be king given that it is a face card.

There are *4* Kings in a Deck of Cards which implies that *P(King) = 4/52*

as all the Kings are face Cards so *P(Face|King) = 1*

there are *3* Face Cards in a Suit of *13 cards* and there are *4 Suits* in total so *P(Face) = 12/52*

Therefore,

*P(King|face) = P(face|king)\*P(king)/P(face) = 1/3*

**Code : Implementing Naive Bayes algorithm from scratch using Python**

| # Importing library  import math  import random  import csv      # the categorical class names are changed to numeric data  # eg: yes and no encoded to 1 and 0  def encode\_class(mydata):  classes = []  for i in range(len(mydata)):  if mydata[i][-1] not in classes:  classes.append(mydata[i][-1])  for i in range(len(classes)):  for j in range(len(mydata)):  if mydata[j][-1] == classes[i]:  mydata[j][-1] = i  return mydata      # Splitting the data  def splitting(mydata, ratio):  train\_num = int(len(mydata) \* ratio)  train = []  # initially testset will have all the dataset  test = list(mydata)  while len(train) < train\_num:  # index generated randomly from range 0  # to length of testset  index = random.randrange(len(test))  # from testset, pop data rows and put it in train  train.append(test.pop(index))  return train, test      # Group the data rows under each class yes or  # no in dictionary eg: dict[yes] and dict[no]  def groupUnderClass(mydata):  dict = {}  for i in range(len(mydata)):  if (mydata[i][-1] not in dict):  dict[mydata[i][-1]] = []  dict[mydata[i][-1]].append(mydata[i])  return dict      # Calculating Mean  def mean(numbers):  return sum(numbers) / float(len(numbers))    # Calculating Standard Deviation  def std\_dev(numbers):  avg = mean(numbers)  variance = sum([pow(x - avg, 2) for x in numbers]) / float(len(numbers) - 1)  return math.sqrt(variance)    def MeanAndStdDev(mydata):  info = [(mean(attribute), std\_dev(attribute)) for attribute in zip(\*mydata)]  # eg: list = [ [a, b, c], [m, n, o], [x, y, z]]  # here mean of 1st attribute =(a + m+x), mean of 2nd attribute = (b + n+y)/3  # delete summaries of last class  del info[-1]  return info    # find Mean and Standard Deviation under each class  def MeanAndStdDevForClass(mydata):  info = {}  dict = groupUnderClass(mydata)  for classValue, instances in dict.items():  info[classValue] = MeanAndStdDev(instances)  return info      # Calculate Gaussian Probability Density Function  def calculateGaussianProbability(x, mean, stdev):  expo = math.exp(-(math.pow(x - mean, 2) / (2 \* math.pow(stdev, 2))))  return (1 / (math.sqrt(2 \* math.pi) \* stdev)) \* expo      # Calculate Class Probabilities  def calculateClassProbabilities(info, test):  probabilities = {}  for classValue, classSummaries in info.items():  probabilities[classValue] = 1  for i in range(len(classSummaries)):  mean, std\_dev = classSummaries[i]  x = test[i]  probabilities[classValue] \*= calculateGaussianProbability(x, mean, std\_dev)  return probabilities      # Make prediction - highest probability is the prediction  def predict(info, test):  probabilities = calculateClassProbabilities(info, test)  bestLabel, bestProb = None, -1  for classValue, probability in probabilities.items():  if bestLabel is None or probability > bestProb:  bestProb = probability  bestLabel = classValue  return bestLabel      # returns predictions for a set of examples  def getPredictions(info, test):  predictions = []  for i in range(len(test)):  result = predict(info, test[i])  predictions.append(result)  return predictions    # Accuracy score  def accuracy\_rate(test, predictions):  correct = 0  for i in range(len(test)):  if test[i][-1] == predictions[i]:  correct += 1  return (correct / float(len(test))) \* 100.0      # driver code    # add the data path in your system  filename = r'E:\user\MACHINE LEARNING\machine learning algos\Naive bayes\filedata.csv'      # load the file and store it in mydata list  mydata = csv.reader(open(filename, "rt"))  mydata = list(mydata)  mydata = encode\_class(mydata)  for i in range(len(mydata)):  mydata[i] = [float(x) for x in mydata[i]]      # split ratio = 0.7  # 70% of data is training data and 30% is test data used for testing  ratio = 0.7  train\_data, test\_data = splitting(mydata, ratio)  print('Total number of examples are: ', len(mydata))  print('Out of these, training examples are: ', len(train\_data))  print("Test examples are: ", len(test\_data))    # prepare model  info = MeanAndStdDevForClass(train\_data)    # test model  predictions = getPredictions(info, test\_data)  accuracy = accuracy\_rate(test\_data, predictions)  print("Accuracy of your model is: ", accuracy) |
| --- |

**Output:**

Total number of examples are: 200

Out of these, training examples are: 140

Test examples are: 60

Accuracy of your model is: 71.2376788

-----------------X---------------X

At the end of the lecture, the mentor should tell the students not to get overwhelmed with such advanced code. This code is only to explain Naive Bayes better and it should be encouraged to the students to create some functions by themselves and try converting theoretical NB to practical code.

----------------------------------