# Lab 8 - Naive Bayes

Submitted By
Name: Harsha KG
Register Number: 19112005
Class: **5 BSc Data Science**

## Lab Overview

1. Import the Wine dataset as available in Sklearn Library using the inbuilt load method
2. Illustrate the usage of Naive Bayes Classifier for the above-mentioned dataset
3. Evaluate the results in terms of hyper parameters and justify why we should/should not use Naive Bayes for this particular dataset

### Objectives

Understand the dataset and perform Naive Bayes algorithm and give valid reason for using it in dataset.

### Problem Definition

Understand the Dataset & Features and then perform preprocessing technique and statistical analysis to get insights and then perform Naive Byes algorithm and understand the usage of it in this dataset.

### Approach

Imported the Dataset using Sklearn Library to notebook .Did some pre-processing technique and then build the various models of naive bayes and after that did a accuracy checking by changing paramter values.

### Sections

1. Lab Overview
2. Imported the Required Libraraies
3. Load the Dataset
4. Basic Inference from Data (Data Wrangling)
5. Train-Test Split
6. Using Naive Bayes (Gaussian, Multinomial, Bernouli)
    A. Classification Report
    B. Confusion Matrix
    C. Finding the Accuracy by changing the Parameter
    D. Histogram of Accuracy
    E. HyperParameter Tuning
7. Comparing models
8. List Advantages and Disadvantages of Naive Bayes
9. Conclusion

### References

1. https://pandas.pydata.org/ (https://pandas.pydata.org/)
2. https://matplotlib.org/ (https://matplotlib.org/)
3. https://seaborn.pydata.org/ (https://seaborn.pydata.org/)
4. https://plotly.com/ (https://plotly.com/)
5. https://scikit-learn.org/stable/modules/naive_bayes.html (https://scikit-learn.org/stable/modules/naive_bayes.html)

### About The Dataset

These data are the results of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars.

The analysis determined the quantities of 14 constituents found in each of the three types of wines.

# 1.Import the Wine dataset as available in Sklearn Library using the inbuilt load method

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")
```

In [4]:
```
from sklearn.datasets import load_wine
data = load_wine()
data
```

Out[4]:

```
{'data': array([[1.423e+01, 1.710e+00, 2.430e+00, ..., 1.040e+00, 3.920e+00,
         1.065e+03],
        [1.320e+01, 1.780e+00, 2.140e+00, ..., 1.050e+00, 3.400e+00,
         1.050e+03],
        [1.316e+01, 2.360e+00, 2.670e+00, ..., 1.030e+00, 3.170e+00,
         1.185e+03],
        ...,
        [1.327e+01, 4.280e+00, 2.260e+00, ..., 5.900e-01, 1.560e+00,
         8.350e+02],
        [1.317e+01, 2.590e+00, 2.370e+00, ..., 6.000e-01, 1.620e+00,
         8.400e+02],
        [1.413e+01, 4.100e+00, 2.740e+00, ..., 6.100e-01, 1.600e+00,
         5.600e+02]]),
 'target': array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2]),
 'frame': None,
 'target_names': array(['class_0', 'class_1', 'class_2'], dtype='<U7'),
 'DESCR': '.. _wine_dataset:\n\nWine recognition dataset\n------------------------\n\n*
*Data Set Characteristics:**\n\n    :Number of Instances: 178 (50 in each of three clas
ses)\n    :Number of Attributes: 13 numeric, predictive attributes and the class\n    :
Attribute Information:\n \t\t- Alcohol\n \t\t- Malic acid\n \t\t- Ash\n\t\t- Alcalinity
of ash  \n \t\t- Magnesium\n\t\t- Total phenols\n \t\t- Flavanoids\n \t\t- Nonflavanoid
phenols\n \t\t- Proanthocyanins\n\t\t- Color intensity\n \t\t- Hue\n \t\t- OD280/OD315
of diluted wines\n \t\t- Proline\n\n    - class:\n            - class_0\n            -
class_1\n            - class_2\n\t\t\n    :Summary Statistics:\n    \n    =============
================ ==== ===== ======= =====\n                                Min   Max
Mean     SD\n    ============================ ==== ===== ======= =====\n    Alcohol:
11.0  14.8    13.0   0.8\n    Malic Acid:                  0.74  5.80    2.34  1.12\n
Ash:                         1.36  3.23    2.36  0.27\n    Alcalinity of Ash:
10.6  30.0    19.5   3.3\n    Magnesium:                   70.0 162.0    99.7  14.3\n
Total Phenols:               0.98  3.88    2.29  0.63\n    Flavanoids:
0.34  5.08    2.03  1.00\n    Nonflavanoid Phenols:        0.13  0.66    0.36  0.12\n
Proanthocyanins:             0.41  3.58    1.59  0.57\n    Colour Intensity:
1.3  13.0     5.1   2.3\n    Hue:                        0.48  1.71    0.96  0.23\n
OD280/OD315 of diluted wines: 1.27  4.00    2.61  0.71\n    Proline:
278  1680     746   315\n    ============================ ==== ===== ======= =====\n\n
:Missing Attribute Values: None\n    :Class Distribution: class_0 (59), class_1 (71), c
lass_2 (48)\n    :Creator: R.A. Fisher\n    :Donor: Michael Marshall (MARSHALL%PLU@io.a
rc.nasa.gov)\n    :Date: July, 1988\n\nThis is a copy of UCI ML Wine recognition datase
ts.\nhttps://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data\n\nThe da
ta is the results of a chemical analysis of wines grown in the same\nregion in Italy by
three different cultivators. There are thirteen different\nmeasurements taken for diffe
rent constituents found in the three types of\nwine.\n\nOriginal Owners: \n\nForina, M.
et al, PARVUS - \nAn Extendible Package for Data Exploration, Classification and Correl
ation. \nInstitute of Pharmaceutical and Food Analysis and Technologies,\nVia Brigata S
alerno, 16147 Genoa, Italy.\n\nCitation:\n\nLichman, M. (2013). UCI Machine Learning Re
pository\n[https://archive.ics.uci.edu/ml]. Irvine, CA: University of California,\nScho
ol of Information and Computer Science. \n\n.. topic:: References\n\n  (1) S. Aeberhar
d, D. Coomans and O. de Vel, \n  Comparison of Classifiers in High Dimensional Setting
s, \n  Tech. Rep. no. 92-02, (1992), Dept. of Computer Science and Dept. of  \n  Mathem
atics and Statistics, James Cook University of North Queensland. \n  (Also submitted to
Technometrics). \n\n  The data was used with many others for comparing various \n  clas
sifiers. The classes are separable, though only RDA \n  has achieved 100% correct class
ification. \n  (RDA : 100%, QDA 99.4%, LDA 98.9%, 1NN 96.1% (z-transformed data)) \n
(All results using the leave-one-out technique) \n\n  (2) S. Aeberhard, D. Coomans and
O. de Vel, \n  "THE CLASSIFICATION PERFORMANCE OF RDA" \n  Tech. Rep. no. 92-01, (199
2), Dept. of Computer Science and Dept. of \n  Mathematics and Statistics, James Cook U
niversity of North Queensland. \n  (Also submitted to Journal of Chemometrics).\n',
 'feature_names': ['alcohol',
  'malic_acid',
  'ash',
  'alcalinity_of_ash',
```

```
 'magnesium',
 'total_phenols',
 'flavanoids',
 'nonflavanoid_phenols',
 'proanthocyanins',
 'color_intensity',
 'hue',
 'od280/od315_of_diluted_wines',
 'proline']}
```

In [6]:
```
df = pd.DataFrame(data.data, columns = data.feature_names)
df['target'] = pd.Series(data.target)
```

# Basic Inference from Data (Data Wrangling)

In [7]:
```
df.shape
```

Out[7]:
```
(178, 14)
```

In [8]:
```
df.columns
```

Out[8]:
```
Index(['alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash', 'magnesium',
       'total_phenols', 'flavanoids', 'nonflavanoid_phenols',
       'proanthocyanins', 'color_intensity', 'hue',
       'od280/od315_of_diluted_wines', 'proline', 'target'],
      dtype='object')
```

In [9]:
```
df.head()
```

Out[9]:

| | alcohol | malic_acid | ash | alcalinity_of_ash | magnesium | total_phenols | flavanoids | nonflavanoid_phenols |
|---|---|---|---|---|---|---|---|---|
| **0** | 14.23 | 1.71 | 2.43 | 15.6 | 127.0 | 2.80 | 3.06 | 0.28 |
| **1** | 13.20 | 1.78 | 2.14 | 11.2 | 100.0 | 2.65 | 2.76 | 0.26 |
| **2** | 13.16 | 2.36 | 2.67 | 18.6 | 101.0 | 2.80 | 3.24 | 0.30 |
| **3** | 14.37 | 1.95 | 2.50 | 16.8 | 113.0 | 3.85 | 3.49 | 0.24 |
| **4** | 13.24 | 2.59 | 2.87 | 21.0 | 118.0 | 2.80 | 2.69 | 0.39 |

In [10]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 178 entries, 0 to 177
Data columns (total 14 columns):
 #   Column                        Non-Null Count  Dtype  
---  ------                        --------------  -----  
 0   alcohol                       178 non-null    float64
 1   malic_acid                    178 non-null    float64
 2   ash                           178 non-null    float64
 3   alcalinity_of_ash             178 non-null    float64
 4   magnesium                     178 non-null    float64
 5   total_phenols                 178 non-null    float64
 6   flavanoids                    178 non-null    float64
 7   nonflavanoid_phenols          178 non-null    float64
 8   proanthocyanins               178 non-null    float64
 9   color_intensity               178 non-null    float64
 10  hue                           178 non-null    float64
 11  od280/od315_of_diluted_wines  178 non-null    float64
 12  proline                       178 non-null    float64
 13  target                        178 non-null    int32  
dtypes: float64(13), int32(1)
memory usage: 18.9 KB
```

In [11]:
```
df.isna().sum()
df.isnull().sum()
```

Out[11]:
```
alcohol                         0
malic_acid                      0
ash                             0
alcalinity_of_ash               0
magnesium                       0
total_phenols                   0
flavanoids                      0
nonflavanoid_phenols            0
proanthocyanins                 0
color_intensity                 0
hue                             0
od280/od315_of_diluted_wines    0
proline                         0
target                          0
dtype: int64
```

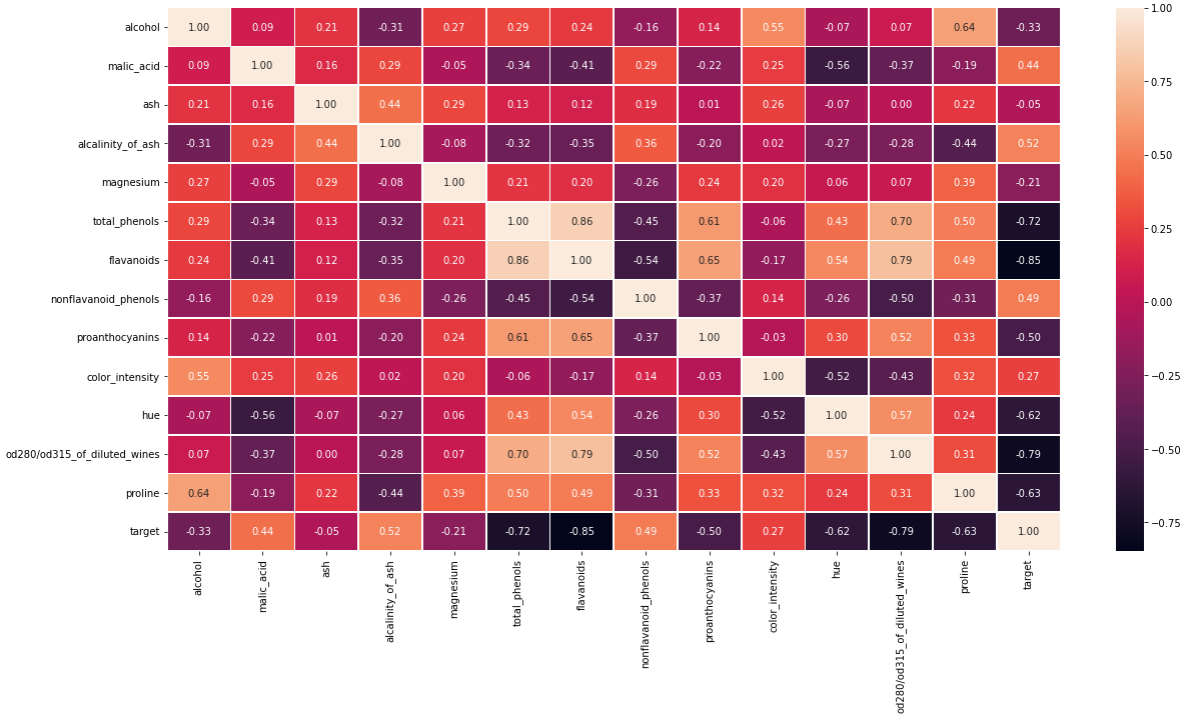
In [12]:
```
df.describe()
```

Out[12]:

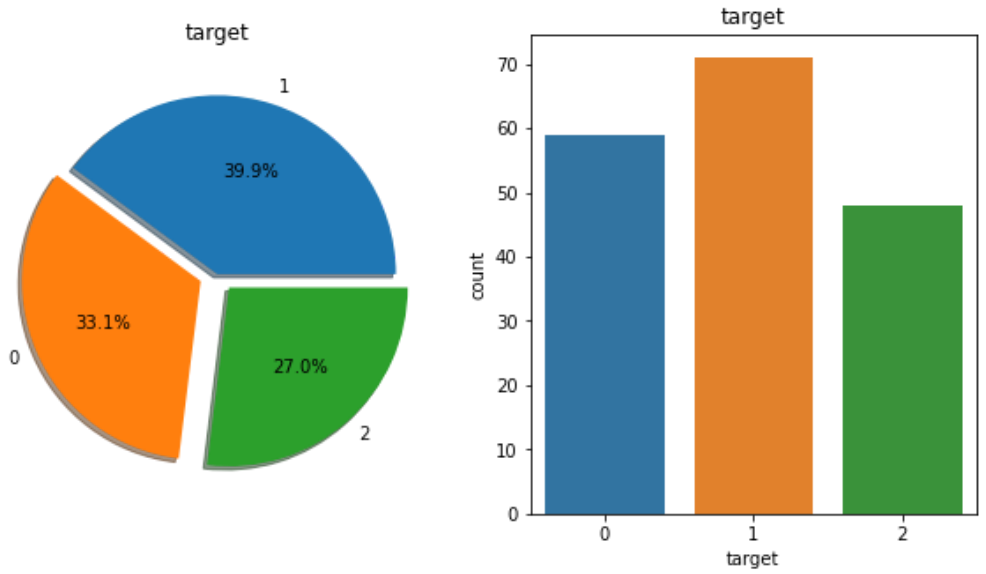
| | alcohol | malic_acid | ash | alcalinity_of_ash | magnesium | total_phenols | flavanoids | nonflava |
|---|---|---|---|---|---|---|---|---|
| **count** | 178.000000 | 178.000000 | 178.000000 | 178.000000 | 178.000000 | 178.000000 | 178.000000 | |
| **mean** | 13.000618 | 2.336348 | 2.366517 | 19.494944 | 99.741573 | 2.295112 | 2.029270 | |
| **std** | 0.811827 | 1.117146 | 0.274344 | 3.339564 | 14.282484 | 0.625851 | 0.998859 | |
| **min** | 11.030000 | 0.740000 | 1.360000 | 10.600000 | 70.000000 | 0.980000 | 0.340000 | |
| **25%** | 12.362500 | 1.602500 | 2.210000 | 17.200000 | 88.000000 | 1.742500 | 1.205000 | |
| **50%** | 13.050000 | 1.865000 | 2.360000 | 19.500000 | 98.000000 | 2.355000 | 2.135000 | |
| **75%** | 13.677500 | 3.082500 | 2.557500 | 21.500000 | 107.000000 | 2.800000 | 2.875000 | |
| **max** | 14.830000 | 5.800000 | 3.230000 | 30.000000 | 162.000000 | 3.880000 | 5.080000 | |

In [67]:
```
plt.figure(figsize=(20,10))
sns.heatmap(df.corr(),annot=True, fmt=".2f",annot_kws={"size":10},linewidths=.7)
```

Out[67]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x20e1c656a60>
```

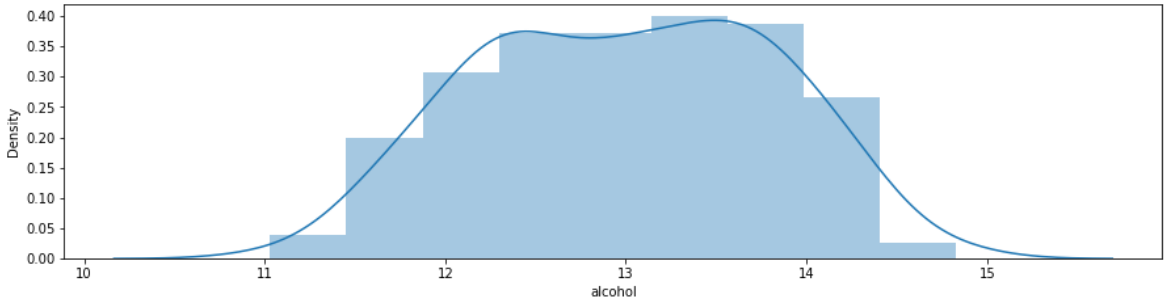


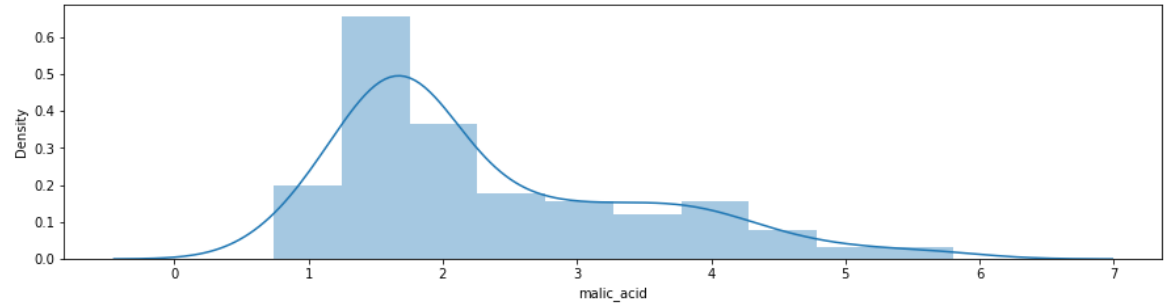
In [69]:
```
f,ax=plt.subplots(1,2,figsize=(10,5))
df['target'].value_counts().plot.pie(explode=[0,0.1,0.1],autopct='%1.1f%%',ax=ax[0],shadow=True)
ax[0].set_title('target')
ax[0].set_ylabel('')
sns.countplot('target',data=df,ax=ax[1])
ax[1].set_title('target')
plt.show()
```

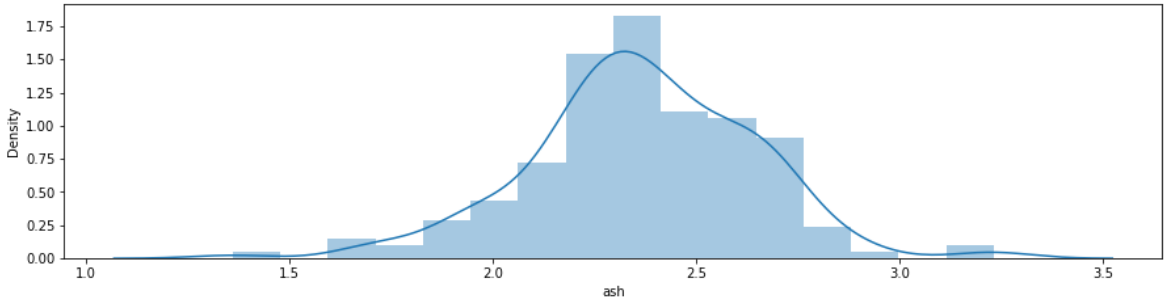


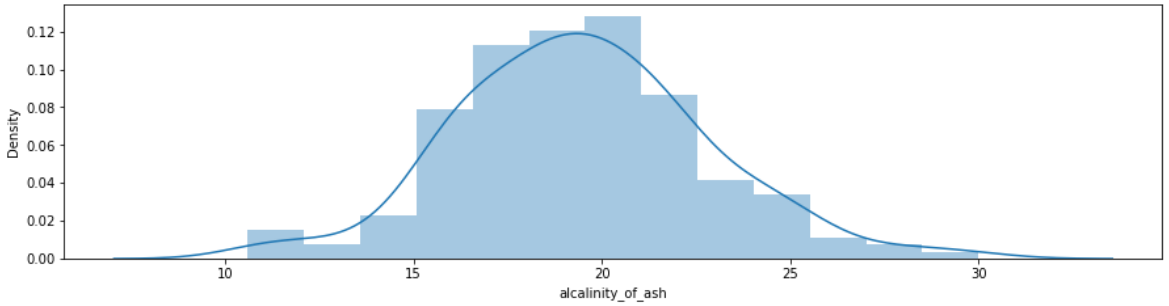
In [93]:
```
columns = list(df.columns)
columns.remove('target')
fig, ax = plt.subplots(nrows = 13, figsize=(15,55))
fig.suptitle("Distribution Plot")
for index, column in enumerate(columns):
    sns.distplot(df[column], ax = ax[index])

plt.show()
```

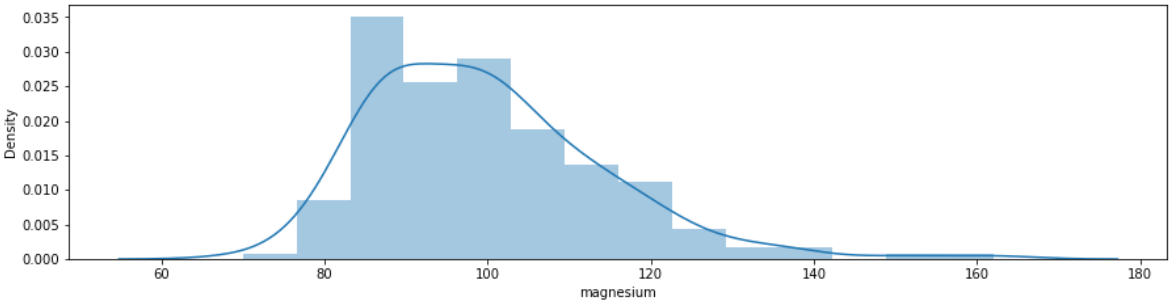
Distribution Plot

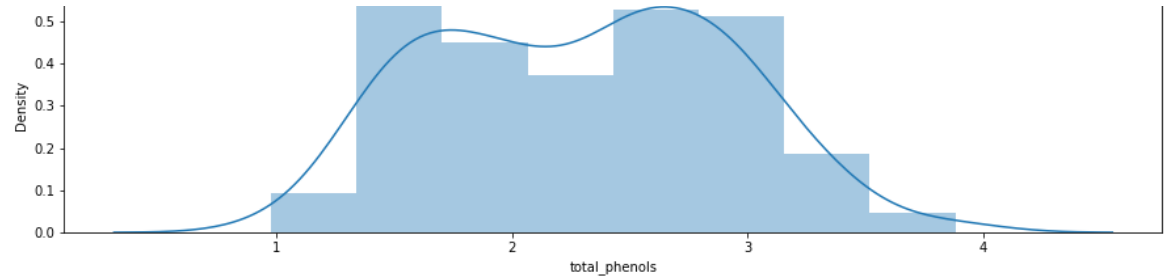


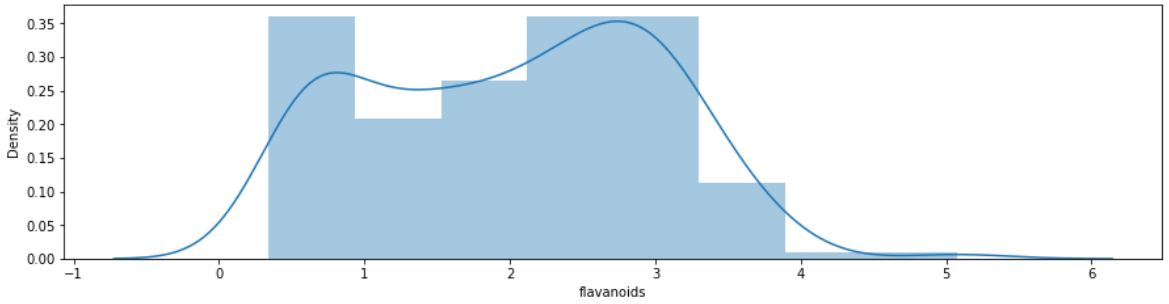



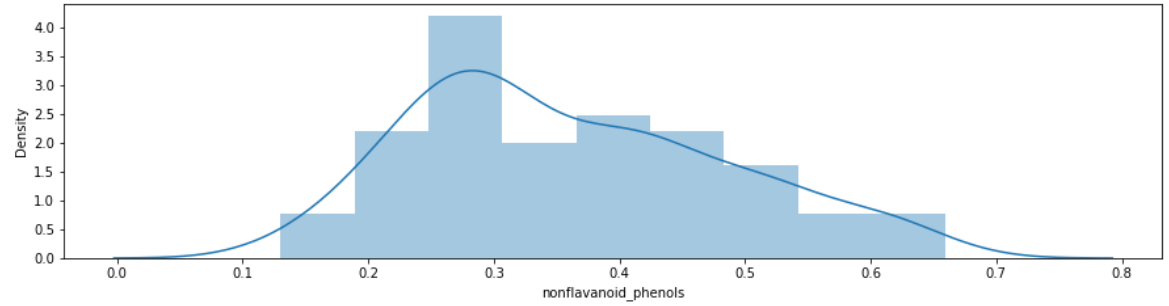



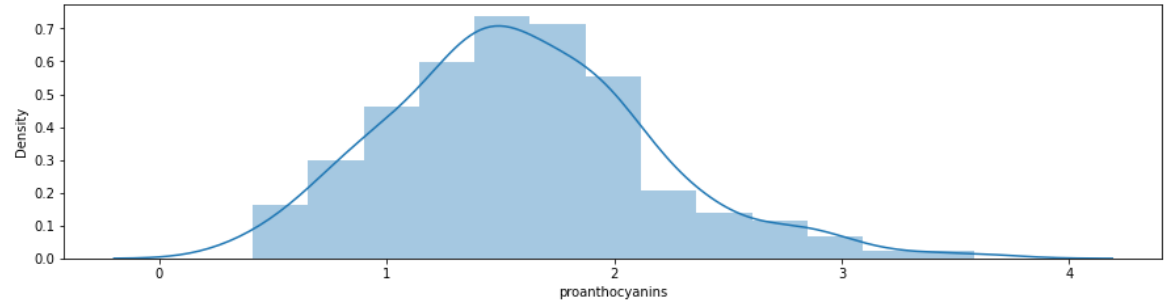



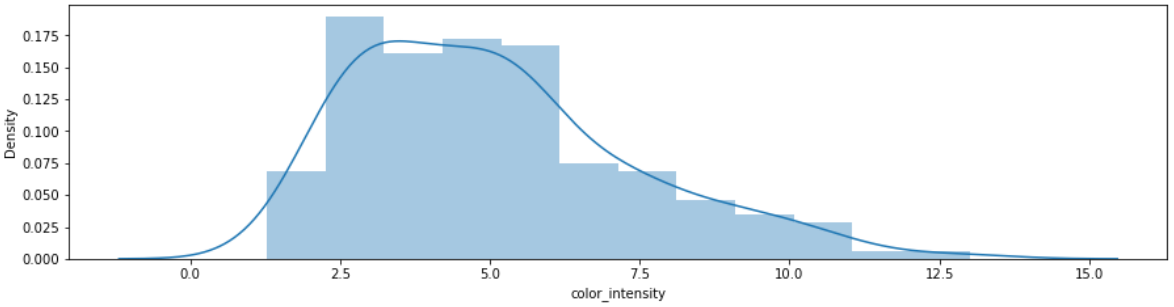



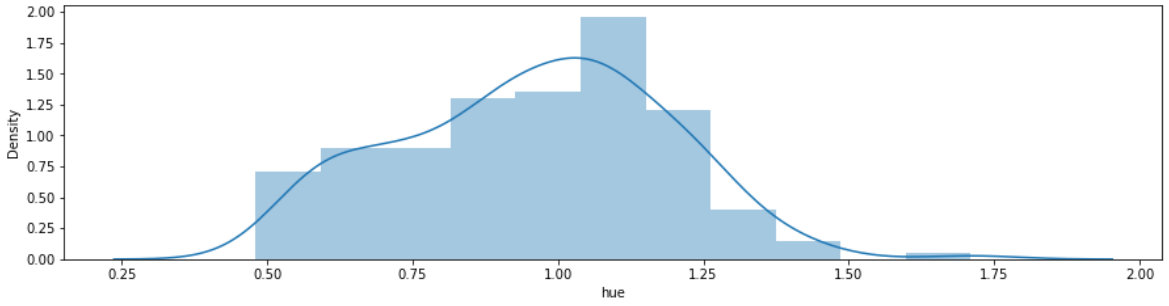

Density
total_phenols

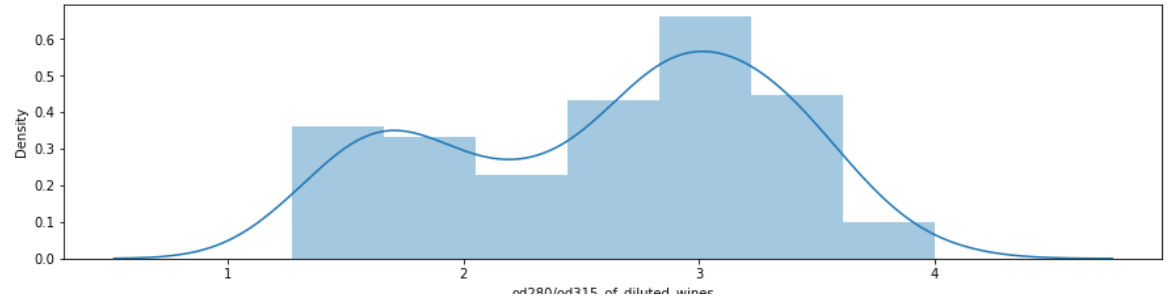

Density
flavanoids

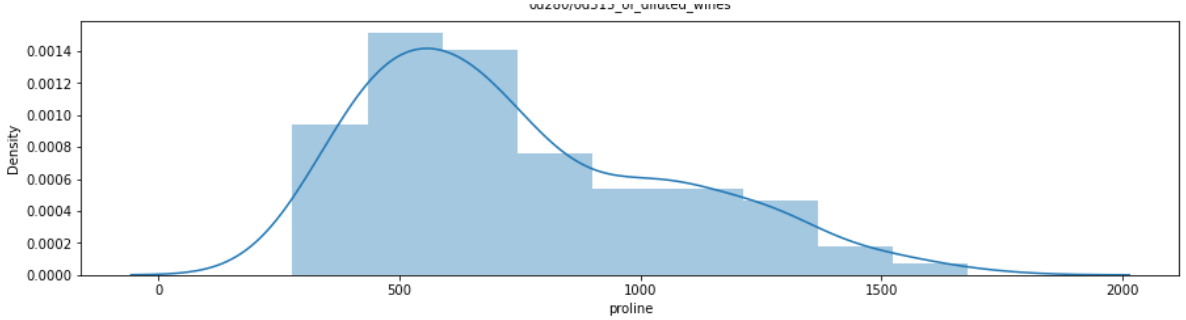

Density
nonflavanoid_phenols


Density
proanthocyanins


Density
color_intensity


Density
hue


Density




# Split into Dependent and Independent Variable

In [13]:
```
Y = df['target']
X = df.drop(['target'], axis=1)
```

# Train-Test Split

In [14]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.30, random_state
= 14)
```

# Naive Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. Bayes' theorem states the following relationship, given class variable $y$ and dependent feature vector $x_1$ through $x_{n,}$ :

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) P(x_1, \dots, x_n \mid y)}{P(x_1, \dots, x_n)}$$

Using the naive conditional independence assumption that

$$P(x_i | y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i | y),$$

for all $i$, this relationship is simplified to

$$P(y \mid x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \dots, x_n)}$$

Since $P(x_1, \dots, x_n)$ is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, \dots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

$$\Downarrow$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

and we can use Maximum A Posteriori (MAP) estimation to estimate $P(y)$ and $P(x_i \mid y)$; the former is then the relative frequency of class $y$ in the training set.

The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(x_i \mid y)$.

In spite of their apparently over-simplified assumptions, naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering. They require a small amount of training data to estimate the necessary parameters. (For theoretical reasons why naive Bayes works well, and on which types of data it does, see the references below.)

Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods. The decoupling of the class conditional feature distributions means that each distribution can be independently estimated as a one dimensional distribution. This in turn helps to alleviate problems stemming from the curse of dimensionality.

On the flip side, although naive Bayes is known as a decent classifier, it is known to be a bad estimator, so the probability outputs from `predict_proba` are not to be taken too seriously.

# 1.1 Gaussian Naive Bayes

`GaussianNB` implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

The parameters $\sigma_y$ and $\mu_y$ are estimated using maximum likelihood.

# Fiting and Prediction

In [16]:
```
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import GaussianNB
NB = GaussianNB()
NB.fit(X_train,Y_train)
NB_predict = NB.predict(X_test)
accuracy_score(NB_predict,Y_test)
```

Out[16]:
```
1.0
```

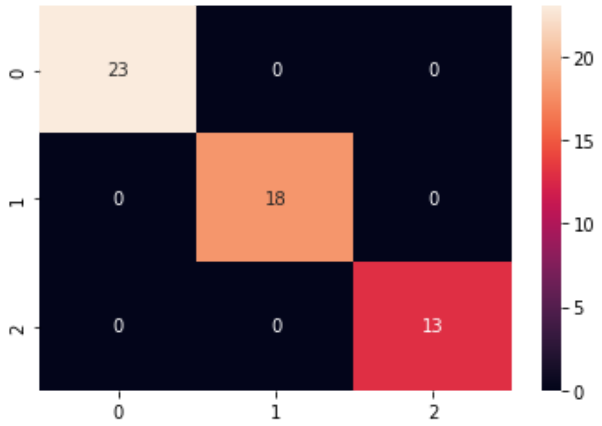
In [17]:
```
NB_predict
```

Out[17]:
```
array([0, 2, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 2, 2,
       1, 2, 1, 2, 2, 1, 0, 1, 0, 1, 0, 1, 0, 0, 2, 1, 2, 1, 2, 2, 0, 2,
       0, 0, 0, 0, 2, 2, 0, 0, 0, 0])
```

# Confusion Matrix

In [73]:
```
from sklearn.metrics import confusion_matrix
conf_matrix = confusion_matrix(Y_test, NB_predict)
dataframe_conf_matrix = conf_matrix
sns.heatmap(dataframe_conf_matrix, annot=True)
```

Out[73]: <matplotlib.axes._subplots.AxesSubplot at 0x20e1d5cf3a0>




# Classification Report

In [74]:
```
from sklearn.metrics import classification_report
print(classification_report(Y_test, NB_predict))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        23
           1       1.00      1.00      1.00        18
           2       1.00      1.00      1.00        13

    accuracy                           1.00        54
   macro avg       1.00      1.00      1.00        54
weighted avg       1.00      1.00      1.00        54
```

In [77]:
```
from sklearn.metrics import confusion_matrix
Cm = confusion_matrix(Y_test,NB_predict)
Cm
```

Out[77]:
```
array([[23,  0,  0],
       [ 0, 18,  0],
       [ 0,  0, 13]], dtype=int64)
```

In [91]:
```
Accuracy = (Cm[0][0] + Cm[1][1]) / (Cm[0][0] + Cm[1][1] + Cm[0][1] + Cm[1][0])
print("Accuracy",Accuracy)
Error_rate = (Cm[0][1] + Cm[1][0]) / (Cm[0][0] + Cm[1][1] + Cm[0][1] + Cm[1][0])
print("Error_rate",Error_rate)
Sensitivity = Cm[0][0]/(Cm[0][0] + Cm[1][0])
print("Sensitivity",Sensitivity)
Specificity = Cm[1][1]/(Cm[1][1] + Cm[0][1])
print("Specificity",Specificity)
Recall = Cm[0][0]/(Cm[0][0] + Cm[1][0])
print("Recall",Recall)
Precision = Cm[0][0]/(Cm[0][0] + Cm[0][1])
print("Precision",Precision)
F1Score = (2*(Precision*Recall))/(Precision + Recall)
print("F1Score",F1Score)
```

```
Accuracy 0.9024390243902439
Error_rate 0.0975609756097561
Sensitivity 0.9523809523809523
Specificity 0.85
Recall 0.9523809523809523
Precision 0.8695652173913043
F1Score 0.909090909090909
```

# Parameter change and accuracy Check

In [20]:
```
def doGaussianNBClassifier(X, Y, test_size = 0.30, randomstate = None):
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = test_size, ran
dom_state = randomstate)
    cls1 = GaussianNB(priors = None, var_smoothing = 1e-09)
    cls1.fit(X_train,Y_train)
    predA = cls1.predict(X_test)
    acc_score = accuracy_score(predA, Y_test)
    return acc_score
```

In [21]:
```
test_size = [0.30, 0.25, 0.20, 0.10]
random_states = [14, 21, 42, 84]
```

In [22]:
```
df_GB = pd.DataFrame(columns = ['Test_Size','Random_States','GaussianNB_Accuracy'])
```

In [24]:
```
for t_size in test_size:
    for r_state in random_states:

        Algo1 = doGaussianNBClassifier(X, Y, t_size, r_state)
        NB_G = {}
        NB_G['Test_Size'] = t_size
        NB_G['Random_States'] = r_state
        NB_G['GaussianNB_Accuracy'] = Algo1
        df_GB = df_GB.append(NB_G, ignore_index = True)
```

In [25]:
```
df_GB
```

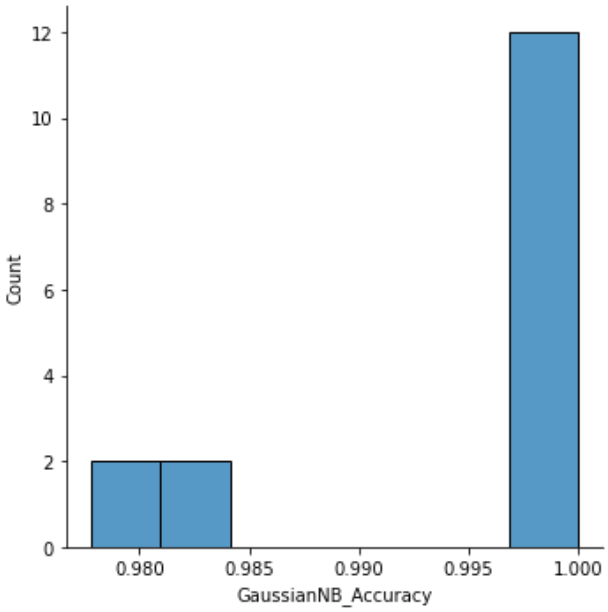
Out[25]:

| | Test_Size | Random_States | GaussianNB_Accuracy |
|---|---|---|---|
| **0** | 0.30 | 14.0 | 1.000000 |
| **1** | 0.30 | 21.0 | 0.981481 |
| **2** | 0.30 | 42.0 | 1.000000 |
| **3** | 0.30 | 84.0 | 0.981481 |
| **4** | 0.25 | 14.0 | 1.000000 |
| **5** | 0.25 | 21.0 | 0.977778 |
| **6** | 0.25 | 42.0 | 1.000000 |
| **7** | 0.25 | 84.0 | 0.977778 |
| **8** | 0.20 | 14.0 | 1.000000 |
| **9** | 0.20 | 21.0 | 1.000000 |
| **10** | 0.20 | 42.0 | 1.000000 |
| **11** | 0.20 | 84.0 | 1.000000 |
| **12** | 0.10 | 14.0 | 1.000000 |
| **13** | 0.10 | 21.0 | 1.000000 |
| **14** | 0.10 | 42.0 | 1.000000 |
| **15** | 0.10 | 84.0 | 1.000000 |

# Histogram Plot of Accuracy

In [63]:
```
sns.displot(x = 'GaussianNB_Accuracy', data =df_GB)
```

Out[63]:
```
<seaborn.axisgrid.FacetGrid at 0x20e1c489f40>
```




# BernoulliNB

`BernoulliNB` implements the naive Bayes training and classification algorithms for data that is distributed according to multivariate Bernoulli distributions; i.e., there may be multiple features but each one is assumed to be a binary-valued (Bernoulli, boolean) variable. Therefore, this class requires samples to be represented as binary-valued feature vectors; if handed any other kind of data, a `BernoulliNB` instance may binarize its input (depending on the `binarize` parameter).

The decision rule for Bernoulli naive Bayes is based on

$$P(x_i \mid y) = P(i \mid y)x_i + (1 - P(i \mid y))(1 - x_i)$$

which differs from multinomial NB's rule in that it explicitly penalizes the non-occurrence of a feature $i$ that is an indicator for class $y$, where the multinomial variant would simply ignore a non-occurring feature.

In the case of text classification, word occurrence vectors (rather than word count vectors) may be used to train and use this classifier. `BernoulliNB` might perform better on some datasets, especially those with shorter documents. It is advisable to evaluate both models, if time permits.

# Fiting and Prediction

In [26]:
```
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import BernoulliNB
NB_B = BernoulliNB()
NB_B.fit(X_train,Y_train)
NB_B_predict = NB_B.predict(X_test)
accuracy_score(NB_B_predict,Y_test)
```

Out[26]:
```
0.3333333333333333
```

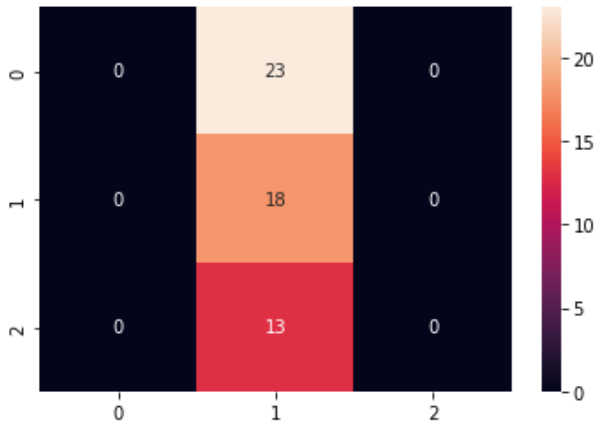
In [27]:
```
NB_predict
```

Out[27]:
```
array([0, 2, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 2, 2,
       1, 2, 1, 2, 2, 1, 0, 1, 0, 1, 0, 1, 0, 0, 2, 1, 2, 1, 2, 2, 0, 2,
       0, 0, 0, 0, 2, 2, 0, 0, 0, 0])
```

# Confusion Matrix

In [80]:
```
from sklearn.metrics import confusion_matrix
conf_matrix = confusion_matrix(Y_test, NB_B_predict)
dataframe_conf_matrix = conf_matrix
sns.heatmap(dataframe_conf_matrix, annot=True)
```

Out[80]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x20e1d8b3e20>
```



In [83]:
```
from sklearn.metrics import confusion_matrix
Cm = confusion_matrix(Y_test,NB_B_predict)
Cm
```

Out[83]:
```
array([[ 0, 23,  0],
       [ 0, 18,  0],
       [ 0, 13,  0]], dtype=int64)
```

# Classification Report

In [28]:
```
from sklearn.metrics import classification_report
print(classification_report(Y_test, NB_B_predict))
```

```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        23
           1       0.33      1.00      0.50        18
           2       0.00      0.00      0.00        13

    accuracy                           0.33        54
   macro avg       0.11      0.33      0.17        54
weighted avg       0.11      0.33      0.17        54
```

In [85]:
```
Accuracy = (Cm[0][0] + Cm[1][1]) / (Cm[0][0] + Cm[1][1] + Cm[0][1] + Cm[1][0])
print("Accuracy",Accuracy)
Error_rate = (Cm[0][1] + Cm[1][0]) / (Cm[0][0] + Cm[1][1] + Cm[0][1] + Cm[1][0])
print("Error_rate",Error_rate)
Sensitivity = Cm[0][0]/(Cm[0][0] + Cm[1][0])
print("Sensitivity",Sensitivity)
Specificity = Cm[1][1]/(Cm[1][1] + Cm[0][1])
print("Specificity",Specificity)
Recall = Cm[0][0]/(Cm[0][0] + Cm[1][0])
print("Recall",Recall)
Precision = Cm[0][0]/(Cm[0][0] + Cm[0][1])
print("Precision",Precision)
F1Score = (2*(Precision*Recall))/(Precision + Recall)
print("F1Score",F1Score)
```

```
Accuracy 0.43902439024390244
Error_rate 0.5609756097560976
Sensitivity nan
Specificity 0.43902439024390244
Recall nan
Precision 0.0
F1Score nan
```

# Parameter change and accuracy Check

In [31]:
```
def doBernoulliNBClassifier(X, Y, test_size = 0.30, randomstate = None):
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = test_size, ran
dom_state = randomstate)
    cls2 = BernoulliNB(alpha = 1.0, binarize = 0.0, fit_prior = True, class_prior = None
)
    cls2.fit(X_train,Y_train)
    predB = cls2.predict(X_test)
    acc_score = accuracy_score(predB, Y_test)
    return acc_score
```

In [32]:
```
test_size = [0.30, 0.45, 0.20, 0.50]
random_states = [14, 30, 42, 80]
```

In [33]:
```
df_BB = pd.DataFrame(columns = ['Test_Size','Random_States','BernoulliNB_Accuracy'])
```

In [35]:
```
for t_size in test_size:
    for r_state in random_states:

        Algo2 = doBernoulliNBClassifier(X, Y, t_size, r_state)
        NB_B = {}
        NB_B['Test_Size'] = t_size
        NB_B['Random_States'] = r_state
        NB_B['BernoulliNB_Accuracy'] = Algo2
        df_BB = df_BB.append(NB_B, ignore_index = True)
```

In [36]:
```
df_BB
```

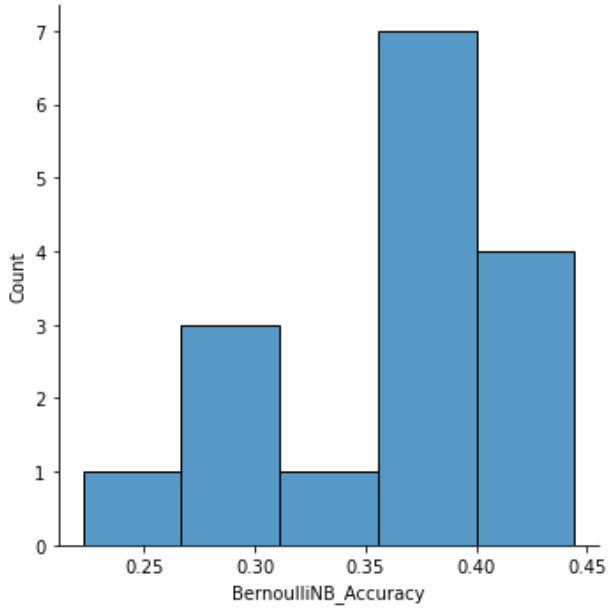
Out[36]:

| | Test_Size | Random_States | BernoulliNB_Accuracy |
|---|---|---|---|
| **0** | 0.30 | 14.0 | 0.333333 |
| **1** | 0.30 | 30.0 | 0.425926 |
| **2** | 0.30 | 42.0 | 0.388889 |
| **3** | 0.30 | 80.0 | 0.277778 |
| **4** | 0.45 | 14.0 | 0.370370 |
| **5** | 0.45 | 30.0 | 0.283951 |
| **6** | 0.45 | 42.0 | 0.382716 |
| **7** | 0.45 | 80.0 | 0.382716 |
| **8** | 0.20 | 14.0 | 0.444444 |
| **9** | 0.20 | 30.0 | 0.444444 |
| **10** | 0.20 | 42.0 | 0.388889 |
| **11** | 0.20 | 80.0 | 0.222222 |
| **12** | 0.50 | 14.0 | 0.393258 |
| **13** | 0.50 | 30.0 | 0.280899 |
| **14** | 0.50 | 42.0 | 0.382022 |
| **15** | 0.50 | 80.0 | 0.404494 |

# Histogram Plot of Accuracy

In [61]:
```
sns.displot(x = 'BernoulliNB_Accuracy', data =df_BB)
```

Out[61]:
```
<seaborn.axisgrid.FacetGrid at 0x20e1c39fd30>
```




# MultinomialNB

`MultinomialNB` implements the naive Bayes algorithm for multinomially distributed data, and is one of the two classic naive Bayes variants used in text classification (where the data are typically represented as word vector counts, although tf-idf vectors are also known to work well in practice). The distribution is parametrized by vectors $\theta_y = (\theta_{y1}, \ldots, \theta_{yn})$ for each class $y$, where $n$ is the number of features (in text classification, the size of the vocabulary) and $\theta_{yi}$ is the probability $P(x_i \mid y)$ of feature $i$ appearing in a sample belonging to class $y$.

The parameters $\theta_y$ is estimated by a smoothed version of maximum likelihood, i.e. relative frequency counting:

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n}$$

where $N_{yi} = \sum_{x \in T} x_i$ is the number of times feature $i$ appears in a sample of class $y$ in the training set $T$, and $N_y = \sum_{i=1}^{n} N_{yi}$ is the total count of all features for class $y$.

The smoothing priors $\alpha \geq 0$ accounts for features not present in the learning samples and prevents zero probabilities in further computations. Setting $\alpha = 1$ is called Laplace smoothing, while $\alpha < 1$ is called Lidstone smoothing.

# Fiting and Prediction

In [38]:
```
from sklearn.metrics import accuracy_score
from sklearn.naive_bayes import MultinomialNB
NB_M = MultinomialNB()
NB_M.fit(X_train,Y_train)
NB_M_predict = NB_M.predict(X_test)
accuracy_score(NB_M_predict,Y_test)
```

Out[38]:
```
0.8518518518518519
```

In [39]:
```
NB_M_predict
```

Out[39]:
```
array([0, 2, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 2,
       1, 2, 1, 2, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 2, 1, 2, 0, 0, 2,
       1, 0, 0, 0, 2, 2, 0, 0, 0, 1])
```

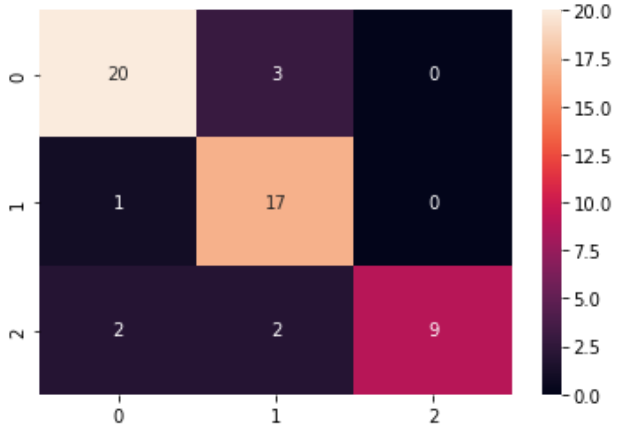
# Confusion Matrix

In [86]:
```
from sklearn.metrics import confusion_matrix
Cm = confusion_matrix(Y_test,NB_M_predict)
Cm
```

Out[86]:
```
array([[20,  3,  0],
       [ 1, 17,  0],
       [ 2,  2,  9]], dtype=int64)
```

In [89]:
```
from sklearn.metrics import confusion_matrix
conf_matrix = confusion_matrix(Y_test, NB_M_predict)
dataframe_conf_matrix = conf_matrix
sns.heatmap(dataframe_conf_matrix, annot=True)
```

Out[89]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x20e1c562310>
```




# Classification report

In [88]:
```
from sklearn.metrics import classification_report
print(classification_report(Y_test, NB_M_predict))
```

```
              precision    recall  f1-score   support

           0       0.87      0.87      0.87        23
           1       0.77      0.94      0.85        18
           2       1.00      0.69      0.82        13

    accuracy                           0.85        54
   macro avg       0.88      0.84      0.85        54
weighted avg       0.87      0.85      0.85        54
```

In [90]:
```
Accuracy = (Cm[0][0] + Cm[1][1]) / (Cm[0][0] + Cm[1][1] + Cm[0][1] + Cm[1][0])
print("Accuracy",Accuracy)
Error_rate = (Cm[0][1] + Cm[1][0]) / (Cm[0][0] + Cm[1][1] + Cm[0][1] + Cm[1][0])
print("Error_rate",Error_rate)
Sensitivity = Cm[0][0]/(Cm[0][0] + Cm[1][0])
print("Sensitivity",Sensitivity)
Specificity = Cm[1][1]/(Cm[1][1] + Cm[0][1])
print("Specificity",Specificity)
Recall = Cm[0][0]/(Cm[0][0] + Cm[1][0])
print("Recall",Recall)
Precision = Cm[0][0]/(Cm[0][0] + Cm[0][1])
print("Precision",Precision)
F1Score = (2*(Precision*Recall))/(Precision + Recall)
print("F1Score",F1Score)
```

```
Accuracy 0.9024390243902439
Error_rate 0.0975609756097561
Sensitivity 0.9523809523809523
Specificity 0.85
Recall 0.9523809523809523
Precision 0.8695652173913043
F1Score 0.909090909090909
```

# Parameter change and accuracy Check

In [42]:
```
def doMultinomialNBClassifier(X, Y, test_size = 0.30, randomstate = None):
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = test_size, ran
dom_state = randomstate)
    cls3 = MultinomialNB( alpha=1.0, fit_prior = True, class_prior = None)
    cls3.fit(X_train,Y_train)
    predC = cls3.predict(X_test)
    acc_score = accuracy_score(predC, Y_test)
    return acc_score
```

In [43]:
```
test_size = [0.30, 0.67, 0.90, 0.11]
random_states = [14, 31, 42, 64]
```

In [44]:
```
df_M = pd.DataFrame(columns = ['Test_Size','Random_States','MultinomialNB_Accuracy'])
```

In [49]:
```
for t_size in test_size:
    for r_state in random_states:
        Algo3 = doMultinomialNBClassifier(X, Y, t_size, r_state)
        NB_M= {}
        NB_M['Test_Size'] = t_size
        NB_M['Random_States'] = r_state
        NB_M['MultinomialNB_Accuracy'] = Algo3
        df_M= df_M.append(NB_M, ignore_index = True)
```

In [50]:
```
df_M
```

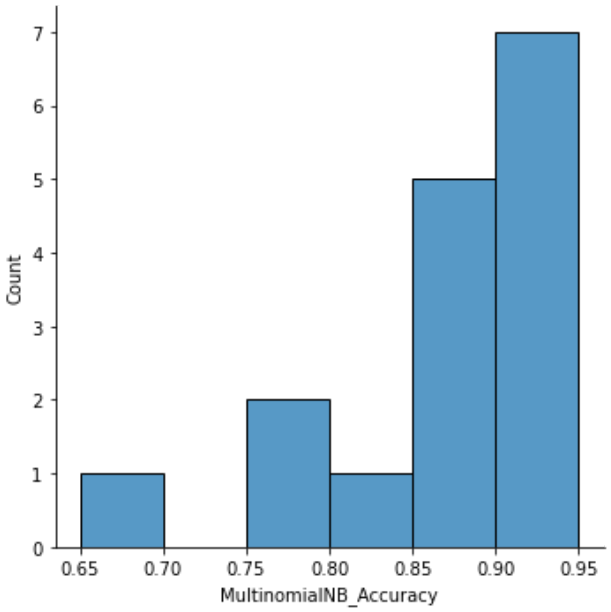
Out[50]:

| | Test_Size | Random_States | MultinomialNB_Accuracy |
|---|---|---|---|
| **0** | 0.30 | 14.0 | 0.851852 |
| **1** | 0.30 | 31.0 | 0.796296 |
| **2** | 0.30 | 42.0 | 0.888889 |
| **3** | 0.30 | 64.0 | 0.944444 |
| **4** | 0.67 | 14.0 | 0.925000 |
| **5** | 0.67 | 31.0 | 0.916667 |
| **6** | 0.67 | 42.0 | 0.925000 |
| **7** | 0.67 | 64.0 | 0.850000 |
| **8** | 0.90 | 14.0 | 0.832298 |
| **9** | 0.90 | 31.0 | 0.850932 |
| **10** | 0.90 | 42.0 | 0.869565 |
| **11** | 0.90 | 64.0 | 0.788820 |
| **12** | 0.11 | 14.0 | 0.950000 |
| **13** | 0.11 | 31.0 | 0.650000 |
| **14** | 0.11 | 42.0 | 0.900000 |
| **15** | 0.11 | 64.0 | 0.900000 |

# Histogram Plot of Accuracy

In [59]:
```
sns.displot(x = 'MultinomialNB_Accuracy', data =df_M )
```

Out[59]:
```
<seaborn.axisgrid.FacetGrid at 0x20e1c274820>
```




In [51]:
```
df_Final = pd.DataFrame(columns = ['Test_Size','Random_States','GaussianNB_Accuracy','BernoulliNB_Accuracy','MultinomialNB_Accuracy'])
```

In [52]:
```
for t_size in test_size:
    for r_state in random_states:
        Algo1 = doGaussianNBClassifier(X, Y, t_size, r_state)
        Algo2 = doBernoulliNBClassifier(X, Y, t_size, r_state)
        Algo5 = doMultinomialNBClassifier(X, Y, t_size, r_state)
        Final = {}
        Final['Test_Size'] = t_size
        Final['Random_States'] = r_state
        Final['GaussianNB_Accuracy'] = Algo1
        Final['BernoulliNB_Accuracy'] = Algo2
        Final['MultinomialNB_Accuracy'] = Algo5
        df_Final = df_Final.append(Final, ignore_index = True)
```

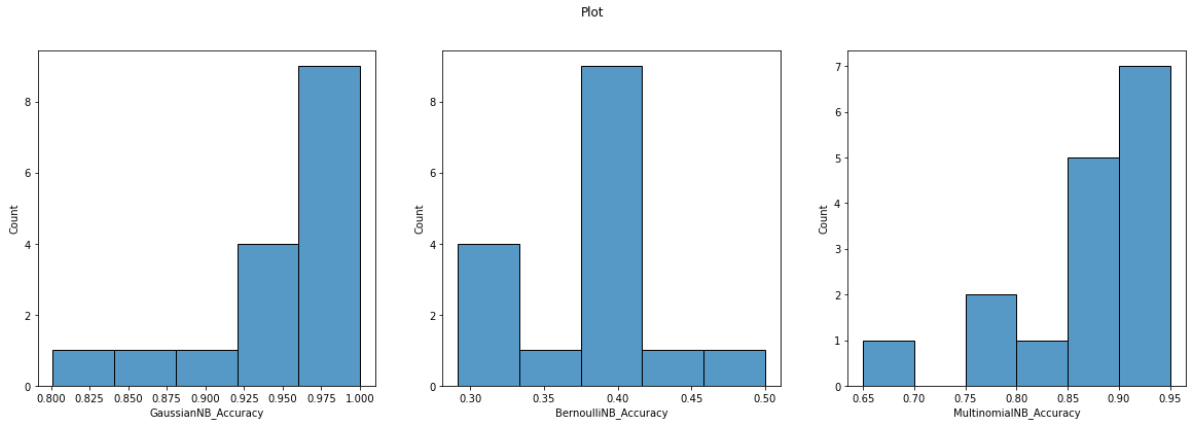
In [53]:
```
df_Final
```

Out[53]:

| | Test_Size | Random_States | GaussianNB_Accuracy | BernoulliNB_Accuracy | MultinomialNB_Accuracy |
|---|---|---|---|---|---|
| **0** | 0.30 | 14.0 | 1.000000 | 0.333333 | 0.851852 |
| **1** | 0.30 | 31.0 | 0.962963 | 0.388889 | 0.796296 |
| **2** | 0.30 | 42.0 | 1.000000 | 0.388889 | 0.888889 |
| **3** | 0.30 | 64.0 | 0.981481 | 0.351852 | 0.944444 |
| **4** | 0.67 | 14.0 | 0.975000 | 0.375000 | 0.925000 |
| **5** | 0.67 | 31.0 | 0.916667 | 0.391667 | 0.916667 |
| **6** | 0.67 | 42.0 | 0.958333 | 0.400000 | 0.925000 |
| **7** | 0.67 | 64.0 | 0.983333 | 0.291667 | 0.850000 |
| **8** | 0.90 | 14.0 | 0.925466 | 0.403727 | 0.832298 |
| **9** | 0.90 | 31.0 | 0.925466 | 0.391304 | 0.850932 |
| **10** | 0.90 | 42.0 | 0.801242 | 0.378882 | 0.869565 |
| **11** | 0.90 | 64.0 | 0.857143 | 0.316770 | 0.788820 |
| **12** | 0.11 | 14.0 | 1.000000 | 0.500000 | 0.950000 |
| **13** | 0.11 | 31.0 | 0.950000 | 0.450000 | 0.650000 |
| **14** | 0.11 | 42.0 | 1.000000 | 0.400000 | 0.900000 |
| **15** | 0.11 | 64.0 | 1.000000 | 0.300000 | 0.900000 |

# EDA of all types of models accuracy

In [66]:
```
columns = list(df_Final.columns)
columns.remove('Test_Size')
columns.remove('Random_States')
fig, ax = plt.subplots(ncols = 3, figsize=(20,6))
fig.suptitle("Plot")
for index, column in enumerate(columns):
    sns.histplot(df_Final[column], ax = ax[index])
plt.show()
```




# Evaluate the results in terms of hyper parameters and justify why we should/should not use Naive Bayes for this particular dataset

## Why we should use Naive Bayes

It is easy and fast to predict the class of the test data set. It also performs well in multi-class prediction.

When assumption of independence holds, a Naive Bayes classifier performs better compare to other models like logistic regression and you need less training data.

It perform well in case of categorical input variables compared to numerical variable(s). For numerical variable, normal distribution is assumed (bell curve, which is a strong assumption).

## Why we should not use Naive Bayes

Naive Bayes is the assumption of independent predictors. In real life, it is almost impossible that we get a set of predictors which are completely independent( here target variable is dependent on other independent variables)

If categorical variable has a category (in test data set), which was not observed in training data set, then model will assign a 0 (zero) probability and will be unable to make a prediction. This is often known as “Zero Frequency”. To solve this, we can use the smoothing technique. One of the simplest smoothing techniques is called Laplace estimation.

naive Bayes is also known as a bad estimator, so the probability outputs from predict_proba are not to be taken too seriously.

Because of the lack of categorical input variables, absence of independency in dataset and all here instead of naive bayes it's better to consider other supervised machine learning algorithm (Decision tree, random forest etc.)

# Conclusion

In this lab, we tried to understand about naive bayes and different types of models of naive bayes.

Bernouli is having lowest accuracy and Gaussian is having the highest.

Because of the lack of categorical input variable, absence of independency instead of naive bayes it's better to consider other supervised machine learning algorithm (Decision tree, random forest etc.) for wine dataset.

In [ ]: