When 100 samples for training with gaussian distribution

Test Instances Number 4077
Correctly Classified Instances 2285
Misclassification Instances 1792
TP for young : 678
TP for middle-aged : 1100
TP for old : 507
[[678, 0, 0],
[0, 1100, 0],
[0, 0, 507]]
Mean Accuracy: 56.046

When 1000 samples for training with gaussian distribution

Test Instances Number 3177
Correctly Classified Instances 1782
Misclassification Instances 1395
TP for young : 565
TP for middle-aged : 861
TP for old : 356
[[565, 0, 0],
[0, 861, 0],
[0, 0, 356]]
Mean Accuracy: 56.091

When 100 samples for training with Naive Estimator

Test Instances Number 4077
Correctly Classified Instances 2598
Misclassification Instances 1479
TP for young : 452
TP for middle-aged : 2097
TP for old : 49
[[452, 0, 0],
[0, 2097, 0],
[0, 0, 49]]
Mean Accuracy: 63.723

When 1000 samples for training with Naive Estimator

Test Instances Number 3177
Correctly Classified Instances 2125
Misclassification Instances 1052
TP for young : 396
TP for middle-aged : 1723
TP for old : 6
[[396, 0, 0],
[0, 1723, 0],
[0, 0, 6]]
Mean Accuracy: 66.887

I obtained max accuracy by using Naive Estimator when 1000 samples for training.
I regulated bin size according to values of columns and as I regulated, accuracy increased.
When I use 1000 training set, because of well-done trained algorithm, I obtained more beautiful result.

```
************************************* gaussian distribution************************************
# Naive Bayes On The Abalone Dataset
from random import randrange
from math import sqrt
from math import exp
from math import pi
from random import seed

def load_txt(filename):
    dataset = list()
    with open(filename) as txt_file:
        my_list = txt_file.readlines()
        for row in my_list:
            dataset.append(row.split())
    return dataset

# for continuous values, convert string to float
def convert_str_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())

# for last column, string to int '1' -> 1
def convert_str_to_int_for_output(dataset, column):
    for row in dataset:
        row[column] = int(row[column])

def convert_str_to_int(dataset, column):
        class_values = [row[column] for row in dataset]
        unique = set(class_values)
        lookup = dict()
        for i, value in enumerate(unique):
                lookup[value] = i
        for row in dataset:
                row[column] = lookup[row[column]]
        return lookup

def train_test_split(dataset, train_numbers):
    dataset_training = list()
    while len(dataset_training) < train_numbers:
        index = randrange(len(dataset))
        dataset_training.append(dataset.pop(index))
    return dataset_training

def mean(numbers):
    return sum(numbers)/float(len(numbers))

def calculate_stdev(numbers):
    avg = mean(numbers)
    variance = sum([(x-avg)**2 for x in numbers]) / float(len(numbers)-1)
    return sqrt(variance)

def calculate_Gaussian_probability(x, mean, stdev):
    exponent = exp(-((x-mean)**2 / (2 * stdev**2)))
    return (1 / (sqrt(2 * pi) * stdev)) * exponent

def calculate_first_column_prob(separated,row,class_value):
    count=0
```

```
    for i in range(len(separated[class_value])):
        if row[0] == separated[class_value][i][0]:
            count += 1
    prob = count/ len(separated[class_value])
    return prob

def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    misclassified_instances = len(actual) - correct
    print("%s%s" % ('Test Instances Number  ', len(actual)))
    print("%s%s" % ('Correctly Classified Instances  ', correct))
    print("%s%s" % ('Misclassification Instances  ', misclassified_instances))
    return correct / float(len(actual)) * 100.0

def confusionMatrix(actual, predicted):
        tp1=0
        tp2=0
        tp3=0
        for i in range(len(actual)):
            if actual[i] == predicted[i] == 1:
                tp1 += 1
            if actual[i] == predicted[i] == 2:
                tp2 += 1
            if actual[i] == predicted[i] == 3:
                tp3 += 1
        our_confusion_matrix = [[tp1, 0, 0],[0, tp2, 0],[0, 0, tp3]]

        print(our_confusion_matrix)

# Split the dataset by class values, returns a dictionary
def separate_by_class(train_dataset):
    separated = dict()
    for i in range(len(train_dataset)):
        vector = train_dataset[i]
        class_value = vector[-1]
        if (class_value not in separated):
            separated[class_value] = list()
        separated[class_value].append(vector)
    return separated

# Calculate the mean, stdev and count for each column in a dataset
def summarize_dataset(train_dataset_row):
    summaries = [(mean(column), calculate_stdev(column), len(column))
            for column in zip(*train_dataset_row)]
    del(summaries[-1])
    del(summaries[0])
    return summaries

# Evaluate an algorithm using a train_test_split
def evaluate_algorithm(dataset, algorithm, train_numbers, *args):
    training_set = train_test_split(dataset, train_numbers)
    test_set = list(dataset)
    dataset.clear()

    predicted = algorithm(training_set, test_set, *args)
    actual = [row[-1] for row in test_set]
    accuracy = accuracy_metric(actual, predicted)
```

```
    confusionMatrix(actual, predicted)
    return accuracy

def naive_bayes(train, test):
    separated = separate_by_class(train)
    summaries = dict()
    for class_value, rows in separated.items():
        summaries[class_value] = summarize_dataset(rows)

    predictions = list()
    for row in test:
        output = predict(separated, summaries, row)
        predictions.append(output)
    return(predictions)

# Predict the class for a given row
def predict(separated, summaries, row):
    probabilities = calculate_class_probabilities(separated, summaries, row)
    best_label, best_prob = None, -1
    for class_value, probability in probabilities.items():
        if best_label is None or probability > best_prob:
            best_prob = probability
            best_label = class_value
    return best_label

# Calculate the probabilities of predicting each class for a given row
def calculate_class_probabilities(separated, summaries, row):
    total_rows = sum([summaries[label][0][2] for label in summaries])
    probabilities = dict()
    for class_value, class_summaries in summaries.items():
        probabilities[class_value] = summaries[class_value][0][2] /float(total_rows)
        probabilities[class_value] *= calculate_first_column_prob(separated,row,class_value)
        for i in range(len(class_summaries)):
            mean, stdev, _ = class_summaries[i]
            probabilities[class_value] *= calculate_Gaussian_probability(row[i+1], mean, stdev)
    return probabilities

# Test Naive Bayes on Abalone Dataset
seed(1)
file_name = 'abalone_dataset.txt'
dataset = load_txt(file_name)

column_numbers = len(dataset[0])
for i in range(column_numbers-1):
    if(i == 0):
        continue
    convert_str_to_float(dataset, i)

convert_str_to_int_for_output(dataset, column_numbers-1)
convert_str_to_int(dataset,0)

train_numbers = 17
accuracy = evaluate_algorithm(dataset, naive_bayes, train_numbers)

print('Mean Accuracy: %.3f' % accuracy)
```

************************************naive estimator***********************************

```
# Naive Bayes On The Abalone Dataset
from random import randrange
from random import seed

def load_txt(filename):
    dataset = list()
    with open(filename) as txt_file:
        my_list = txt_file.readlines()
        for row in my_list:
            dataset.append(row.split())
    return dataset

# for continuous values, convert string to float
def convert_str_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())

# for last column, string to int '1' -> 1
def convert_str_to_int_for_output(dataset, column):
    for row in dataset:
        row[column] = int(row[column])

def convert_str_to_int(dataset, column):
        class_values = [row[column] for row in dataset]
        unique = set(class_values)
        lookup = dict()
        for i, value in enumerate(unique):
                lookup[value] = i
        for row in dataset:
                row[column] = lookup[row[column]]
        return lookup

def train_test_split(dataset, train_numbers):
    dataset_training = list()
    while len(dataset_training) < train_numbers:
        index = randrange(len(dataset))
        dataset_training.append(dataset.pop(index))
    return dataset_training

# Split the dataset by class values, returns a dictionary
def separate_by_class(train_dataset):
    separated = dict()
    for i in range(len(train_dataset)):
        vector = train_dataset[i]
        class_value = vector[-1]
        if (class_value not in separated):
            separated[class_value] = list()
        separated[class_value].append(vector)
    return separated

def calculate_first_column_prob(separated,row,class_value):
    count=0
    for i in range(len(separated[class_value])):
        if row[0] == separated[class_value][i][0]:
            count += 1
    prob = count/ len(separated[class_value])
    return prob

def accuracy_metric(actual, predicted):
```

```
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    misclassified_instances = len(actual) - correct
    print("%s%s" % ('Test Instances Number  ', len(actual)))
    print("%s%s" % ('Correctly Classified Instances  ', correct))
    print("%s%s" % ('Misclassification Instances  ', misclassified_instances))
    return correct / float(len(actual)) * 100.0

def confusionMatrix(actual, predicted):
        tp1=0
        tp2=0
        tp3=0
        for i in range(len(actual)):
            if actual[i] == predicted[i] == 1:
                tp1 += 1
            if actual[i] == predicted[i] == 2:
                 tp2 += 1
            if actual[i] == predicted[i] == 3:
                 tp3 += 1
        our_confusion_matrix = [[tp1, 0, 0],[0, tp2, 0],[0, 0, tp3]]

        print(our_confusion_matrix)

def mean(numbers):
    return sum(numbers)/float(len(numbers))

# Evaluate an algorithm using a train_test_split
def evaluate_algorithm(dataset, algorithm, train_numbers, *args):
    training_set = train_test_split(dataset, train_numbers)
    test_set = list(dataset)
    dataset.clear()

    predicted = algorithm(training_set, test_set, *args)
    actual = [row[-1] for row in test_set]
    accuracy = accuracy_metric(actual, predicted)
    confusionMatrix(actual, predicted)
    return accuracy

def naive_estimator(train, test):
    separated = separate_by_class(train)


    predictions = list()
    for row in test:
        output = predict(separated, train, row)
        predictions.append(output)
    return(predictions)

# Predict the class for a given row
def predict(separated, train, row):
    probabilities = calculate_class_probabilities(separated, train, row)
    best_label, best_prob = None, -1
    for class_value, probability in probabilities.items():
        if best_label is None or probability > best_prob:
            best_prob = probability
            best_label = class_value
    return best_label
```

```
def calculate_probability(x, value, bin_size):
    w=abs(x-value)/bin_size
    w_result = 0
    if w<1 : w_result =  0.5
    return w_result

# Calculate the probabilities of predicting each class for a given row
def calculate_class_probabilities(separated, train, row):
    total_rows = len(train)
    probabilities = dict()

    for class_value, class_separated in separated.items():
        probabilities[class_value] = len(separated[class_value]) /float(total_rows)
        probabilities[class_value] *= calculate_first_column_prob(separated,row,class_value)

        column_numbers =0
        bin_size = 0
        for column in zip(*class_separated):
            w_result=0
            if column_numbers == 0:
                column_numbers += 1
                continue
            if column_numbers == len(class_separated[0])-1:
                break
            if column_numbers ==1 :
                bin_size = 0.3
            if column_numbers ==2 :
                bin_size = 0.01
            if column_numbers ==3 :
                 bin_size = 0.02
            if column_numbers ==4 :
                bin_size = 0.3
            if column_numbers ==5 :
                bin_size = 0.1
            if column_numbers ==6 :
                bin_size = 0.03
            if column_numbers ==7:
                bin_size = 0.1

            for i in range(len(column)):
                col_value = column[i]
                w_result += calculate_probability(row[column_numbers], col_value, bin_size)
            column_numbers += 1
            prob = (1/ (total_rows*1)*bin_size)*w_result
            probabilities[class_value] *= prob

    return probabilities

# Test Naive Bayes on Abalone Dataset
seed(1)
file_name = 'abalone_dataset.txt'
dataset = load_txt(file_name)

column_numbers = len(dataset[0])
for i in range(column_numbers-1):
    if(i == 0):
        continue
    convert_str_to_float(dataset, i)

convert_str_to_int_for_output(dataset, column_numbers-1)
```

```
convert_str_to_int(dataset,0)

train_numbers = 100
accuracy = evaluate_algorithm(dataset, naive_estimator, train_numbers)

print('Mean Accuracy: %.3f' % accuracy)
```