Init: Import packages

[1]:
```
import numpy as np
import pandas as pd
from sklearn.naive_bayes import CategoricalNB # import Naive Bayes Classifier␣
↪for categroial distributed features
from sklearn.preprocessing import OrdinalEncoder
import matplotlib.pyplot as plt
from scipy.stats import norm
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as QDA
from sklearn.naive_bayes import GaussianNB
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn.inspection import DecisionBoundaryDisplay
```

Solution 1: Naive Bayes

a)

When using the naive Bayes classifier, the features $\mathbf{x} := (x_{\text{Color}}, x_{\text{Form}}, x_{\text{Origin}})$ are assumed to be conditionally independent of each other, given the category $y = k \in \{\text{yes}, \text{no}\}$, s.t.

$$P(\mathbf{x} \mid y = k) = P((x_{\text{Color}}, x_{\text{Form}}, x_{\text{Origin}}) \mid y = k) = P(x_{\text{Color}} \mid y = k) \cdot P(x_{\text{Form}} \mid y = k) \cdot P(x_{\text{Origin}} \mid y = k).$$

Recall Bayes' theorem:

$$\pi_k(\mathbf{x}) = P(y = k \mid \mathbf{x}) = \frac{P(\mathbf{x}|y = k)P(y = k)}{P(\mathbf{x})}.$$

As the denominator is constant across all classes, the following holds for the posterior probabilities:

$$\pi_k(\mathbf{x}) \propto \underbrace{\pi_k \cdot P(x_{\text{Color}} \mid y = k) \cdot P(x_{\text{Form}} \mid y = k) \cdot P(x_{\text{Origin}} \mid y = k)}_{=:\alpha_k(x)}$$
$$\iff \exists c \in \mathbb{R} : \pi_k(\mathbf{x}) = c \cdot \alpha_k(\mathbf{x}),$$

where $\pi_k = P(y = k)$ is the prior probability of class k and c is the normalizing constant.

From this and since the posterior probabilities need to sum up to 1, we know that

$$1 = c \cdot \alpha_{\text{yes}}(\mathbf{x}) + c \cdot \alpha_{\text{no}}(\mathbf{x}) \iff c = \frac{1}{\alpha_{\text{yes}}(\mathbf{x}) + \alpha_{\text{no}}(\mathbf{x})}.$$

This means that, in order to compute $\pi_{\text{yes}}(\mathbf{x})$, the scores $\alpha_{\text{yes}}(\mathbf{x})$ and $\alpha_{\text{no}}(\mathbf{x})$ are needed.

Now we want to estimate for a new fruit the posterior probability $\hat{\pi}_{yes}((\text{yellow}, \text{round}, \text{imported}))$.

Obviously, we do not know the *true* prior probability and the *true* conditional densities. Here – since the target and the features are categorical – we use a categorical distribution, i.e., the simplest distribution over a g-way event that is fully specified by the individual probabilities for each class (which must of course sum to 1). This is a generalization of the Bernoulli distribution to the multi-class case. We can estimate the distribution parameters via the relative frequencies encountered in the data:

$$\begin{aligned}\hat{\alpha}_{\text{yes}}(\mathbf{x}_*) &= \hat{\pi}_{yes} \cdot \hat{P}(x_{\text{Color}} = \text{yellow} \mid y = \text{yes}) \cdot \hat{P}(x_{\text{Form}} = \text{round} \mid y = \text{yes}) \cdot \hat{P}(x_{\text{Origin}} = \text{imported} \mid y = \text{yes}) \\ &= \frac{3}{8} \cdot \frac{1}{3} \cdot \frac{1}{3} \cdot 1 = \frac{1}{24} \approx 0.042, \\ \hat{\alpha}_{\text{no}}(\mathbf{x}_*) &= \hat{\pi}_{no} \cdot \hat{P}(x_{\text{Color}} = \text{yellow} \mid y = \text{no}) \cdot \hat{P}(x_{\text{Form}} = \text{round} \mid y = \text{no}) \cdot \hat{P}(x_{\text{Origin}} = \text{imported} \mid y = \text{no}) \\ &= \frac{5}{8} \cdot \frac{2}{5} \cdot \frac{3}{5} \cdot \frac{2}{5} = \frac{3}{50} = 0.060.\end{aligned}$$

At this stage we can already see that the predicted label is "no", since $\hat{\alpha}_{\text{no}}(\mathbf{x}_*) = 0.060 > \frac{1}{24} = \hat{\alpha}_{\text{yes}}(\mathbf{x}_*)$ – that is, if we threshold at 0.5 for predicting yes.

With the above we can compute the posterior probability

$$\hat{\pi}_{\text{yes}}(\mathbf{x}_*) = \frac{\hat{\alpha}_{\text{yes}}(\mathbf{x}_*)}{\hat{\alpha}_{\text{yes}}(\mathbf{x}_*) + \hat{\alpha}_{\text{no}}(\mathbf{x}_*)} \approx 0.410 < 0.5,$$

and check our calculations against the corresponding `Python` results:

[2]:
```
# Write dictionary for pandas Data Frame to save Bananas data
dic_bananas = {'ID': [1,2,3,4,5,6,7,8],
               'Color':
 ↪['yellow','yellow','yellow','brown','brown','green','green','red'],
               'Form':
 ↪['oblong','round','oblong','oblong','round','round','oblong','round'],
               'Origin':
 ↪['imported','domestic','imported','imported','domestic','imported','domestic','imported'],
               'Bananas':['yes','no','no','yes','no','yes','no','no']}
data_banana = pd.DataFrame(dic_bananas)
print(data_banana)
```

```
   ID   Color    Form    Origin Bananas
0   1  yellow  oblong  imported     yes
1   2  yellow   round  domestic      no
2   3  yellow  oblong  imported      no
3   4   brown  oblong  imported     yes
4   5   brown   round  domestic      no
5   6   green   round  imported     yes
6   7   green  oblong  domestic      no
7   8     red   round  imported      no
```

[3]:
```
#Transform your data with an ordial Encoder to get required input
enc = OrdinalEncoder() #Initialize Encoder
enc.fit(data_banana[["Color","Form", "Origin","Bananas"]]) # Fit encoder on␣
 ↪needed coulumns
#actually transform data and save in old data frame
```

```
data_banana[["Color","Form", "Origin","Bananas"]] = enc.
 ↪transform(data_banana[["Color","Form", "Origin","Bananas"]])

# split the data into inputs and outputs
X = data_banana.iloc[:, 1:4].values
y = data_banana.iloc[:, 4].values

x_new = np.asarray([3.,1.,1.])
x_new = np.reshape(x_new,(1,3)) #reshape needed for predict function


print(X)
print(y)
print(x_new)
```

```
[[3. 0. 1.]
 [3. 1. 0.]
 [3. 0. 1.]
 [0. 0. 1.]
 [0. 1. 0.]
 [1. 1. 1.]
 [1. 0. 0.]
 [2. 1. 1.]]
[1. 0. 0. 1. 0. 1. 0. 0.]
[[3. 1. 1.]]
```

```
[4]: # initializaing the NB
classifer = CategoricalNB(alpha=0) # alpha = 0 for no smoothing towards uniform␣
 ↪distribution!!

# training the model
classifer.fit(X, y)

# testing the model
y_pred = classifer.predict(x_new)
print("Prdection (0 = no, 1 = yes)", y_pred)
# Prediction is "not Banana"

y_prop = classifer.predict_proba(x_new)
print("Propabilities for (no - yes)", y_prop)
```

```
Prdection (0 = no, 1 = yes) [0.]
Propabilities for (no - yes) [[0.59016393 0.40983607]]
```

b)

Before, we only had categorical features and could use the empirical frequencies as our parameters in a categorical distribution. For the distribution of a numerical feature, given the the category, we need to define a probability distribution with continuous support. A popular choice is to use Gaussian distributions. For example, for the information x_{Length} we could assume that

$$P(x_{\text{Length}} \mid y = \text{yes}) \sim \mathcal{N}(\mu_{\text{yes}}, \sigma^2_{\text{yes}})$$

and

$$P(x_{\text{Length}} \mid y = \text{no}) \sim \mathcal{N}(\mu_{\text{no}}, \sigma^2_{\text{no}}).$$

In order to fully specify these normal distributions we need to estimate their parameters $\mu_{\text{yes}}, \mu_{\text{no}}, \sigma^2_{\text{yes}}, \sigma^2_{\text{no}}$ from the data via the usual estimators (empirical mean and empirical variance with bias correction).

Solution 2: Discriminant Analysis

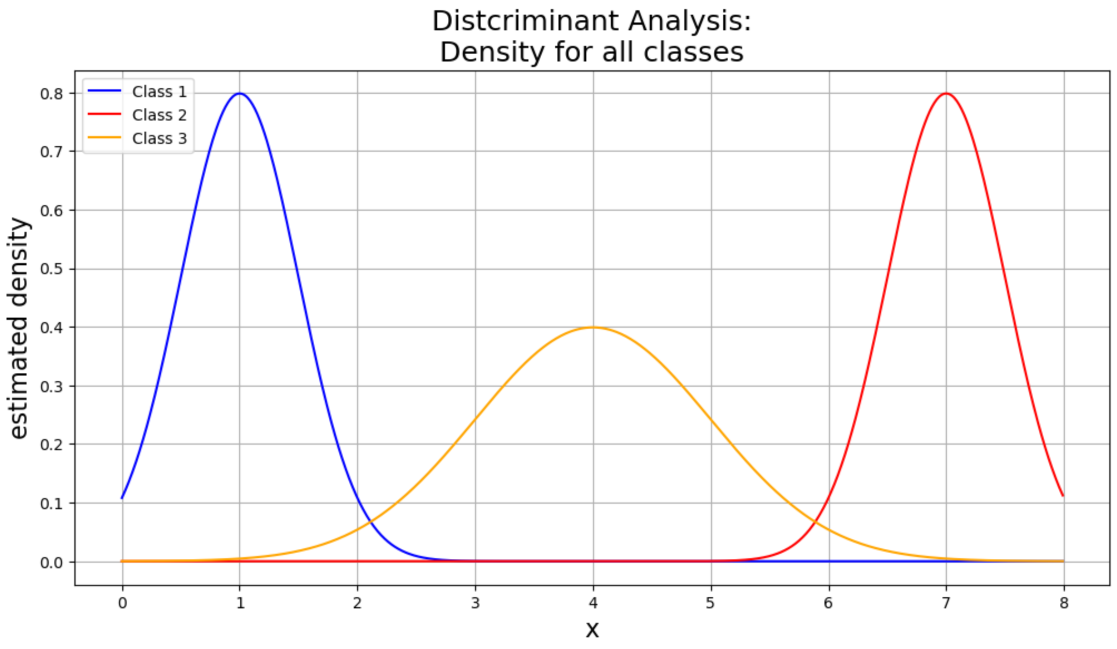
a) (i)

As the data seem to be pretty symmetric conditional on the respective class, we estimate the class means to lie roughly in the middle of the data clusters: $\hat{\mu}_1 = 1$, $\hat{\mu}_2 = 7$, $\hat{\mu}_3 = 4$.

(ii)

We see that the variances in classes 1 and 2 are similar and also much smaller than in class 3. Therefore, the densities could look roughly like this:

[5]:
```python
# Plot differences

#x-axis ranges from -3 and 3 with .001 steps
x = np.arange(0, 8, 0.01)

#plot normal distribution with mean 0 and standard deviation 1
plt.figure(figsize=(12, 6))
plt.plot(x, norm.pdf(x, 1, 0.5), color = 'blue', label = 'Class 1')
plt.plot(x, norm.pdf(x, 7, 0.5), color = 'red', label = 'Class 2')
plt.plot(x, norm.pdf(x, 4, 1), color = 'orange', label = 'Class 3')

# title & label axes
plt.grid(True)
plt.title('Distcriminant Analysis:\nDensity for all classes', size=18)
plt.xlabel('x', size=16)
plt.ylabel('estimated density', size=16)
plt.legend(loc='upper left', prop={'size': 10})
plt.show()
```



(iii)

Since LDA assumes constant variances across all classes (also if this does not reflect the data situation), all densities would have the same shape and only differ in location.

(iv)

As we have already noted, the assumption of equal class variances is not justified here, but LDA is confined to equivariant distributions. Therefore, the more flexible QDA is preferable in this case. Note, however, that the Gaussian distributions both variants of discriminant analysis use might not be perfectly appropriate, as the data seems to be more uniformly distributed (conditional on the classes).

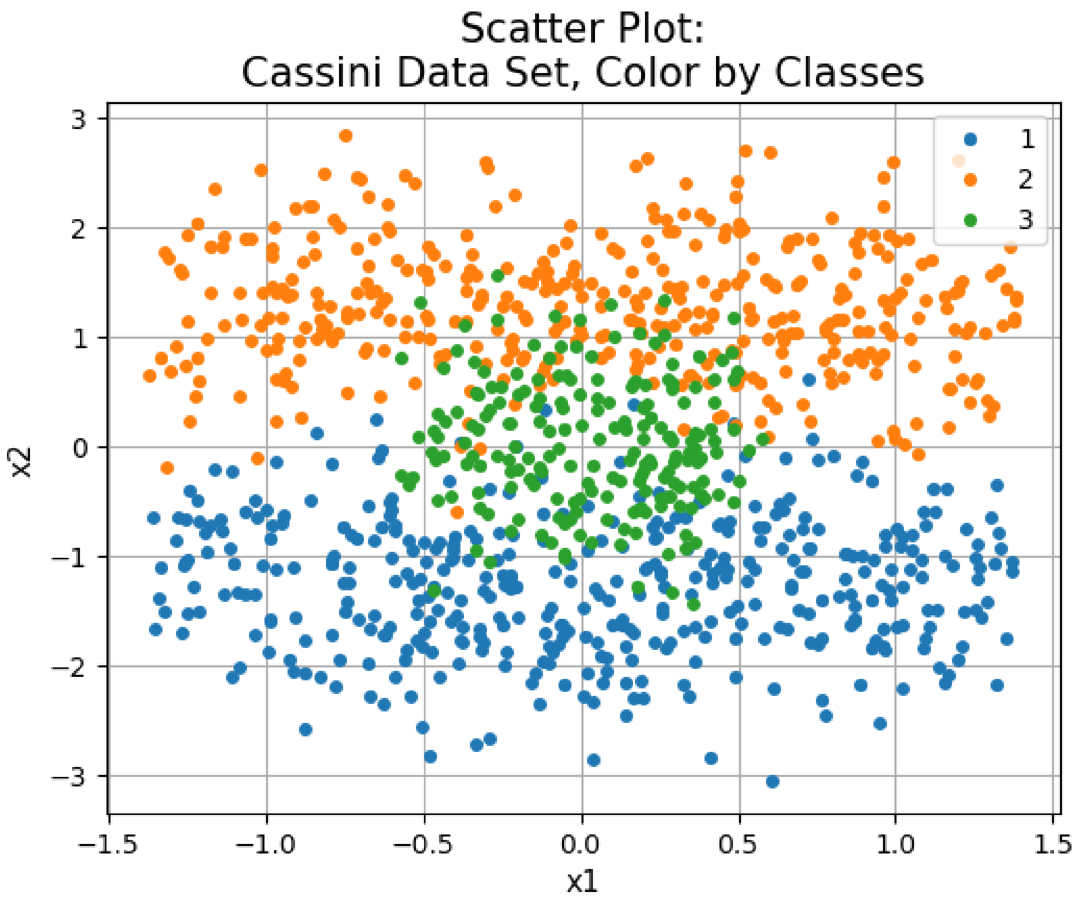
b)

The prediction for $\mathbf{x}_{*1}$ will probably be $\hat{z}_{*1} = 3$ because the density of class 3 has much larger variance and will therefore overshoot the density of class 1. For $\mathbf{x}_{*2}$ the case is clear and we have $\hat{z}_{*2} = 2$.

Exercise 3: Decision Boundaries

[7]:
```
# reading the CSV file
cassini = pd.read_csv('cassini_data.csv')

np.random.seed(43)
cassini["x.2"] = cassini["x.2"] + np.random.normal(loc=0.0, scale=0.5, size=1000)
```

[8]:
```
# Plot data by group
groups = cassini.groupby('classes')
for name, group in groups:
    plt.plot(group["x.1"], group["x.2"], marker='o', linestyle='', markersize=4,␣
 ↪label=name)

plt.legend()
plt.grid(True)
plt.title('Scatter Plot:\nCassini Data Set, Color by Classes', size=15)
plt.xlabel('x1', size=11)
plt.ylabel('x2', size=11)
plt.show()
```

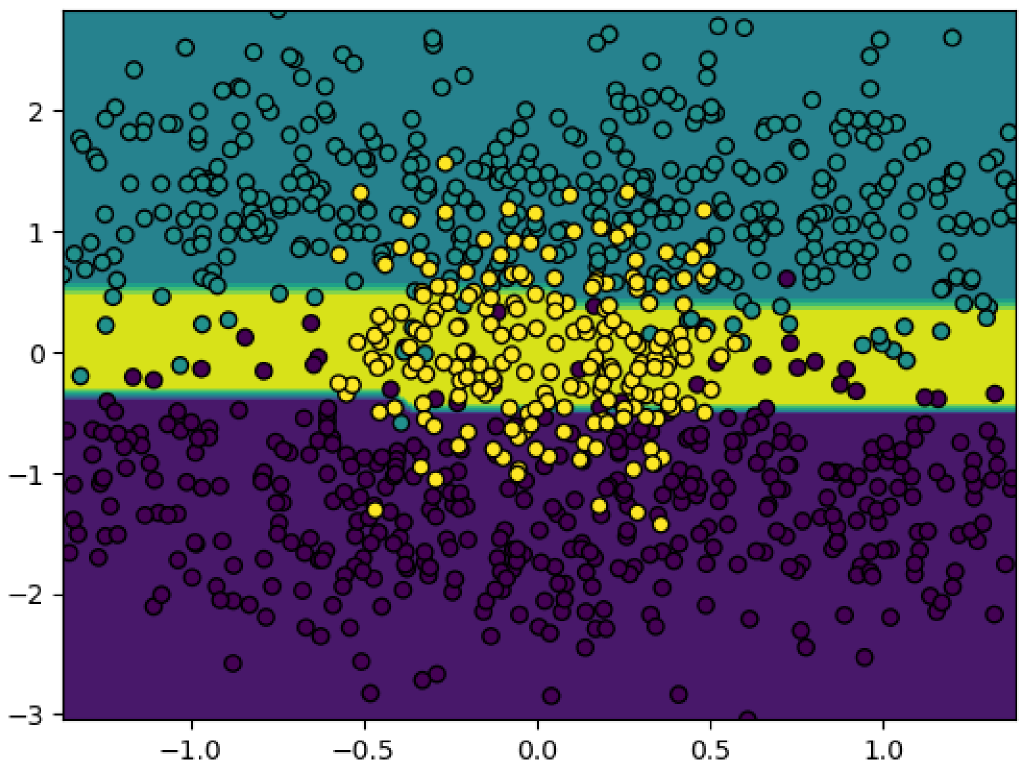


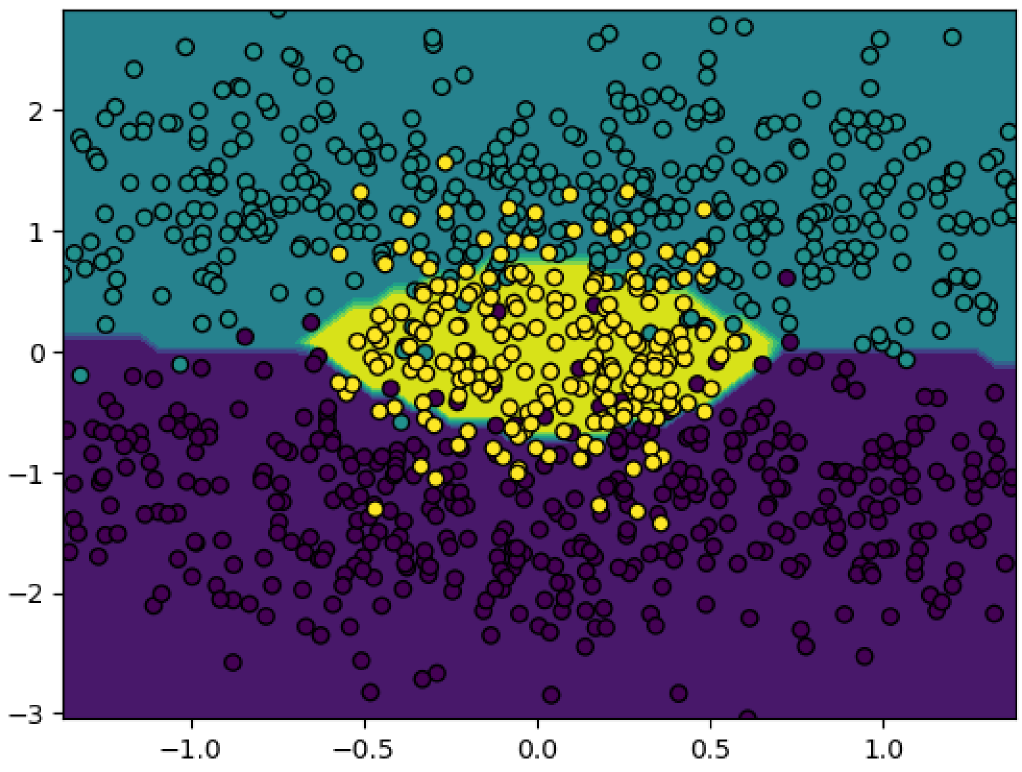
[9]:
```
# Train all 3 models
X_cass = cassini.iloc[:, 0:2].values
y_cass = cassini.iloc[:, 2].values

# LDA
lda = LDA()
lda.fit(X_cass, y_cass)

# QDA
qda = QDA()
qda.fit(X_cass, y_cass)
```

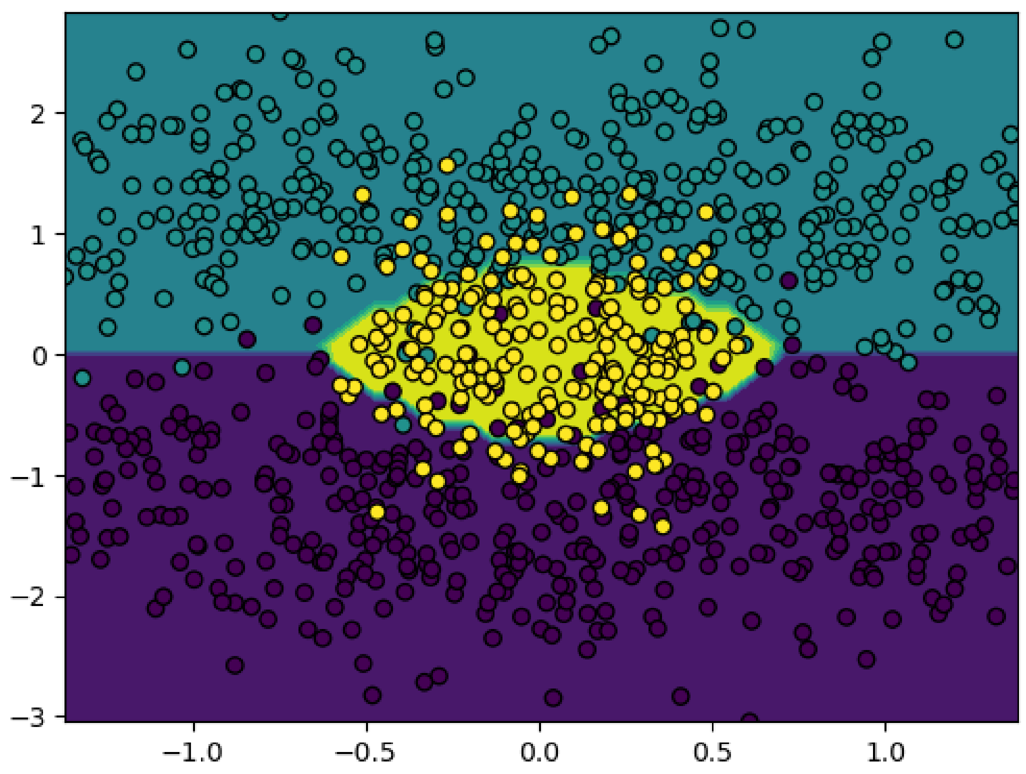
```
# Naive Bayes
# Use Gaussian Naive Bayes
gnb = GaussianNB(var_smoothing=0) # no smoothing wanted here
gnb.fit(X_cass, y_cass)
```

[9]:
```
GaussianNB(var_smoothing=0)
```

[10]:
```
# Plot Decision Boundaries
def plot_dec_bound(obj):
    """
    Method to produce Decision Boundary Plots
    Input: trained classifier object
    Output: -
    """
    feature_1, feature_2 = np.meshgrid(
        np.linspace(X_cass[:, 0].min(), X_cass[:, 0].max()),
        np.linspace(X_cass[:, 1].min(), X_cass[:, 1].max()))
    grid = np.vstack([feature_1.ravel(), feature_2.ravel()]).T

    y_pred = np.reshape(obj.predict(grid), feature_1.shape)
    display = DecisionBoundaryDisplay(xx0=feature_1, xx1=feature_2,␣
 ↪response=y_pred)
    display.plot()

    display.ax_.scatter(X_cass[:, 0], X_cass[:, 1], c=y_cass, edgecolor="black")

    plt.show()


class_list = [lda, qda, gnb]

for obj in class_list:
    plot_dec_bound(obj)
```


2
1
0
−1
−2
−3
−1.0
−0.5
0.0
0.5
1.0

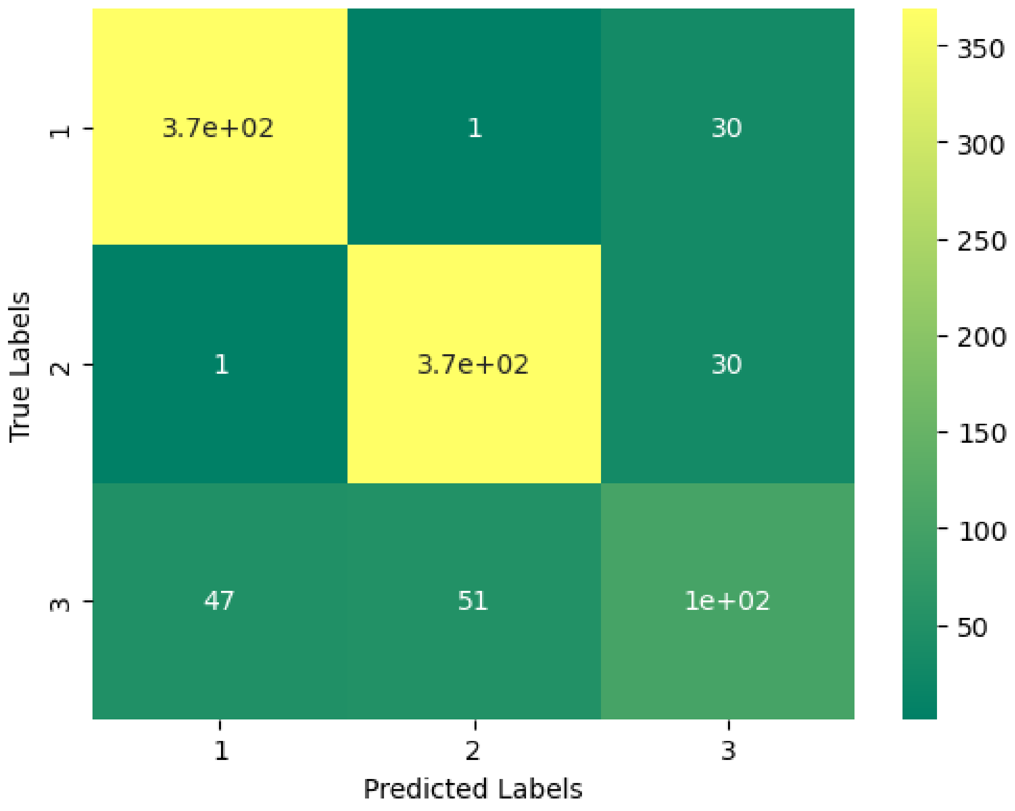

2
1
0
−1
−2
−3
−1.0
−0.5
0.0
0.5
1.0



We see how LDA, with its confinement to linear decision boundaries, is not able to classify the data very well. QDA and NB, on the other hand, get the shape of the boundaries right. It also becomes obvious that NB is a quadratic classifier just like QDA - their decision surfaces look pretty much alike.

Extra: Confusion Matrix for Multiclass Classifier

[11]:
```
# plot confusion matrix for multiclass
# LDA
sns.heatmap(confusion_matrix(y_cass, lda.predict(X_cass)), annot = True,␣
↪xticklabels = np.unique(y_cass), yticklabels = np.unique(y_cass), cmap =␣
↪'summer')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```


3.7e+02
1
30
1
3.7e+02
30
47
51
1e+02
True Labels
Predicted Labels
1
2
3
350
300
250
200
150
100
50