```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC # support vector classification
from sklearn.model_selection import train_test_split
from helperFunctions import measurePerformance, groupScatter, plotGNB,
plotSVM

snails = pd.read_csv("Snails.csv")
print("Entire Dataset.,")
print(snails)
print("Header.,that is column names in the dataset.,")
print(snails.columns.values)
print('Number of data in the dataframe = ',len(snails))
print("Minimum values: ", snails.min())
print("Maximum values:", snails.max())
print("Unique labels:",snails['species'].unique())
```

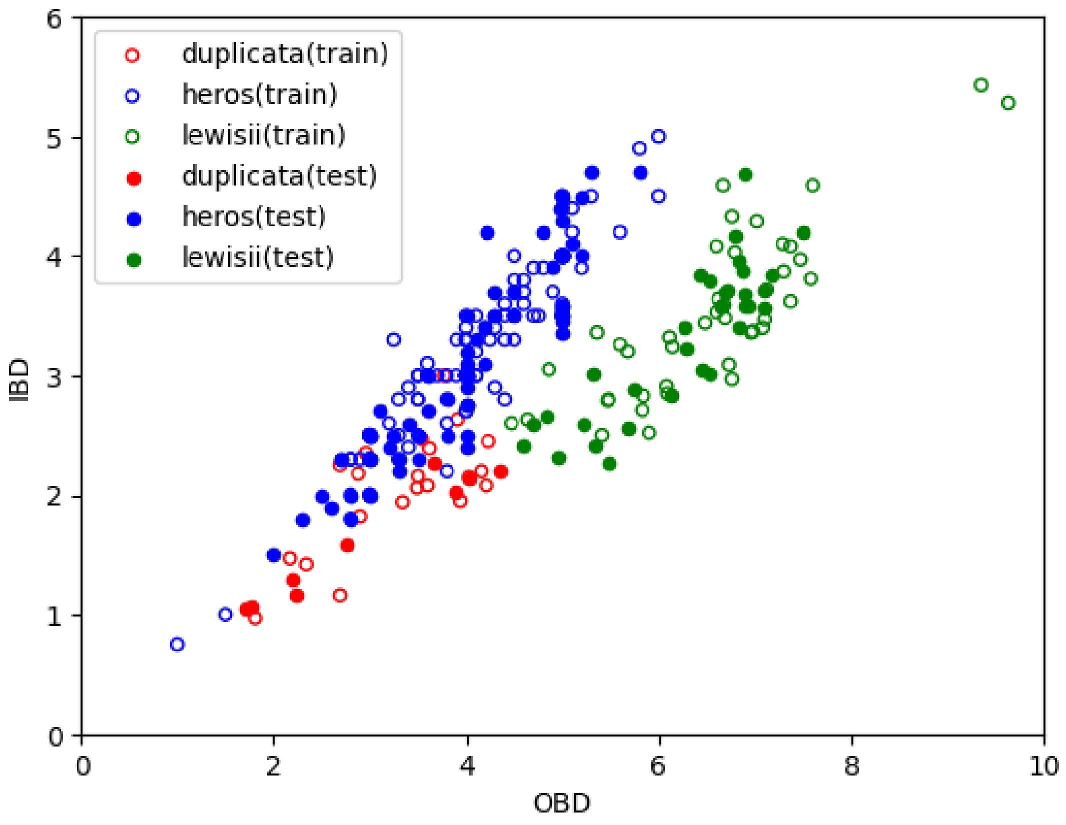
```
Entire Dataset.,
       species   OBD   IBD
0    duplicata  3.67  2.27
1    duplicata  4.35  2.20
2    duplicata  4.02  2.14
3    duplicata  3.94  1.95
4    duplicata  4.21  2.08
..         ...   ...   ...
281      heros  5.00  3.35
282      heros  5.00  4.00
283      heros  5.00  3.50
284      heros  5.00  4.50
285      heros  6.00  5.00

[286 rows x 3 columns]
Header.,that is column names in the dataset.,
['species' 'OBD' 'IBD']
Number of data in the dataframe =  286
Minimum values:  species    duplicata
OBD                1.0
IBD               0.75
dtype: object
Maximum values: species    lewisii
OBD             9.63
IBD             5.43
dtype: object
Unique labels: ['duplicata' 'lewisii' 'heros']
```

```python
# Now that you know the range of the data features and the unique set
of label names
# you can define some sensible plotting limits and plotting colours.
```

```
xlim = [0,10]
ylim = [0,6]
colors  = {'duplicata':'r','lewisii':'g','heros':'b'}
```

1)We don't just grab the first half of the data for training because it could make our model biased. Instead, we mix things up by shuffling the data randomly. This helps our model learn more effectively, preventing it from getting stuck in any order-specific patterns.

```
# Randomly split the data into training and test sets
train, test = train_test_split(snails, test_size=0.4,
random_state=2024)

# Group data by species
train_grouped = train.groupby('species')
test_grouped = test.groupby('species')

# Extract features and labels
train_features = train[['OBD', 'IBD']].values
train_labels = train['species']
test_features = test[['OBD', 'IBD']].values
test_labels = test['species']

#Number of data points in the training and test sets
print("Number of data points in training set:", len(train))
print("Number of data points in test set:", len(test))
# Scatter plot using groupScatter function
groupScatter(train_grouped,colors,xlim,ylim,'OBD','IBD',20,1,'(train)'
)
groupScatter(test_grouped,colors,xlim,ylim,'OBD','IBD',20,0,'(test)')
plt.show()

Number of data points in training set: 171
Number of data points in test set: 115

/usr/local/lib/python3.10/dist-packages/pandas/plotting/_matplotlib/
core.py:1259: UserWarning: No data for colormapping provided via 'c'.
Parameters 'cmap' will be ignored
  scatter = ax.scatter(
```



2)Number of data points in training set: 171

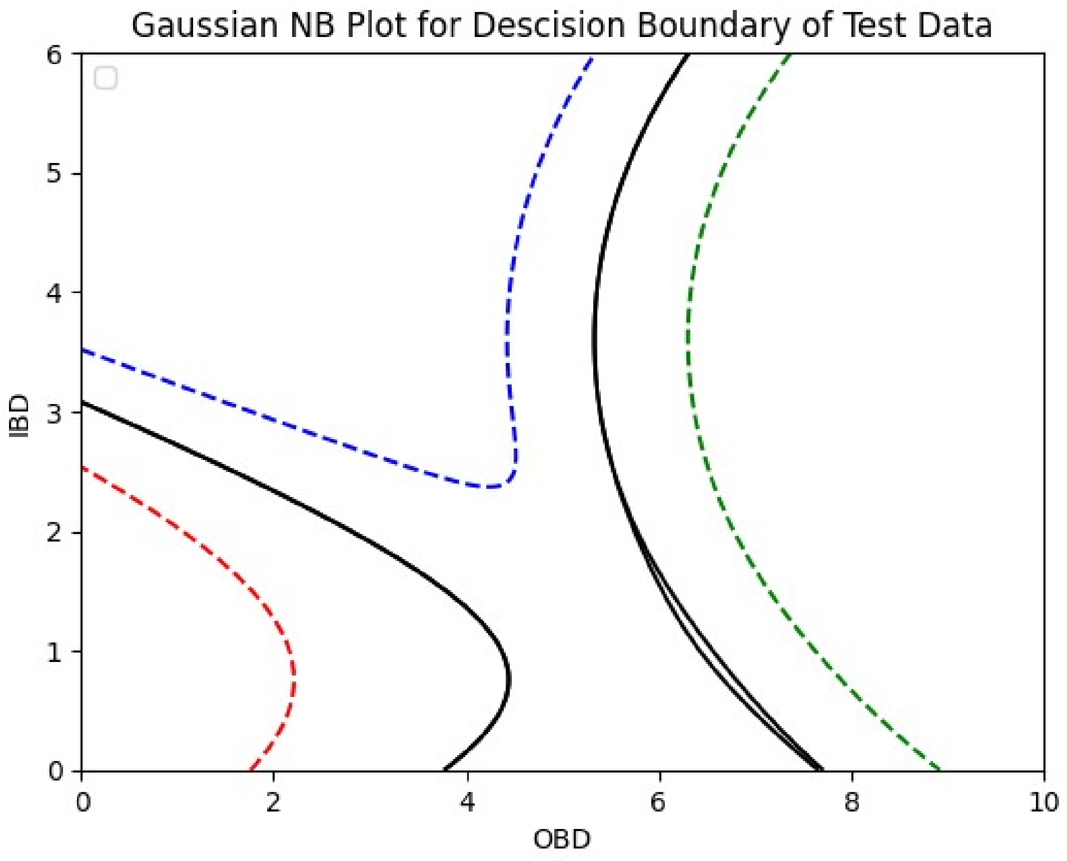
Number of data points in test set: 115

```
def plotGNB(model,groups,colors,xlim,ylim):
   # Predict the classification probabilities on a grid:
   ax = plt.gca()
   n = 500 # number of grid points in each direction
   x = np.linspace(xlim[0],xlim[1],n)
   y = np.linspace(ylim[0],ylim[1],n)
   X, Y = np.meshgrid(x,y)
   xy = model.predict_proba(np.c_[X.ravel(), Y.ravel()])
   sh = np.shape(xy)
   ncol = sh[1]
   i = 0
   for key, group in groups:
      Z = xy[:,i].reshape(X.shape)
      # Plot 50% confidence contour:
      ax.contour(X,Y,Z,levels=[0.5],colors='k')
      # Plot 95% confidence contour:

ax.contour(X,Y,Z,levels=[0.95],colors=colors[key],linestyles='dashed')
      i += 1
```

```
model0 = GaussianNB()
model0.fit(test_features,test_labels)
plotGNB(model0,test_grouped,colors,xlim,ylim)
plt.title("Gaussian NB Plot for Descision Boundary of Test Data")
plt.legend(loc='upper left')
plt.xlabel("OBD")
plt.ylabel("IBD")
```

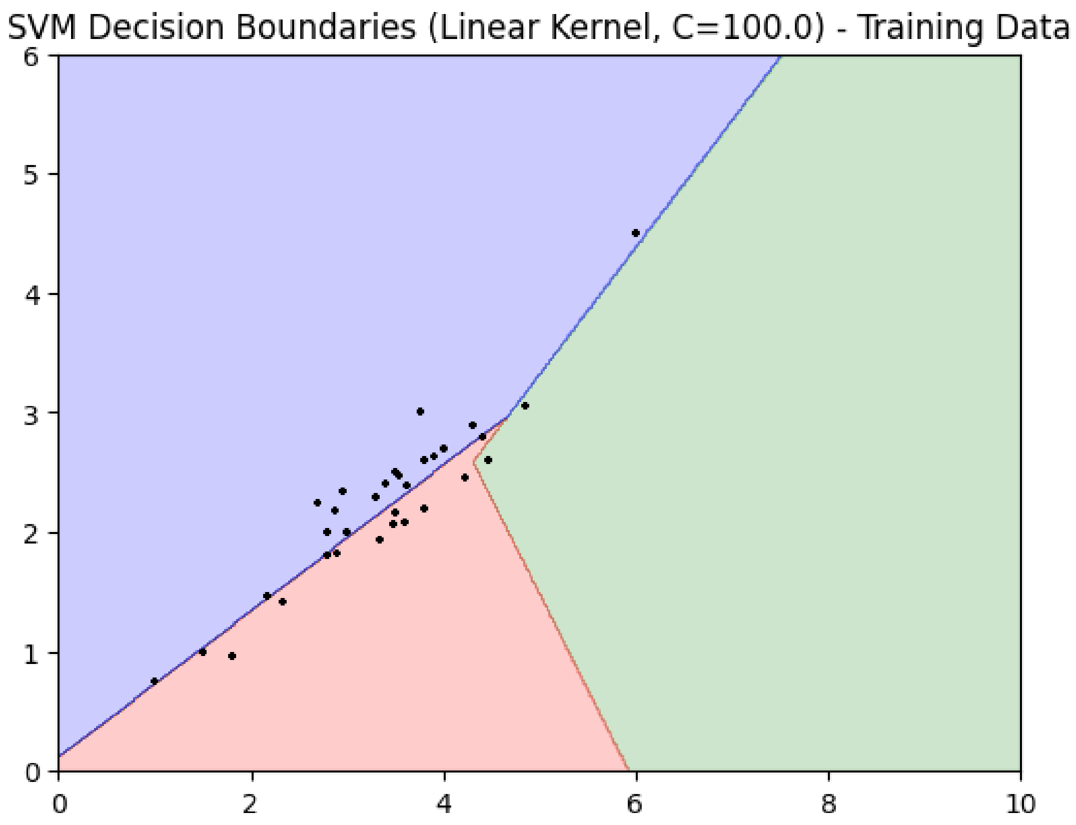
```
WARNING:matplotlib.legend:No artists with labels found to put in
legend.  Note that artists whose label start with an underscore are
ignored when legend() is called with no argument.
```

```
Text(0, 0.5, 'IBD')
```



```
# SVM with linear kernel and C=1E2.
C = 1E2
model = SVC(kernel='linear', C=C)

# Training the SVM model.
model.fit(train_features, train_labels)

# Plotting SVM decision boundaries and training data.
plotSVM(model, train_grouped, colors, xlim, ylim, markersize=8)
```

```
# Set the title separately.
plt.title(f'SVM Decision Boundaries (Linear Kernel, C={C}) - Training
Data')
plt.show()
```



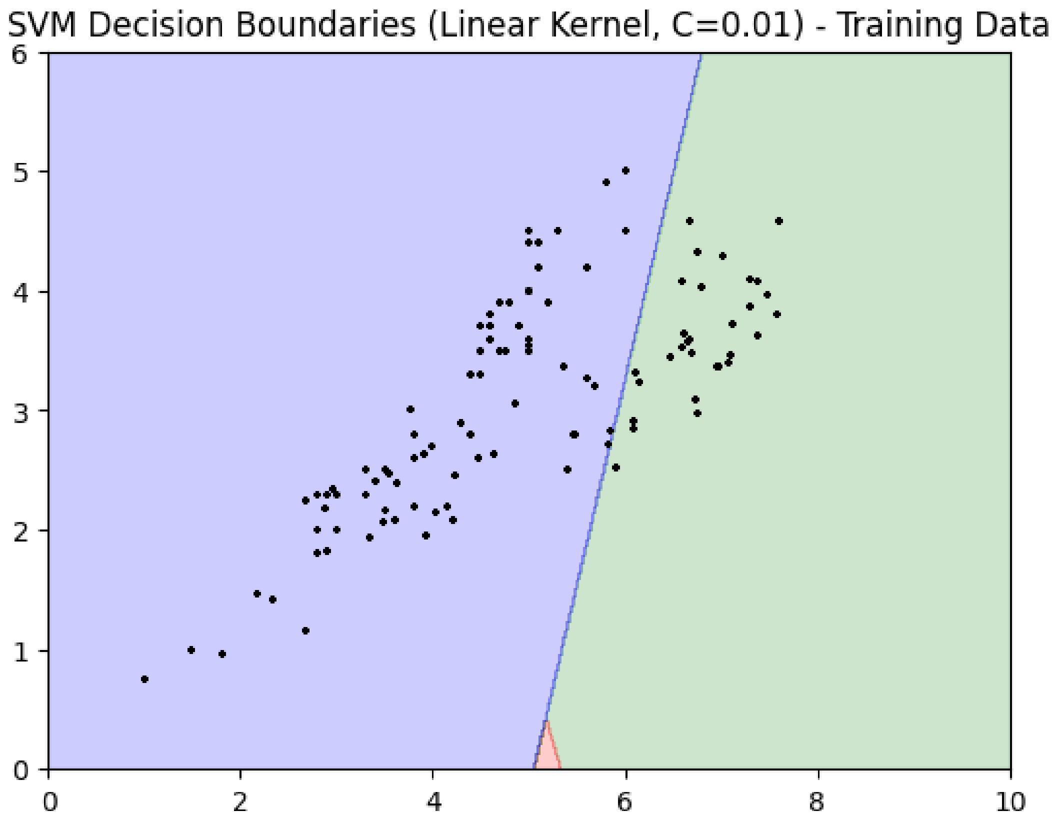
3)High C (C_linear = 1E2) - Linear Kernel: Starting with a high C, the SVM model prioritizes precise classification of training points, potentially resulting in a complex decision boundary.

Low C (C_low_linear = 1E-2) - Linear Kernel: Reducing C introduces more regularization, favoring a simpler decision boundary that may generalize better to unseen data.

which is seen below in the plotting

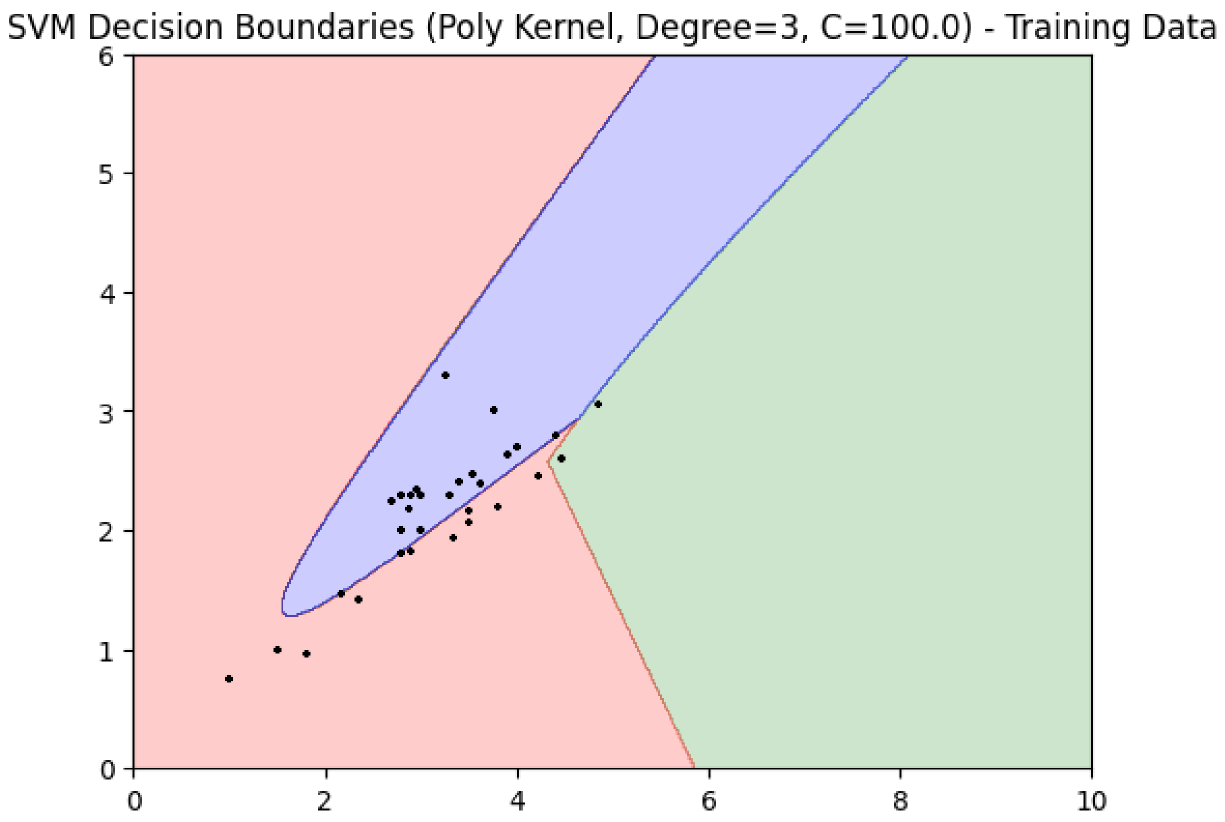
```
# Try SVM with the linear kernel again but with lower values of C.
# You could also try higher values of C if you like.
# WARNING: higher values of C can take a long time to run. Be careful!

# Lower value of C
C_low = 1E-2
model_low_C = SVC(kernel='linear', C=C_low)

# Training the SVM model with lower C.
model_low_C.fit(train_features, train_labels)
```

```python
# Plotting SVM decision boundaries and training data for lower C.
plotSVM(model_low_C, train_grouped, colors, xlim, ylim, markersize=8)

# Set the title separately.
plt.title(f'SVM Decision Boundaries (Linear Kernel, C={C_low}) - 
Training Data')
plt.show()
```

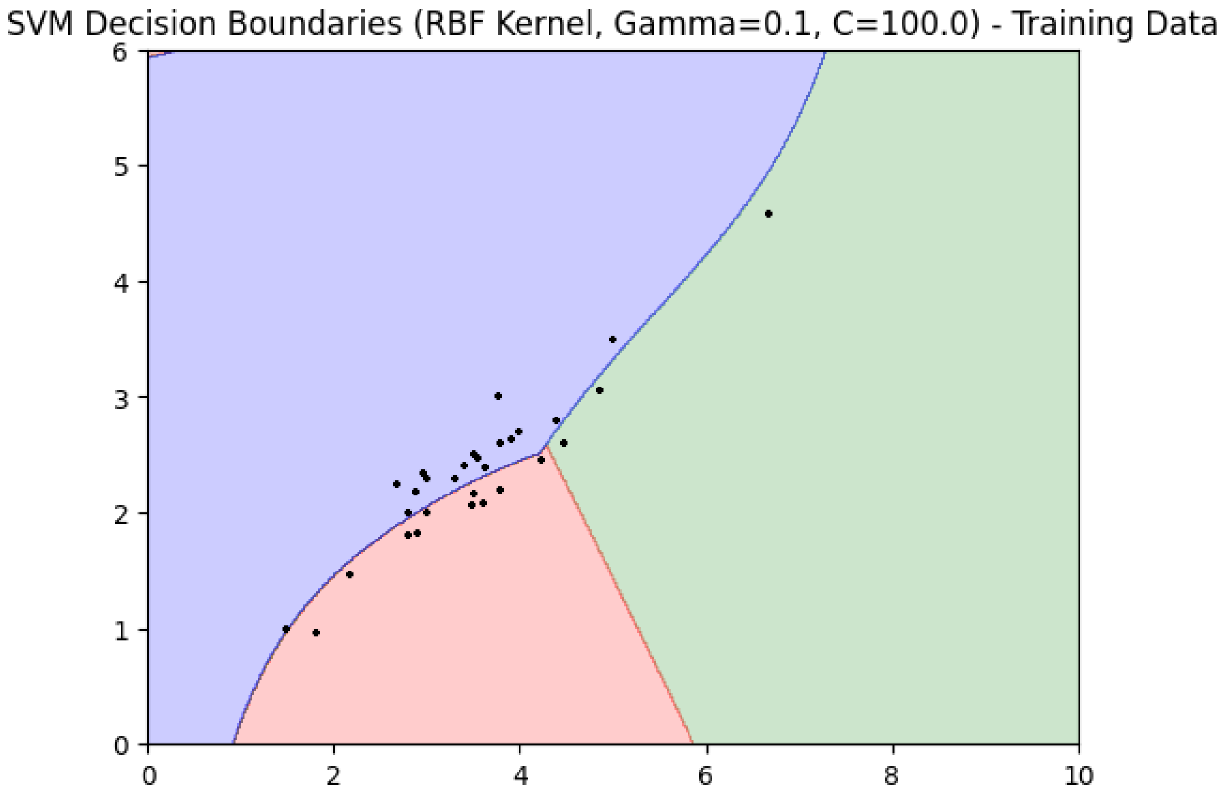


```python
# Polynomial Kernel
C_poly = 1E2
degree_poly = 3
model_poly = SVC(kernel='poly', degree=degree_poly, C=C_poly)

# Training the SVM model with polynomial kernel.
model_poly.fit(train_features, train_labels)

# Plotting SVM decision boundaries and training data for polynomial 
kernel.
plotSVM(model_poly, train_grouped, colors, xlim, ylim, markersize=8)
plt.title(f'SVM Decision Boundaries (Poly Kernel, 
Degree={degree_poly}, C={C_poly}) - Training Data')
plt.show()
```

```
# Measure performance on test data.
percent_matched_test_poly = measurePerformance(model_poly,
test_features, test_labels)

print(percent_matched_test_poly)
```



```
97.3913043478261

# RBF Kernel
C_rbf = 1E2
gamma_rbf = 0.1
model_rbf = SVC(kernel='rbf', gamma=gamma_rbf, C=C_rbf)

# Training the SVM model with RBF kernel.
model_rbf.fit(train_features, train_labels)

# Plotting SVM decision boundaries and training data for RBF kernel.
plotSVM(model_rbf, train_grouped, colors, xlim, ylim, markersize=8)
plt.title(f'SVM Decision Boundaries (RBF Kernel, Gamma={gamma_rbf},
C={C_rbf}) - Training Data')
plt.show()

# Measure performance on test data.
percent_matched_test_rbf = measurePerformance(model_rbf,
test_features, test_labels)

print(percent_matched_test_rbf)
```

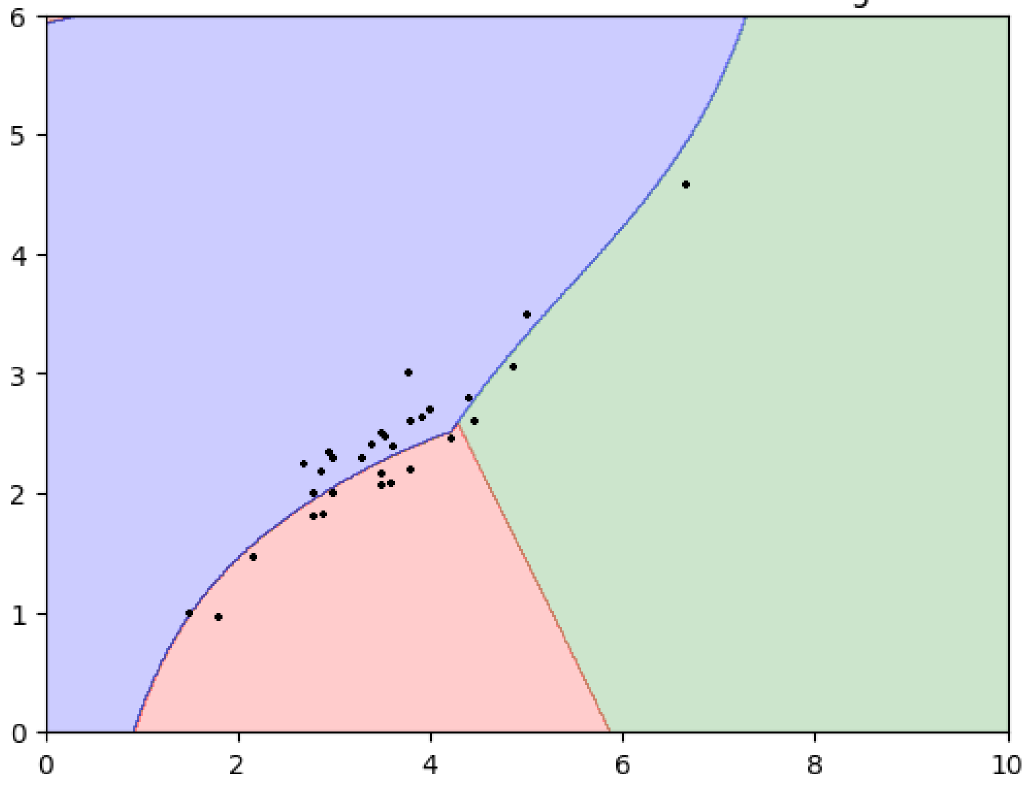


```
97.3913043478261

# For ease of grading, include the modelling and performance
calculation for your best classifier here.
# Also generate a plot like you did previously of the decision
boundaries and training data.

# Example for Gaussian Naive Bayes
gnb_model = GaussianNB()
gnb_model.fit(train_features, train_labels)

# Measure performance on test data for Gaussian Naive Bayes.
percent_matched_test_gnb = measurePerformance(gnb_model,
test_features, test_labels)

# Example for SVM with RBF Kernel
C_rbf = 1E2
gamma_rbf = 0.1
svm_rbf_model = SVC(kernel='rbf', gamma=gamma_rbf, C=C_rbf)
svm_rbf_model.fit(train_features, train_labels)
```

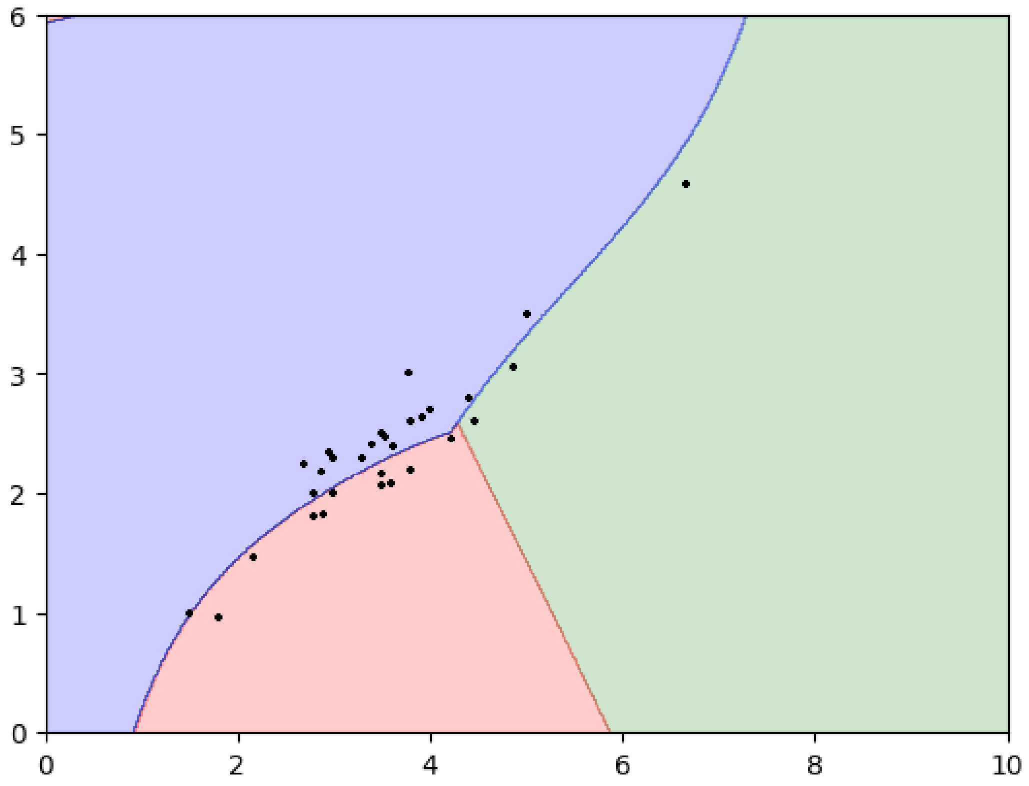
```
# Measure performance on test data for SVM with RBF Kernel.
percent_matched_test_rbf = measurePerformance(svm_rbf_model,
test_features, test_labels)

# Compare and print performances
print('Performance Comparison:')
print(f'Gaussian Naive Bayes: {percent_matched_test_gnb:.2f}%
correctly matched on test data')
print(f'SVM with RBF Kernel: {percent_matched_test_rbf:.2f}% correctly
matched on test data')

# Choose the best-performing classifier
best_classifier = svm_rbf_model

# Print performance for the best classifier
percent_matched_test_best = measurePerformance(best_classifier,
test_features, test_labels)
print(f'Percent matched for the best classifier:
{percent_matched_test_best:.2f}%')

# Plot decision boundaries and training data for the best classifier
plotSVM(best_classifier, train_grouped, colors, xlim, ylim,
markersize=8)
plt.title(f'Best Classifier Decision Boundaries and Training Data')
plt.show()

# Plot decision boundaries and test data for the best classifier
plotSVM(best_classifier, test_grouped, colors, xlim, ylim,
markersize=8)
plt.title(f'Best Classifier Decision Boundaries and Test Data')
plt.show()
```

```
Performance Comparison:
Gaussian Naive Bayes: 76.52% correctly matched on test data
SVM with RBF Kernel: 97.39% correctly matched on test data
Percent matched for the best classifier: 97.39%
```

Best Classifier Decision Boundaries and Training Data



Best Classifier Decision Boundaries and Test Data



4)Performance Comparison: Gaussian Naive Bayes: 76.52% correctly matched on test data SVM with RBF Kernel: 97.39% correctly matched on test data Percent matched for the best classifier: 97.39%

5)In SVMs, I feel that adjusting specific parameters plays a crucial role in controlling how closely the model tailors itself to the training data. For the linear type, if I want a looser fit, I'd decrease the "C" parameter. On the other hand, if I want the model to stick more closely to the data, I'd opt to increase C. When it comes to the polynomial type, adjusting the "degree" parameter allows me to manage the model's complexity. Higher degrees make the model more detailed, potentially fitting the training data too closely, while lower degrees keep things simpler. Lastly, for the RBF type, manipulating the "gamma" parameter influences how focused the model is on individual points. More gamma results in increased focus, which might lead to fitting too much to the training data, whereas less gamma fosters a more generalized approach. Overall, I find that these adjustments provide a nuanced control over the trade-off between fitting the training data and avoiding overfitting.