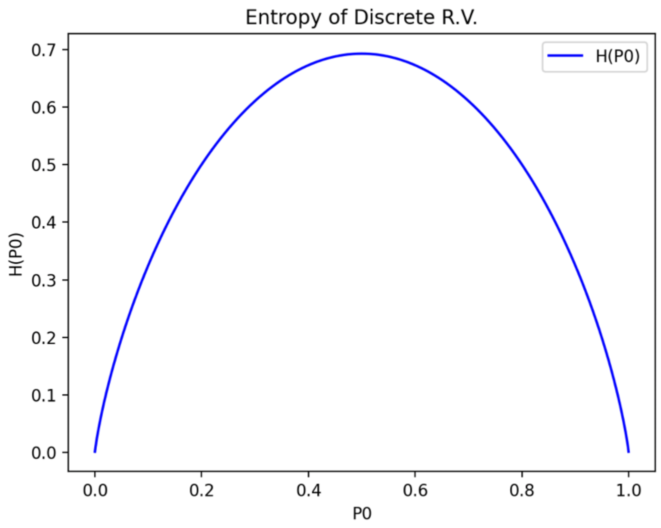
Chapter 1

Part 1:

to compute the entropy of a discrete random variable I implemented the H(X) = $\sum_{K=1}^{Nx} p_k \ln p_k$ formula with a function called "entropy" which takes as input the probability mass function of the random variable and iteratively computes the $p_k \ln p_k$ for each k and add them together and returns the entropy as its output.

Part 2:

To plot the entropy of a binary random variable, first I created 1000 random numbers between 0 and 1 which are probabilities of occurrence of first value in 1000 different cases with different probability. Then I computed the probabilities of occurrence of second value, which is equal to 1 - P_{event1}. Then for each case I passed the PMF of that case [P_{event1},P_{event2}] to the "entropy" function, to compute the entropy of that specific case. Then I plot the data so that the X axis of the plot is the probabilities and the Y axis of the plot is the entropy.



P_{event1}	P_{event2}	H(x)
0.001100	0.9988991	0.001021
0.002101	0.9978983	0.008598
0.003102	0.9968975	0.015055
…	…	
0.997898	0.0021016	0.015055
0.998899	0.9988991	0.008598
0.999999	9.9999e-05	0.001021

As it is shown in the plot, it is a concave function and its maximum is when $P_{event1} = P_{event2}$

Part 3:

to compute the joint entropy of two discrete random variable I implemented the $H(X,Y) \overset{\text{def}}{=} \sum_{i=1}^{Nx} \sum_{j=1}^{Ny} P(x_i, y_j) \log_2 \frac{1}{P(x_i,y_j)}$ formula with a function called "jointEntropy" which takes as input their joint PMF and with a nested loop do the computation for all "i and j"s and sum them together and finally returns the joint entropy as its output.

Part 4:

to compute the joint entropy of two discrete random variable I started with the $H(X|Y) \stackrel{\text{def}}{=} \sum_{i=1}^{Nx}\sum_{j=1}^{Ny} P(x_i, y_j)\log_2\frac{1}{P(x_i|y_j)}$ formula but since the conditional probability $P(x_i|y_j)$ in not given, I replaced it with $P(x_i|y_j) = \frac{P(x_i,y_j)}{P(y_j)}$ which both $P(x_i, y_j)$ and $P(y_j)$ are available. So I implemented the function "conditionalEntropy" with the $H(X|Y) = \sum_{i=1}^{Nx}\sum_{j=1}^{Ny} P(x_i, y_j)\log_2\frac{P(y_j)}{P(x_i,y_j)}$ formula instead, which takes as input joint PMF of two random variables and the marginal PMF of second random variable and with a nested loop do the computation for all "i and j"s and sum them together and finally returns the conditional entropy as its output.

Part 5:

to compute the mutual information of two discrete random variable I implemented the $I(X;Y) \stackrel{\text{def}}{=} \sum_{i=1}^{Nx}\sum_{j=1}^{Ny} P(x_i, y_j)\log_2\frac{P(x_i,y_j)}{P(x_i)P(y_j)}$ formula with a function called "mutualInfo" which takes as input their joint PMF and their marginal PMFs and with a nested loop do the computation for all "i and j"s and sum them together and finally returns the mutual information as its output.

Part 6:

1) to normalize the conditional entropy, I used the $\eta_{CE}(X|Y) = \frac{H(X|Y)}{H(X)}$ formula which normalizes the conditional entropy in base of entropy of first random variable, and I implemented it with a function called "NormalConditionalEntropy" which takes as input joint PMF and marginal PMFs of two random variables. This function first computes the conditional entropy using the "conditionalEntropy" function which I defined it before, then compute the entropy of first random variable using the "entropy" function and divide the conditional Entropy by entropy of first random variable. The result will be the normal conditional entropy which is the output of the function.

2) to normalize the joint entropy, I used the $\eta_{JE}(X,Y) = \frac{H(X,Y)}{H(X)+H(Y)}$ formula which normalizes the joint entropy in base of sum of entropies of random variables, and I implemented it with a function called "NormalJointEntropy" which takes as input joint PMF and marginal PMFs of two random variables. This function first computes the joint entropy using the "jointEntropy" function which I defined it before, then compute the entropy of two random variables using the "entropy" function and sum the entropies together and divide the joint Entropy by the result of summation. The result will be the normal joint entropy which is the output of the function.

3) to normalize the mutual information, I used the $\eta_{MI}(X;Y) = \frac{1}{\eta_{JE}(X,Y)} - 1$ formula which computes the normal mutual information based on normal joint entropy, and I implemented it with a function called "NormalMutualInfo" which takes as input joint

PMF and marginal PMFs of two random variables. This function first computes the normal joint entropy with the “NormalJointEntropy” function and by applying the normalization formula, computes the normal mutual information and returns it.

Chapter 2

Part1:

To Compute the difference between the entropy and the entropy of estimated pdf, first I choose a PMF = [0.4, 0.1, 0.2, 0.3] and passed it as input to the function “entropy” that I defined in Chapter1. This will give the entropy of entropy of a discrete random variable given its PMF vector:

entropy (PMF) = 1.279854

Next I selected the possible values of the random variable as [10, 20, 30, 40] then I generated 1500 random discrete samples based on the chosen PMF and values using “np.random.choice()” function from numpy library:

samples = [10, 40, 20, …, 40, 10, 10]

Then I estimated the PMF of this generated samples using “pmf_estimator” function which uses “np.unique()” function for its estimations. The output of “pmf_estimator” function will be the estimated PMF

PMF_estimated = [0.386, 0.102, 0.192, 0.318]

So, we can pass the estimated PMF as input to “entropy” function.

entropy (PMF_estimated) = 1.295744

Now we have also the entropy of estimated PMF and we can compute their difference.

difference = 0.002109

Part 2:

to compute the differential entropy of a continuous random variable I implemented the $H(X,Y) = -\int_a^b f_X(x) \ln f_X(x) dx$ formula with a function called “differential_entropy” which takes as input its PDF and minimum and maximum of support set (possible values) as its boundary, and compute the integral using integrand function. Integrand function takes as input the PDF and a value of x, and return the probability of that specific x. using this function, the “differential_entropy” function can compute the probability of every single possible value, so in this way it will calculate the integral and return as its output the result of integral, which is the differential entropy.

Part 3:

for this part I chose the mu = 5 and std = 10. Based on these parameters I created the gaussian PDF using “stats.norm” function from scipy library. Now we have the pdf, so we can pass it as input of “differential_entropy” and compute the differential entropy like the way I explained it in part 2.

Differential entropy = 3.710826

Now to estimate this PDF based on a set of samples generated from same PDF, first we need to generate the set of samples. To do this step, I created a “sapmle_generator” function which generates 1500 samples based on given mean and standard derivation using “np.random.normal”.

Continuous samples = [7.868, 15.611, 11.0414, ..., -3.838, 6.705, 20.324]

After that, we need to estimate the PDF of this sample set. Since in the assignment it is written to consider the impact of bandwidth, kernel function and number of samples, I used the kernel density estimator and I implemented this in “kde_estimator” function. Main job of this function is to take a set of samples, estimate the PDF and return the KDE object using “KernelDensity” function from “sklearn.neighbors” library. In this function I considered 2 parameters, first is the kernel function which in this case it is set to gaussian, second is the bandwidth which includes the length of samples and its standard derivation.

Now we just need a function that takes as input, this estimated PDF and a value of the continuous random variable and return as output, the probability of occurrence of that specific value. This job is done by “probability_calculator” function.

So, now we have a function that estimates the PDF and a function that returns the probability of each value of the continuous random variable based on the estimated PDF.

Furthermore, I defined a new function “estimated_differential_entropy” which is like the “differetial_entropy” function, but it takes another input which is the kernel object (estimated pdf).

So, to conclude, to compute the estimated differential entropy, we need to solve the integral, which is done by “estimated_differential_entropy” function. Inside the integral, we need the probability of each x (value of continuous random variable) which is calculated by “probability_calculator” function based on estimated PDF (kde object) using “kde_estimator” function, and “kde_estimator” function estimate the PDF based on sample set that we generated.

Estimated differential entropy = 3.725368

Now we have both differential entropy and estimated differential entropy, so we can compute the difference.

Difference = 0.0145424

Chapter 3

Part 1:

To compute the PMF of Iris dataset, first I loaded the dataset using “datasets.load_iris” function from “sklearn” library. next I divided the data to “data_matrix” and “class_vector”.

Data_matrix				Class_vector
F1	F2	F3	F4	labels
5.1	3.5	1.4	0.2	0
4.9	3.0	1.4	0.2	0
…	…	…	…	0
7.0	3.2	4.7	1.4	1
6.4	3.2	4.5	1.5	1
…	…	…	…	1
6.3	3.3	6.0	2.5	2
5.8	2.7	5.1	1.9	2
…	…	…	…	2

Then we need to discretize the dataset by multiplying values by 10 and then rounding them:

discretized_data_matrix			
F1	F2	F3	F4
51	35	14	2
49	30	14	2
…	…	…	…
70	32	47	14
64	32	45	15
…	…	…	…
63	33	60	25
58	27	51	19

Next, I computed the PMF in two ways: 1) PMF of each feature individually 2) PMF of all features together

1) to compute PMF of each feature, I just divide the data set to 4 columns and then convert them to 1d array using “np.transpose” function

   f1 = [51, 49, 47, …, 65, 62, 59]
   f2 = [35, 30, 32, …, 30, 34, 30]
   f3 = [14, 14, 13, …, 52, 54, 51]
   f4 = [51, 49, 47, …, 65, 62, 59]

   now we can pass these features to the “pmf_estimator” function to calculate their PMF

   unique_values_f1 = [43, 44, 45, 46, …]
   PMF1 = [0.0067, 0.0200, 0.0066, 0.0266, …]
   unique_values_f2 = [20, 22, 23, 24, …]
   PMF2 = [0.0066, 0.0200, 0.0266, 0.0200, …]

```
unique_values_f3 = [10, 11, 12, 13, …]
PMF3 = [0.0066, 0.0066, 0.0133, 0.0466, …]
unique_values_f4 = [1, 2, 3, 4, …]
PMF4 = [0.0333, 0.1933, 0.0466, 0.0466, …]
```

2) To compute the PMF of all features together, we just need to pass the whole dataset to the "pmf_multivariate" function which computes the probability of each row.

```
unique_rows = [[43, 30, 11, 1], [44, 29, 14, 2], [44, 30, 13, 2], …]
PMF_rows = [0.0066, 0.0066, 0.0066, 0.0066, …]
```

Part 2:

To compute the entropy of features we just need to pass the estimated PMFs of features that we estimated in the previous part, as the input of "entropy" function that I defined in chapter 1.

```
Entropy of feature 1 = 3.3423
Entropy of feature 2 = 2.7886
Entropy of feature 3 = 3.4896
Entropy of feature 4 = 2.8071
Entropy of rows= 5.0013
```

Part 3:

To compute mutual information between 2 features, I defined "mutualInfo" function. The input of this function is the whole dataset, and the numbers of columns for which the mutual information should be calculated.

```
Data set = [[43, 30, 11, 1], [44, 29, 14, 2], [44, 30, 13, 2], …]
```

First thing that the function does, is to separate and transpose the wanted features, and create these 3 arrays:

```
feature_samples1 = [51, 49, 47, … ]
feature_samples2 = [35, 30, 32, … ]
selected_data_samples = [[35, 2], [30, 2], [32, 2], … ]
```

which are useful to estimate the marginal PMF of f1 and marginal PMF of f2 and the joint PMF of f1 and f2. To estimate these PMFs we used the "pmf_estimator" function. This "pmf_estimator" function also gives as output the set of unique values of each feature which we use it later.

```
unq_values1 = [43, 44, 45, 46, …] # possible values of feature 1
margpmf1 = [0.0067, 0.0200, 0.0066, 0.0266, …] # marginal PMF of feature 1
unq_values1 = [20, 22, 23, 24, …] # possible values of feature 2
margpmf1 = [0.0066, 0.0200, 0.0266, 0.0200, …] # marginal PMF of feature 2
full_unq_rows = [[43 30], [44 29], [44 30], …] # possible rows of feature 1 and 2
jntpmf = [0.0066, 0.0066, 0.0066, …] # joint PMF of feature 1 and 2
```

Mutual information formula:

$$I(X;Y) \overset{\text{def}}{=} \sum_{i=1}^{Nx} \sum_{j=1}^{Ny} P(x_i, y_j) \log_2 \frac{P(x_i, y_j)}{P(x_i)P(y_j)}$$

I implemented this summation with a nested loop, which each loop will handles one of the 2 SIGMAs. Since the $P(x_i, y_j)$ is 0 for some "i and j"s, to compute the mutual information, it is necessary that in each iteration first check for that specific i and j, the x_i and y_j exists in the "full_unq_rows" or not. If not, it means for these two x_i and y_j, the $P(x_i, y_j)$ is not exists in the joint PDF of f1 and f2, which means its joint probability is equal to 0, so we set the $P(x_i, y_j) = 0$ and jump to the next loop.

Let's make an example to clarify. For example, in first iteration i = 0 and j = 0 which are indexes of "unq_values1" and "unq_values2", so the $x_i = 51$ and $y_j = 35$. To get the value of $P(x_i, y_j)$, we search for the [51, 35] row in the "full_unq_rows". If it was there, we take its index using this line of code:

```
value_index = full_unq_rows.index([51, 35])
```

and we read the $P(51,34)$ just by looking at: jntpmf[value_index]

To conclude, in each loop we calculate $P(x_i, y_j) \log_2 \frac{P(x_i,y_j)}{P(x_i)P(y_j)}$ for all possible values of x_i and y_j (if the $P(x_i, y_j)$ for that pair does not equal to 0) and sum all of these results together. The final result will be the mutual information of these two features which is the output of this function.

So, we call this function for each feature pairs and we get these results:

```
mutual info btw features 1 & 2 = 2.08984
mutual info btw features 2 & 3 = 2.22742
mutual info btw features 3 & 4 = 2.69484
mutual info btw features 1 & 3 = 3.00286
mutual info btw features 1 & 4 = 2.24068
mutual info btw features 2 & 4 = 1.67597
```

chapter 4

Part 1:
To implement the Bayes classifier, I defined the “bayes_classifier” function which its inputs are listed below:

```
train_set example :
        [[5.1 3.5 1.4 0.2]
         [4.9 3.  1.4 0.2]
         [4.7 3.2 1.3 0.2]
         …]
Train_class_lbl example: [0 0 0 … 1 1 1 … 2 2 2 …]
Test_set example :
        [[5.  3.  1.6 0.2]
         [5.  3.4 1.6 0.4]
         [5.2 3.5 1.5 0.2]
         …]
```

First step is to estimate the PMF of class lables and find the unique classes, so we pass the Train_class_lbl to pmf_estimator function. We use these values for maximization of bayes formula.

```
unique_classes=[0,1,2]
class_lbl_pmf=[0.333,0.333,0.333]
```

next we separate the training set based on their classes:

```
tr_sets:
[ [ [5.1, 3.5, 1.4, 0.2], [4.9, 3. , 1.4, 0.2], … ], #first index or first class values
  [ [7. , 3.2, 4.7, 1.4], [6.4, 3.2, 4.5, 1.5], … ], #second index or second class values
  [ [6.3, 3.3, 6. , 2.5], [5.8, 2.7, 5.1, 1.9], … ], #third index or third class values
]
```

Next step is classification and creation of the class label vector of test set based on maximization of bayes formula:

$$P\left(C_j|\underline{X}\right) = \arg max_{C_J} P\left(\underline{X}\middle|C_j\right). P(C_j)$$

To do this we need to calculate the probability of each row in test for all classes. For example, for first row of test set, we pass it to the “kde_estimator_multivar” function. Also this function needs the part of training set which belongs to first class (tr_sets[0]). “kde_estimator_multivar” estimator function first estimate the PDF of first class, then calculate the probability of occurrence of this specific row of test set in the first class and returns this value.

$P\left(\underline{X}\middle|C_0\right)$ = 0.0172

By multiplying this value by the probability of first class which we calculated before in (class_lbl_pmf) we computed the value of $P(\underline{X}|C_0).P(C_0)$ = 0.0057

We do this for all other classes and we select the class (j) that has the maximum value of $P(\underline{X}|C_j).P(C_j)$ as the estimated class label of the first row of test set.

$P(\underline{X}|C_0).P(C_0)$ = 0.0057
$P(\underline{X}|C_1).P(C_1)$ = 0.0001
$P(\underline{X}|C_2).P(C_2)$ = 4.6813e-06
$\arg max_{C_J} P(\underline{X}|C_j).P(C_j)$ = 0.0057 => j = 0

By doing all of these steps for all rows in the test set, we will create the estimated class label of the test set iteratively.

Part 2:

"bayes_naive_classifier" function is very similar to "bayes_classifier" function.
First thing that the "bayes_naive_classifier" function does, is separating the input multivariate training set, to multiple univariate features:

```
tr_sets_univar=
[
        [                               # class0
                [5.1, 4.9, 4.7, … ]     # all values of feature1 in class1
                [3.5, 3. , 3.2, … ]     # all values of feature2 in class1
                [1.4, 1.4, 1.3, … ]     # all values of feature3 in class1
                [0.2, 0.2, 0.2, … ]     # all values of feature4 in class1
        ],
        [                               # class1
                [7. , 6.4, 6.9, … ]     # all values of feature1 in class2
                [3.2, 3.2, 3.1, … ]     # all values of feature2 in class2
                [4.7, 4.5, 4.9, … ]     # all values of feature3 in class2
                [1.4, 1.5, 1.5, … ]     # all values of feature4 in class2
        ],
        [                               # class2
                [6.3, 5.8, 7.1, … ]     # all values of feature1 in class3
                [3.3, 2.7, 3. , … ]     # all values of feature2 in class3
                [6. , 5.1, 5.9, … ]     # all values of feature3 in class3
                [2.5, 1.9, 2.1, … ]     # all values of feature4 in class3
        ]
]
```

Naïve bayes formula :

$$P(\underline{X}|C_j) = \prod_{k=1}^{d} P_{X_k}(X_k|C_j)$$

The computation is like "bayes_classifier" function but here we assume the features are independent and for this reason we use "kde_estimator_univar" function.

Let's clarify the process with an example. For first row of test set, we have 4 features. In first step we pass all values of first feature which have class lable = 0 (means tr_sets_univar[0][0]) and the first value of first feature (means test_set[0][0]) to the "kde_estimator_univar" function. This function estimates the PDF of first feature in class1, then calculates the probability of that specific value in the first class. Now we have $P_{X_1}(X_1|C_0)$. We do this step for all features in the first class. In this way we can calculate $P(\underline{X}|C_0)$ which is $\prod_{k=1}^{4} P_{X_k}\,(X_k|C_0)$. Then we multiply $P(\underline{X}|C_0)$ with $P(C_0)$ which its calculation is explained in Part1 and we achieve $P(C_0|\underline{X})$ which is $P(\underline{X}|C_0).P(C_0)$

Now just like Part 1, we need to do this step for all classes and select the class (j) that has the maximum value of $P(C_j|\underline{X})$ as the estimated class label of the first row of test set. By doing all of these steps for all rows in the test set, we will create the estimated class label of the test set iteratively.

Part 3:

"bayes_naive_classifier_gaussian" function is just like "bayes_naive_classifier" function, except the estimation. Since we assume that the independent features are gaussian distributed, to estimate the PDF, we just need to estimate the mean and the standard derivation of each feature in each class. So, we take the "tr_sets_univar" array which contains the univariate features of each class and then calculate their mean and standard derivation.

```
Mu=
[
        [5.028, 3.48, 1.46, 0.248], # mean of features within class 0
        [6.012, 2.77, 4.312, 1.343], # mean of features within class 1
        [6.575, 2.928, 5.64, 2.044] # mean of features within class 2
]

Std=
[
        [0.392, 0.361, 0.193, 0.102], # std of features within class 0
        [0.536, 0.345, 0.434, 0.202], # std of features within class 0
        [0.709, 0.353, 0.633, 0.249] # std of features within class 0
]
```

Now have mean and standard derivation of each feature. To calculate the probability of each value of each feature in each class, we simply get the mu and std of that feature to the "stats.norm" function from "scipy" library and it creates the gaussian distribution for us. Now we just pass the value of feature to that gaussian distribution and it returns the possibility of occurrence of that value of feature in that specific class which is equal to $P_{X_1}(X_1|C_0)$. From here everything else is just like "bayes_naive_classifier" function which is explained in part 2.

Part 4:

To create the training set and the test set from the Iris data set, I defined the "preparation" function, which first devide the dataset to 3 classes and concatenate first half of all classes and put it in train_set and concatenate second half of all classes and put it in test_set. Also create the train_class_lbl and test_class_lbl in same way which are corresponding class labels.

To compute the accuracy, I defined "accuracy" function which takes the estimated class label of the test set (which is estimate with one of the functions that are explained in part 1-3) and the real class label of the test set as its input, and simply compare the two input arrays. Any difference between these arrays means an error, so it counts the number of errors and divide it to the total length of test set to compute the probability of error. This function should return the accuracy which is equal to 1 - P_{err}.
In the table below we can see the result of estimation by different estimators.

Test Set				Class lbl	Bayes	N Bayes	NG Bayes
5.	3.	1.6	0.2	0	0	0	0
5.	3.4	1.6	0.4	0	0	0	0
5.2	3.5	1.5	0.2	0	0	0	0
5.2	3.4	1.4	0.2	0	0	0	0
4.7	3.2	1.6	0.2	0	0	0	0
4.8	3.1	1.6	0.2	0	0	0	0
5.4	3.4	1.5	0.4	0	0	0	0
5.2	4.1	1.5	0.1	0	0	0	0
5.5	4.2	1.4	0.2	0	0	0	0
4.9	3.1	1.5	0.2	0	0	0	0
5.	3.2	1.2	0.2	0	0	0	0
5.5	3.5	1.3	0.2	0	0	0	0
4.9	3.6	1.4	0.1	0	0	0	0
4.4	3.	1.3	0.2	0	0	0	0
5.1	3.4	1.5	0.2	0	0	0	0
5.	3.5	1.3	0.3	0	0	0	0
4.5	2.3	1.3	0.3	0	0	0	0
4.4	3.2	1.3	0.2	0	0	0	0
5.	3.5	1.6	0.6	0	0	0	0
5.1	3.8	1.9	0.4	0	0	0	0
4.8	3.	1.4	0.3	0	0	0	0
5.1	3.8	1.6	0.2	0	0	0	0
4.6	3.2	1.4	0.2	0	0	0	0
5.3	3.7	1.5	0.2	0	0	0	0
5.	3.3	1.4	0.2	0	0	0	0
6.6	3.	4.4	1.4	1	1	1	1
6.8	2.8	4.8	1.4	1	1	1	1
6.7	3.	5.	1.7	1	1	2	2
6.	2.9	4.5	1.5	1	1	1	1
5.7	2.6	3.5	1.	1	1	1	1
5.5	2.4	3.8	1.1	1	1	1	1
5.5	2.4	3.7	1.	1	1	1	1
5.8	2.7	3.9	1.2	1	1	1	1

6.	2.7	5.1	1.6	1	1	1	1
5.4	3.	4.5	1.5	1	1	1	1
6.	3.4	4.5	1.6	1	1	1	1
6.7	3.1	4.7	1.5	1	1	1	1
6.3	2.3	4.4	1.3	1	1	1	1
5.6	3.	4.1	1.3	1	1	1	1
5.5	2.5	4.	1.3	1	1	1	1
5.5	2.6	4.4	1.2	1	1	1	1
6.1	3.	4.6	1.4	1	1	1	1
5.8	2.6	4.	1.2	1	1	1	1
5.	2.3	3.3	1.	1	1	1	1
5.6	2.7	4.2	1.3	1	1	1	1
5.7	3.	4.2	1.2	1	1	1	1
5.7	2.9	4.2	1.3	1	1	1	1
6.2	2.9	4.3	1.3	1	1	1	1
5.1	2.5	3.	1.1	1	1	1	1
5.7	2.8	4.1	1.3	1	1	1	1
7.2	3.2	6.	1.8	2	2	2	2
6.2	2.8	4.8	1.8	2	1	2	2
6.1	3.	4.9	1.8	2	1	2	2
6.4	2.8	5.6	2.1	2	2	2	2
7.2	3.	5.8	1.6	2	2	2	2
7.4	2.8	6.1	1.9	2	2	2	2
7.9	3.8	6.4	2.	2	2	2	2
6.4	2.8	5.6	2.2	2	2	2	2
6.3	2.8	5.1	1.5	2	1	1	1
6.1	2.6	5.6	1.4	2	2	2	1
7.7	3.	6.1	2.3	2	2	2	2
6.3	3.4	5.6	2.4	2	2	2	2
6.4	3.1	5.5	1.8	2	2	2	2
6.	3.	4.8	1.8	2	1	2	2
6.9	3.1	5.4	2.1	2	2	2	2
6.7	3.1	5.6	2.4	2	2	2	2
6.9	3.1	5.1	2.3	2	2	2	2
5.8	2.7	5.1	1.9	2	1	2	2
6.8	3.2	5.9	2.3	2	2	2	2
6.7	3.3	5.7	2.5	2	2	2	2
6.7	3.	5.2	2.3	2	2	2	2
6.3	2.5	5.	1.9	2	1	2	2
6.5	3.	5.2	2.	2	2	2	2
6.2	3.4	5.4	2.3	2	2	2	2
5.9	3.	5.1	1.8	2	1	2	2

So, the accuracy of each estimator is as follow:

accuracy of general Bayes = 90.67 %
accuracy of Naïve Bayes = 97.33 %
accuracy of Naïve Gaussian Bayes = 96.00 %