OOP CLASS PROJECT

BAYES THEOREM AND ITS APPLICATION

Nguyen Anh Tai - 16128
Huynh Le Tuyet Anh - 16117
Dang Tran Tu - 16664
Department of Computer Science
Vietnamese German University

April 21, 2021

1 Bayes Theorem

Problem: Steve is very shy and withdrawn, invariably helpful but with very little interest in people or in the world of reality. A meek and tidy soul, He has a need for others and structure as well as a passion for detail. The majority of people would think that he is absolutely suitable as a librarian.
However, the point is not whether people hold correct or biased views about the personalities of librarians or farmers, it is that almost no one thinks to incorporate information about the ratio of farmers to librarians into their judgements. For example, we have 200 farmers compared to 10 librarians, the ratio of which is 20 to 1. And 40% of the librarians have the same personalities as mentioned in the problem and 10% of the farmers have those personalities. This corresponds to 4 librarians versus 20 farmers.
P(librarian given those personalities) = 16.7%
Even, we assume a librarian is 4 times as likely as a farmer to fit this description. That is not enough to overcome the fact that there are way more farmers.
The key underlying Bayes' theorem is that new evidence should not completely determine your beliefs in a vacuum, it should update your prior beliefs, as testified by the equation below:

$$\Pr(A|B) = \frac{\Pr(B|A)\Pr(A)}{\Pr(B|A)\Pr(A) + \Pr(B|\neg A)\Pr(\neg A)} = \frac{\Pr(B|A)\Pr(A)}{\Pr(B)} \tag{1}$$

2 Probability knowledge

2.1 Terminology

1. Mean or Expected value (μ): the weighted average of the possible values.
2. Median: Middle value of the list
3. Standard deviation (σ): implies how spread out the data is. It is a measure of how far each observed value is from the mean.
4. Variance (σ^2): the expected value of the squared deviation

2.2 Probability Mass Function

Probability Mass Function (PMF) is the probability distribution of a discrete random variable providing the possible values and their associated probabilities

$$\begin{aligned} &\sum p_X(x_i) = 1 \\ &p(x_i) > 0 \\ &p(x) = 0 \text{ for all other } \mathbf{x} \end{aligned} \tag{2}$$

2.2.1 Bernoulli Trial

Bernoulli Trial is a success or failure experiment with the probability mass function, the expected value and the variance respectively:

$$f(k,n,p) = \Pr(k,n,p) = \Pr(X=k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{3}$$

$$E(x) = np \tag{4}$$

$$Var(X) = np(1-p) \tag{5}$$

2.3 Probability Density Function

In probability theory, a probability density function (PDF), or density of a continuous random variable, is a function whose value at any given sample (or point) in the sample space (the set of possible values taken by the random variable) can be interpreted as providing a relative likelihood that the value of the random variable would equal that sample. In other words, while the absolute likelihood for a continuous random variable to take on any particular value is 0 (since there is an infinite set of possible values to begin with), the value of the PDF at two different samples can be used to infer, in any particular draw of the random variable, how much more likely it is that the random variable would equal one sample compared to the other sample.

2.3.1 Gaussian Distribution

Gaussian Distribution is a type of continuous probability distribution for a real-valued random variable and is used to represent real-valued random variables whose distributions are not known. The general form of its probability density function is:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{6}$$

Central limit theorem: Under some conditions, the average of many samples (observations) of a random variable with finite mean and variance is itself a random variable—whose distribution converges to a normal distribution as the number of samples increases.

3 Probabilistic Model

Naive Bayes is a conditional probability model. This means that given a problem instance to be classified, which is represented by a vector x=(x_1,...x_n) representing some n independent variables, it assigns to this instance probabilities p($C_k|x_1$,x_2,...,x_n) for each of K possible outcomes. Using Bayes' theorem, we can reformulate the model as:

$$p(C_k|x) = \frac{p(C_k)p(x|C_k)}{p(x)} \tag{7}$$

In practice, the numerator of the fraction is of greater importance since the denominator is independent of the class and the value of the features are presented in the data set, which means the denominator is effectively constant and does not interfere with the result. We have the joint probability model which is P(A,B)=P(A|B)P(B). By applying the joint probability model and the chain rule for repeated applications of the definition of conditional probability:

$$\begin{aligned} p(C_k, x_1, ..., x_n) &= p(x_1, ..., x_n, C_k) \\ &= p(x_1|x_2, ..., x_n, C_k)p(x_2, ..., x_n, C_k) = ... \\ &= p(x_1|x_2, ..., x_n, C_k)p(x_2|x_3, ..., x_n, C_k)... \\ &= p(x_1|x_2, ..., x_n, C_k)...p(x_{n-1}|x_n, C_k)p(x_n|C_k)p(C_k) \end{aligned} \tag{8}$$

Based on the naive conditional independence assumptions which is that all features are mutually independent, conditional on the category C_k. Therefore we

have:

$$p(x_i|x_{i+1},...,x_n,C_k) = p(x_i|C_k) \tag{9}$$

Ab the joint probability model can be expressed below:

$$\begin{aligned} p(C_k|x_1,...,x_n) &\propto p(C_k,x_1,...,x_n) \\ &\propto p(C_k)p(x_1|C_k)p(x_2|C_k)p(x_3|C_k)... \\ &\propto p(C_k)\prod_{i=1}^{n} p(x_i|C_k) \end{aligned} \tag{10}$$

4 Gaussian Naive Bayes

When working with continuous data, an assumption often taken is that the continuous values associated with each class are distributed according to a normal (or Gaussian) distribution. The likelihood of the features is assumed to be Gaussian Naive Bayes supports continuous valued features and models each as conforming to a Gaussian (normal) distribution.
An approach to create a simple model is to assume that the data is described by a Gaussian distribution with no co-variance (independent dimensions) between dimensions. This model can be fit by simply finding the mean and standard deviation of the points within each label, which is all what is needed to define such a distribution.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{11}$$

Probability distribution of value v given a class C_k:

$$p\left(x = v \mid C_k\right) = \frac{1}{\sqrt{2\pi\sigma_k^2}} e^{-\frac{(v-\mu_k)^2}{2\sigma_k^2}} \tag{12}$$

5 Gaussian Naive Bayes Sample Code

```
#train.py
import pandas as pd
import matplotlib.pyplot as plt
import pickle #for saving model
from sklearn.model_selection import train_test_split

from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

from sklearn.metrics import classification_report

df = pd.read_csv("weight-height.csv")

#model to split data
train, test = train_test_split(df, test_size=0.2)
print("Test data set")
print(test.head())
print()
```

```
print("Train data set")
print(train.head())
print('')
print("No. of data in test:" +str(len(test)))
print("No. of data in train:" +str(len(train)))

print(train.groupby('Gender').count())
#creating x and y variables
feature_names = ['Height', 'Weight']
x_train = train[feature_names].values.tolist()
y_train = train['Gender']
x_test = test[feature_names].values.tolist()
y_test = test['Gender']

classifier = GaussianNB()
classifier = classifier.fit(x_train, y_train)
predictions = classifier.predict(x_test)

score = accuracy_score(y_test, predictions)
print("AccuracyScore: ", score)

filename = 'model.sav'
pickle.dump(classifier, open(filename, 'wb'))

#main.py
import pickle #for saving model

#load model
loaded_model = pickle.load(open('model.sav', 'rb'))

#Making a prediction for a user
height = input("Input HEIGHT (in cm): ")
weight = input("Input WEIGHT (in kg): ")
input_data = [[int(height) * 0.393701, int(weight) * 2.20462]]
prediction = loaded_model.predict(input_data)

print("\nPrediction probability: ")
print("Female: ", loaded_model.predict_proba(input_data)[0][0])
print("Male: ", loaded_model.predict_proba(input_data)[0][1])
print("\nThis human is probably BIOLOGICALLY a", str(prediction[0]))
```

6 Multinomial Naive Bayes

Multinomial Naive Bayes implements the Naive Bayes algorithm for multinomially distributed data. The distribution is parametrized by vectors θ_y=(θ_{y_1},...,θ_{y_n}) for each class y where n is the number of features (in text classification, the size of the vocabulary) and θ_y is the probability P(x_i|y) for feature i appearing in a sample belonging to class y.
The parameters θ_y is estimated by a smoothes version of maximum likelihood

or relative frequency counting:

$$\theta_{y_i} = \frac{N_{y_i} + \alpha}{N_y + n\alpha} \tag{13}$$

where N_{y_i}=$\sum_{x \in T} x_i$ is the number of times feature i appears in a sample of class y in the training set T, and N_y=$\sum_{i=1}^{n} N_{y_i}$ is the total count of all features in the class y.
In order to avoid the problem of zero probabilities, an additional smoothing term can be added to the multinomial Bayes model. The most common variants of additive smoothing are the so-called Lidstone smoothing (alpha¡1) and Laplace smoothing (alpha=1)

7 Spam Filtering Sample Code

```
import pandas as pd
sms_spam=pd.read_csv('SMSSpamCollection', sep='\t',
header=None, names=['Label', 'SMS'])
print(sms_spam.shape)
# .shape gives the number of rows and columns
sms_spam.head()
sms_spam['Label'].value_counts(normalize=True)
#This is used for estimating the distribution percentage
#of spam and not spam messages

# Randomize the dataset
data_randomized = sms_spam.sample(frac=1, random_state=1)

# Calculate index for split
training_test_index = round(len(data_randomized) * 0.8)
# This is used for estimating the length of training dataset

# Split into training and test sets
training_set = data_randomized[:training_test_index].reset_index(drop=True)
test_set = data_randomized[training_test_index:].reset_index(drop=True)
# reset_index(drop=True) is used to drop all rows containing NaN values
# and reset the index so that you do not have any missing rows

print(training_set.shape)
print(test_set.shape)

training_set['Label'].value_counts(normalize=True)

#Before cleaning
training_set.head(5)
#After cleaning
training_set['SMS']=training_set['SMS'].str.replace('\W',' ')#remove punctuation
training_set['SMS']=training_set['SMS'].str.lower()
training_set.head(5)
```

```
training_set['SMS']=training_set['SMS'].str.split()
vocabulary=[]
for sms in training_set['SMS']:
for word in sms:
vocabulary.append(word)
vocabulary=list(set(vocabulary))

len(vocabulary)

# outputs a list of the length of training_set['SMS'].
word_counts_per_sms = {unique_word: [0] *
len(training_set['SMS']) for unique_word in vocabulary}

for index, sms in enumerate(training_set['SMS']):
for word in sms:
word_counts_per_sms[word][index] += 1

word_counts = pd.DataFrame(word_counts_per_sms)
word_counts.head()

training_set_clean = pd.concat([training_set, word_counts], axis=1)
training_set_clean.head()


# Isolating spam and notspam messages first
spam_messages = training_set_clean[training_set_clean['Label'] == 'spam']
notspam_messages = training_set_clean[training_set_clean['Label'] == 'notspam']

# P(Spam) and P(notspam)
p_spam = len(spam_messages) / len(training_set_clean)
p_notspam = len(notspam_messages) / len(training_set_clean)

# N_Spam
n_words_per_spam_message = spam_messages['SMS'].apply(len)
n_spam = n_words_per_spam_message.sum()

# N_notspam
n_words_per_notspam_message = notspam_messages['SMS'].apply(len)
n_notspam = n_words_per_notspam_message.sum()

# N_Vocabulary
n_vocabulary = len(vocabulary)

# Laplace smoothing
alpha = 1

# Initiate parameters
parameters_spam = {unique_word:0 for
unique_word in vocabulary}
```

```
parameters_notspam = {unique_word:0 for
unique_word in vocabulary}

# Calculate parameters
for word in vocabulary:
n_word_given_spam = spam_messages[word].sum() # spam_messages already defined
p_word_given_spam = (n_word_given_spam + alpha) / (n_spam + alpha*n_vocabulary)
parameters_spam[word] = p_word_given_spam

n_word_given_notspam = notspam_messages[word].sum() # notspam_messages already defined
p_word_given_notspam = (n_word_given_notspam + alpha) / (n_notspam + alpha*n_vocabulary)
parameters_notspam[word] = p_word_given_notspam


import re

def classify(message):
'''
message: a string
'''

message = re.sub('\W', ' ', message)
message = message.lower().split()

p_spam_given_message = p_spam
p_notspam_given_message = p_notspam

for word in message:
if word in parameters_spam:
p_spam_given_message *= parameters_spam[word]

if word in parameters_notspam:
p_notspam_given_message *= parameters_notspam[word]

print('P(Spam|message):', p_spam_given_message)
print('P(notspam|message):', p_notspam_given_message)

if p_notspam_given_message > p_spam_given_message:
print('Label: notspam')
elif p_notspam_given_message < p_spam_given_message:
print('Label: Spam')
else:
print('Equal probabilities')
m='y'
while m!='n':
comment=input()
classify(comment)
m=input()

def classify_test_set(message):
```

```
'''
message: a string
'''

message = re.sub('\W', ' ', message)
message = message.lower().split()

p_spam_given_message = p_spam
p_notspam_given_message = p_notspam

for word in message:
if word in parameters_spam:
p_spam_given_message *= parameters_spam[word]

if word in parameters_notspam:
p_notspam_given_message *= parameters_notspam[word]

if p_notspam_given_message > p_spam_given_message:
return 'notspam'
elif p_spam_given_message > p_notspam_given_message:
return 'spam'
else:
return 'cannot classify'

test_set['predicted'] = test_set['SMS'].apply(classify_test_set)
test_set.head()
correct = 0
total = test_set.shape[0]

for row in test_set.iterrows():
row = row[1]
if row['Label'] == row['predicted']:
correct += 1

print('Correct:', correct)
print('Incorrect:', total - correct)
print('Accuracy:', correct/total)
```

8 Racism Sentiment Analysis

```
from __future__ import division
import pandas as pd

#PART 1
def split_text_train(text_in_label):
data = []
data_dict ={'0': [], '1': []}
vocabulary = 0
PR = 0
```

```
PnR = 0
sumdata = 0

for row in text_in_label:
sumdata += 1
if row[1] == 1:
split = row[0].split()
PR += 1

for i in split:
data.append(i)
if i not in data_dict['1']:
vocabulary += 1
data_dict['1'].append(i)

if row[1] == 0:
split = row[0].split()
PnR += 1

for i in split:
data.append(i)
if i not in data_dict['0']:
vocabulary += 1
data_dict['0'].append(i)

return data, data_dict, vocabulary, PR, PnR, sumdata

#PART 2
def count_class_freq(dictionary):
data_dict_hate = {}
data_dict_not_hate = {}

for word in dictionary['1']:
if word not in data_dict_hate:
data_dict_hate[word] = 0

for word in dictionary['0']:
if word not in data_dict_not_hate:
data_dict_not_hate[word] = 0

return data_dict_hate, data_dict_not_hate

#PART 3: TRAINING
def training(dataset, test_data, data_dictionary, dict_hate_words, dict_not_hate_words,
vocabulary, alpha = 1):
prob_hate_words = {}
prob_not_hate_words = {}
hate_words = {}
not_hate_words = {}
num_of_word_in_hate = len(data_dictionary['1'])
```

```
num_of_word_in_not_hate = len(data_dictionary['0'])

for word in dataset:
if word not in dict_hate_words:
hate_words[word] = 0 #this line is used for adding the word to the set

if word in dict_hate_words:
if word in hate_words:
hate_words[word] += 1 #increment the frequency
if word not in hate_words:
hate_words[word] = 0 #add new words to hate words

if word not in dict_not_hate_words:
not_hate_words[word] = 0 #this line is used for adding the words to the set

if word in dict_not_hate_words:
if word in not_hate_words:
not_hate_words[word] += 1 #increment the frequency
if word not in not_hate_words:
not_hate_words[word] = 0 #this line is used for adding the words

for word in test_data:

if word not in dict_hate_words:
hate_words[word] = 0

if word not in dict_not_hate_words:
not_hate_words[word] = 0

for word in hate_words:
prob_hate_words[word] = (hate_words[word] + alpha)/(num_of_word_in_hate + vocabulary)

for word in not_hate_words:
prob_not_hate_words[word] = (not_hate_words[word] + alpha)/(num_of_word_in_not_hate
+ vocabulary)

return prob_hate_words, prob_not_hate_words

#PART 4
def predict(test_data, dict_prob_hate, dict_prob_not_hate, Phate, Pnothate, sumdata):
probability_hate = []
probability_not_hate = []
prediction = []
hate_meter = 1
not_hate_meter = 1

for word in test_data:
if word in dict_prob_hate:
probability_hate.append(dict_prob_hate[word])
if word in dict_prob_not_hate:
```

```
probability_not_hate.append(dict_prob_not_hate[word])
else:
probability_hate.append(dict_prob_hate[word])
probability_not_hate.append(dict_prob_not_hate[word])

for value in probability_hate:
hate_meter *= value
hate_meter = hate_meter * (Phate/sumdata)

# print('hate meter:  %s' %(hate_meter))

for value in probability_not_hate:
not_hate_meter *= value
not_hate_meter = not_hate_meter * (Pnothate/sumdata)

# print('Not hate meter: %s' %(not_hate_meter))

if hate_meter > not_hate_meter:
prediction.append(1)
if not_hate_meter > hate_meter:
prediction.append(0)
if hate_meter == not_hate_meter:
prediction.append(None)

prediction_val = max(hate_meter, not_hate_meter)

return prediction, prediction_val, not_hate_meter, hate_meter


df = pd.read_csv('twitter_racism_parsed_dataset.csv', encoding='latin-1')

x = df.iloc[:, 0]
y = df.iloc[:, 1]

data = []
for i in range(len(df)):
data.append([x[i], y[i]])

dataset, data_dictionary, vocab, Phate, Pnothate, sumdata = split_text_train(data)
dict_hate, dict_not_hate = count_class_freq(data_dictionary)
# Stop Words are words which do not contain important significance to be used in
# Search Queries.
# Usually, these words are filtered out from search queries because they return a
# vast amount of unnecessary information.
stop_words = ["a", "about", "above", "after", "again", "against", "ain", "all", "am",
"an", "and", "any", "are", "aren", "aren't",
"as", "at", "be", "because", "been", "before", "being", "below", "between", "both",
"but", "by", "can", "couldn", "couldn't",
"d", "did", "didn", "didn't", "do", "does", "doesn", "doesn't", "doing", "don", "don't",
"down", "during", "each", "few", "for",
```

```
"from", "further", "had", "hadn", "hadn't", "has", "hasn", "hasn't", "have", "haven",
"haven't", "having", "he", "her", "here",
"hers", "herself", "him", "himself", "his", "how", "i", "if", "in", "into", "is", "isn",
"isn't", "it", "it's", "its", "itself",
"just", "ll", "m", "ma", "me", "mightn", "mightn't", "more", "most", "mustn", "mustn't",
"my", "myself", "needn", "needn't", "no",
"nor", "not", "now", "o", "of", "off", "on", "once", "only", "or", "other", "our",
"ours", "ourselves", "out", "over", "own", "re",
"s", "same", "shan", "shan't", "she", "she's", "should", "should've", "shouldn",
"shouldn't", "so", "some", "such", "t", "than", "that",
"that'll", "the", "their", "theirs", "them", "themselves", "then", "there", "these",
"they", "this", "those", "through", "to", "too", "under",
"until", "up", "ve", "very", "was", "wasn", "wasn't", "we", "were", "weren", "weren't",
"what", "when", "where", "which", "while", "who", "whom",
"why", "will", "with", "won", "won't", "wouldn", "wouldn't", "y", "you", "you'd",
z"you'll", "you're", "you've", "your", "yours", "yourself", "yourselves",
"could", "he'd", "he'll", "he's", "here's", "how's", "i'd", "i'll", "i'm", "i've",
"let's", "ought", "she'd", "she'll", "that's", "there's", "they'd", "they'll",
"they're", "they've", "we'd", "we'll", "we're", "we've", "what's", "when's", "where's",
"who's", "why's", "would"]

def execute_single_test():

# print(type(comment))
m='y'
while m!='n':
comment=input()
comment=comment.lower()
test_data = comment.split()
for count in range(len(test_data)):
for i, filter in enumerate(test_data):
if filter in stop_words:
test_data.pop(i)
prob_hate, prob_not_hate = training(dataset, test_data, data_dictionary, dict_hate,
dict_not_hate, vocab)
pred, val, no_hate_meter, hate_meter = predict(test_data, prob_hate, prob_not_hate,
Phate, Pnothate, sumdata)

print(('\nPREDICTION:  %s \nprobability measure:  %s') %(pred, val))
print(('\nhate meter:  %s\nNot hate meter:  %s') %(hate_meter, no_hate_meter))
if pred[0]==0:
print("Your input is acceptable!")
else:
print("Your input is related to hatred!")

print(len(data_dictionary['1']), ':1   =    0:', len(data_dictionary['0']))
# print(Phate, Pnothate, sumdata)
# print(vocab)
# print(len(dataset))
# print(prob_not_hate)
```

```
m=input()

def execute_multiple_test():
dg = pd.read_csv('youtube_parsed_dataset.csv', encoding='latin-1')
x_test = dg.iloc[:, 0]
y_test = dg.iloc[:, 1]
label = []
TP = 0
TN = 0
FP = 0
FN = 0

testing = []
for i in range(len(dg)):
testing.append([x_test[i], y_test[i]])

for i, row in enumerate(testing):
split_data = row[0].split()
label.append(row[1])

for c, word in enumerate(split_data):
if word in stop_words:
split_data.pop(c)

prob_hate, prob_not_hate = training(dataset, split_data, data_dictionary, dict_hate,
dict_not_hate, vocab)
pred, val, no_hate_meter, hate_meter = predict(split_data, prob_hate, prob_not_hate,
Phate, Pnothate, sumdata)

if pred[0] == label[-1]:
if pred[0] == 1:
TP += 1
if pred[0] == 0:
TN += 1

if pred[0] != label[-1]:
if pred[0] == 1:
FP += 1
if pred[0] == 0:
FN += 1

Acc = ((TP + TN)/(TP + TN + FP + FN) * 100)


print('\nAccuracy is %s%s' %(Acc, '%'))

execute_single_test()
execute_multiple_test()
```

9 Bernoulli Naive Bayes

This is used for discrete data and it works on Bernoulli distribution. The main feature of Bernoulli Naive Bayes is that it accepts features only as binary values like true or false, yes or no, success or failure, 0 or 1 and so on. So when the feature values are binary we know that we have to use Bernoulli Naive Bayes classifier.
Bernoulli distribution: As we deal with binary values, let's consider 'p' as probability of success and 'q' as probability of failure and q=1-p. For a random variable 'X' in Bernoulli distribution,where 'x' can have only two values either 0 or 1. Bernoulli distribution:

$$F(x) = P[X = x] = \begin{cases} q = 1 - p & x = 0 \\ p & x = 1 \end{cases}$$

Bernoulli Naive Bayes Classifier is based on the following rule:

$$P(x_i|y) = P(i|y)x_i + (1 - P(i|y))(1 - x_i) \tag{14}$$

Advantages of Bernoulli Naive Bayes:
1. They are extremely fast as compared to other classification models.
2. As in Bernoulli Naive Bayes each feature is treated independently with binary values only, it explicitly gives penalty to the model for non-occurrence of any of the features which are necessary for predicting the output y. And the other multinomial variant of Naive Bayes ignores these features instead of penalizing.
3. In case of small amounts of data or small documents(for example in text classification), Bernoulli Naive Bayes gives more accurate and precise results as compared to other models.
4. It is fast and is able to make real-time predictions.
5. It can handle irrelevant features nicely.
6. Results are self explanatory.
Disadvantages of Bernoulli Naive Bayes:
1. Being a naive(showing a lack of experience) classifier, it sometimes makes a strong assumption based on the shape of data.
2. If at times the features are dependent on each other then Naive Bayes assumptions can affect the prediction and accuracy of the model and is sensitive to the given input data.
3. If there is a categorical variable which is not present in training dataset, it results in zero frequency problem. This problem can be easily solved by Laplace estimation.

10 How to increase the accuracy of Naive Bayes Classifier

1. Take the logarithm of your probabilities as input features: We change the probability space to log probability space since we calculate the probability by multiplying probabilities and the result will be very small. When we change to log probability features, we can tackle the under-runs problem.
2. Remove correlated feature: Naive Bayes works based on the assumption of

independence when we have a correlation between features which means one feature depends on others then our assumption will fail.
3. Work with enough data not the huge data: Naive Bayes require less data than logistic regression since it only needs data to understand the probabilistic relationship of each attribute in isolation with the output variable, not the interactions.
4. Check zero frequency error: If the test data set has zero frequency issue, apply smoothing techniques "Laplace Correction" to predict the class of test data set.