Group 12 HW1

Contents

1- SVM Classifier

a) Default SVM

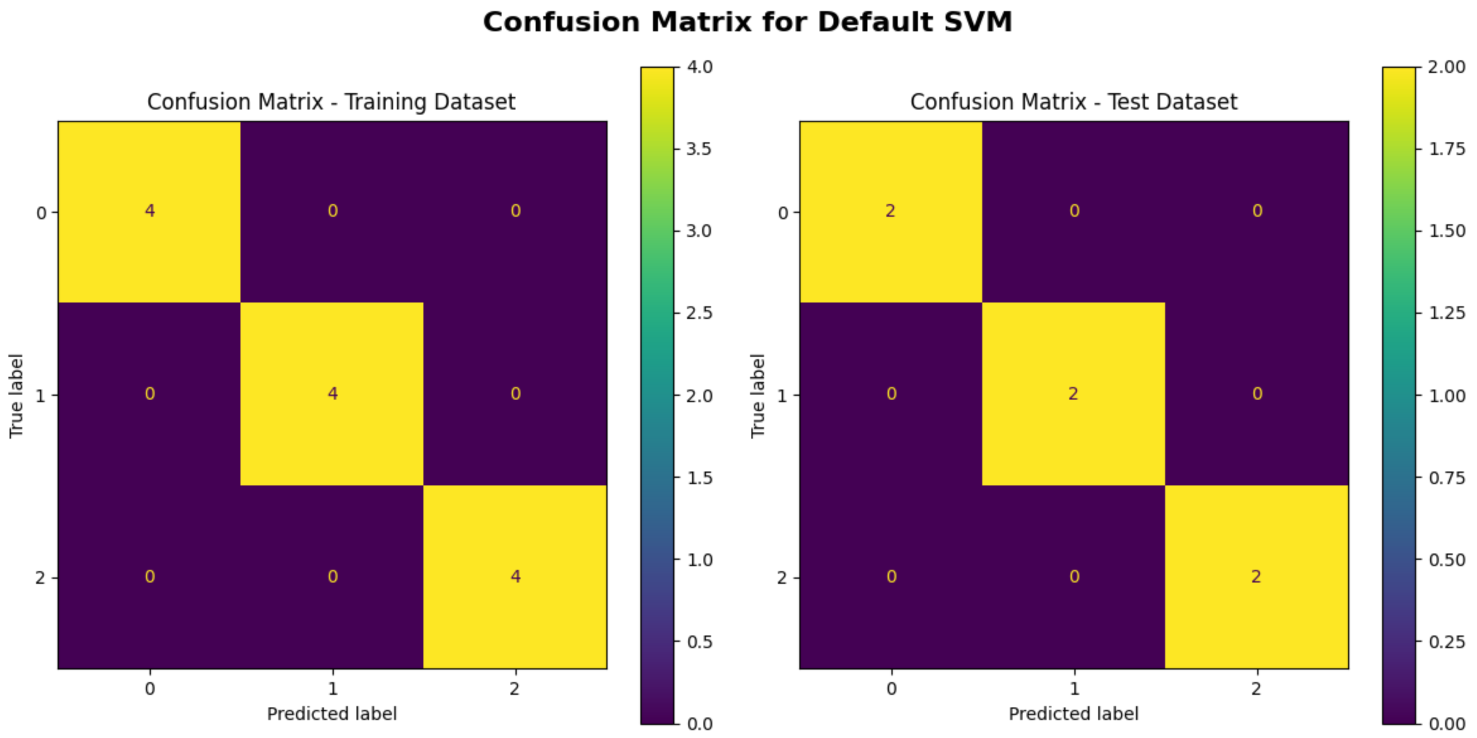
Implement a default SVM classifier. Train the model using the dataset.

```
# Train the SVM classifier
classifier = svm.SVC()
classifier.fit(X_Train, Y_Train)
```

- C: 1.0
- kernel: 'rbf' (Radial basis function kernel)
- degree: 3 (Degree of the polynomial kernel function)
- gamma: 'scale' (Kernel coefficient for 'rbf', 'poly', and 'sigmoid')
- coef0: 0.0 (Independent term in the kernel function)

Obtain confusion matrices for training and test datasets.

```
# Create a figure with two subplots
fig, axs = plt.subplots(1, 2, figsize=(12, 6))

# Plot the confusion matrix for the test dataset
test_cm = ConfusionMatrixDisplay.from_estimator(classifier, X_Test, Y_Test, ax=axs[1])
test_cm.ax_.set_title('Confusion Matrix - Test Dataset')

# Plot the confusion matrix for the training dataset
train_cm = ConfusionMatrixDisplay.from_estimator(classifier, X_Train, Y_Train, ax=axs[0])
train_cm.ax_.set_title('Confusion Matrix - Training Dataset')

# Add a suptitle for the entire plot
plt.suptitle('Confusion Matrix for Default SVM', fontsize=16, fontweight='bold')

# Adjust spacing between subplots
plt.tight_layout()

# Show the plot
plt.show()
```

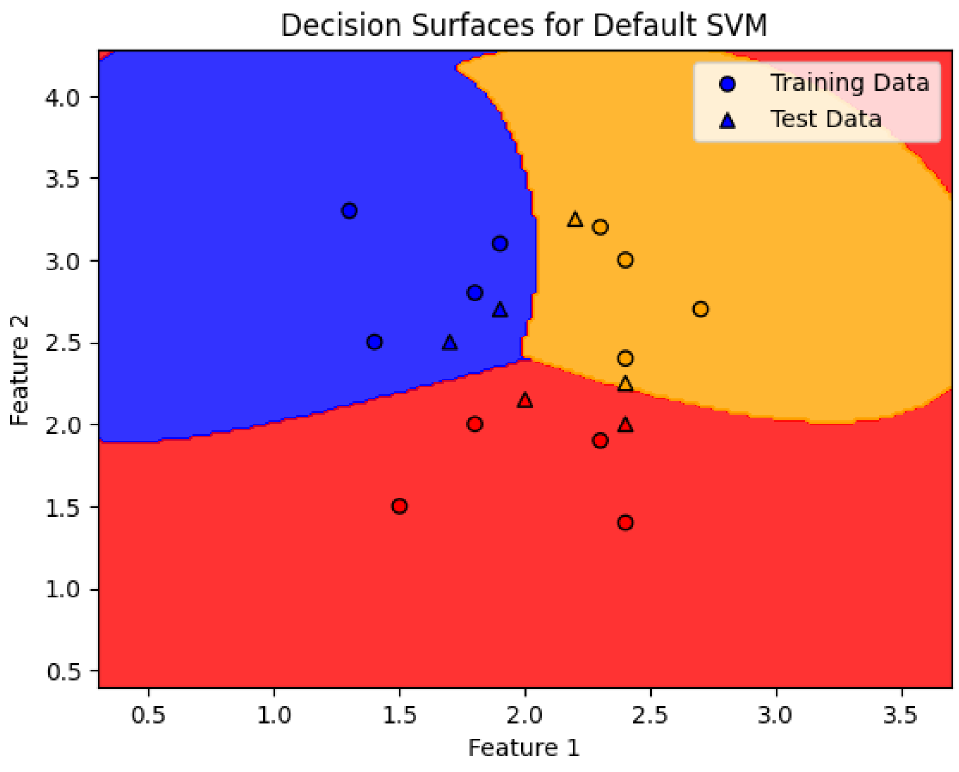


Using blue, red, and orange colors to plot each class.

```
# Create a colormap with custom colors
# Define custom colors
colors = ['blue', 'red', 'orange']
cmap = ListedColormap(colors)
```

Visualize decision surfaces for multi-calss classification (Default SVM).

```
#Get all data in order not to miss any point

X_combined = np.concatenate((X_Train, X_Test))
Y_combined = np.concatenate((Y_Train, Y_Test))

# Create a meshgrid for plotting decision regions
h = 0.02  # Step size in the mesh
x_min, x_max = X_combined[:, 0].min() - 1, X_combined[:, 0].max() + 1
y_min, y_max = X_combined[:, 1].min() - 1, X_combined[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
#this will be used in all future plots xx,yy.
# Make predictions for each point in the meshgrid
Z = classifier.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot the decision surfaces
plt.contourf(xx, yy, Z, alpha=0.8, cmap=cmap)

# Plot training data points make edges black and in circle shape
plt.scatter(X_Train[:, 0], X_Train[:, 1], c=Y_Train, cmap=cmap, edgecolors='k', marker='o',
label='Training Data')

# Plot test data points make edges black and in tringle shape
plt.scatter(X_Test[:, 0], X_Test[:, 1], c=Y_Test, cmap=cmap, edgecolors='k', marker='^',
label='Test Data')

# Add legends and labels
plt.legend()
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Decision Surfaces for Default SVM')

# Show the plot
plt.show()
```



b) (b) One-vs-Rest SVM and Perceptron

Training and test data should be labeled for one vs rest as binary classifier. For example, class 0 vs rest means class 0 will be labeled as 1 and the rest (classes 1 and 2) will be labeled as 0.

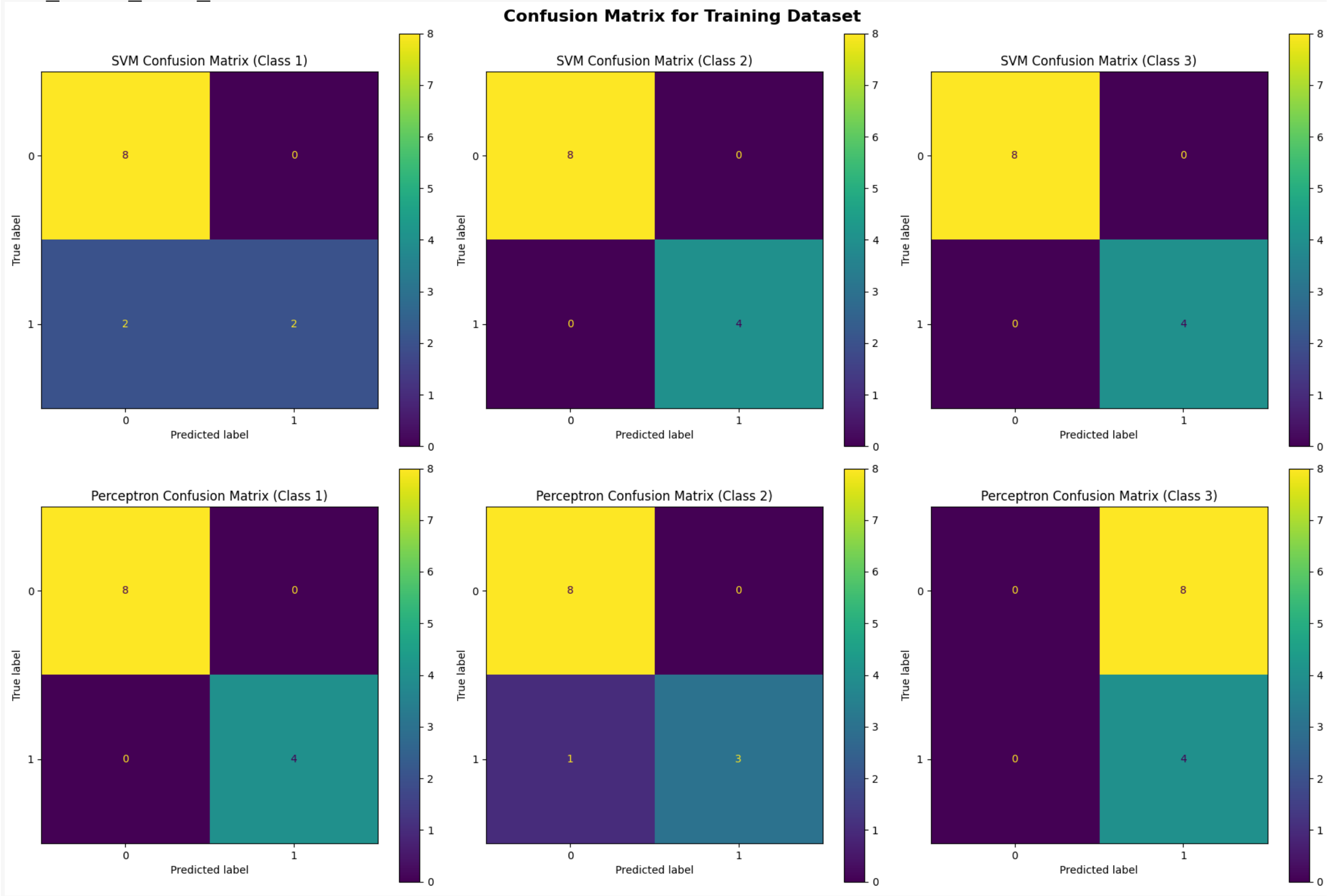
```
# Label data for one-vs-rest as binary classifier

Y_Train_ovr1 = np.where(Y_Train == 0, 1, 0)
Y_Test_ovr1  = np.where(Y_Test ==  0, 1, 0)

Y_Train_ovr2 = np.where(Y_Train == 1, 1, 0)
Y_Test_ovr2  = np.where(Y_Test ==  1, 1, 0)

Y_Train_ovr3 = np.where(Y_Train == 2, 1, 0)
Y_Test_ovr3  = np.where(Y_Test ==  2, 1, 0)
```

Extend the analysis using the one-vs-rest strategy for SVM (We are using Linear Kernal to compare) and Perceptron algorithms.

```
# Train the SVM classifier with one-vs-rest strategy

svm_classifier1 = svm.SVC(kernel='linear')
# svm_classifier1 = svm.SVC()
svm_classifier1.fit(X_Train, Y_Train_ovr1)
.
.
# Train the Perceptron classifier with one-vs-rest strategy
.
.
perceptron_classifier3 = Perceptron()
perceptron_classifier3.fit(X_Train, Y_Train_ovr3)
```

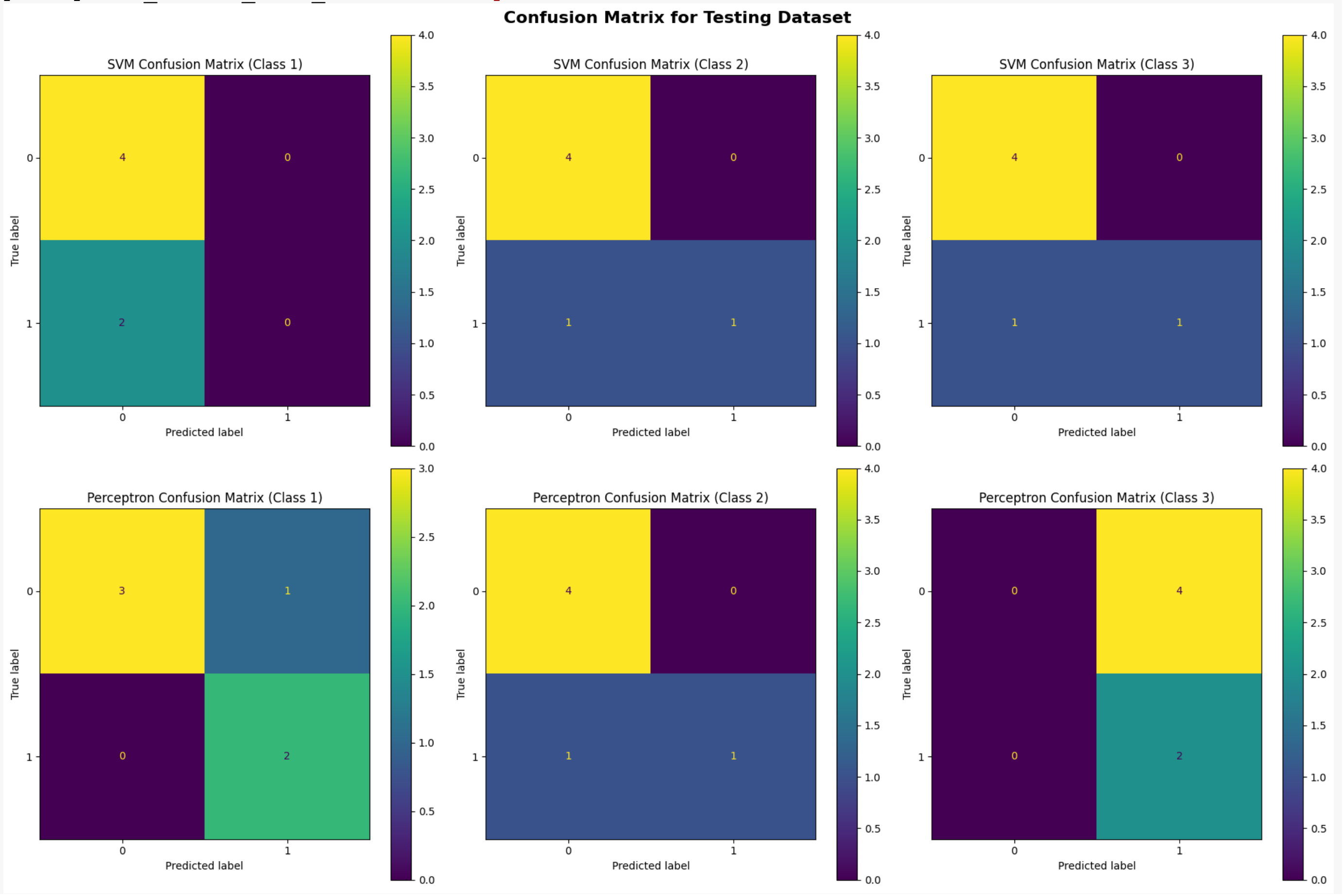
Obtain confusion matrices for training datasets.

```
# Plot the confusion matrix for SVM classifier
svm_cm = ConfusionMatrixDisplay.from_estimator(svm_classifier1, X_Train, Y_Train_ovr1, ax=axs[0, 0])
svm_cm.ax_.set_title('SVM Confusion Matrix (Class 1)')
```



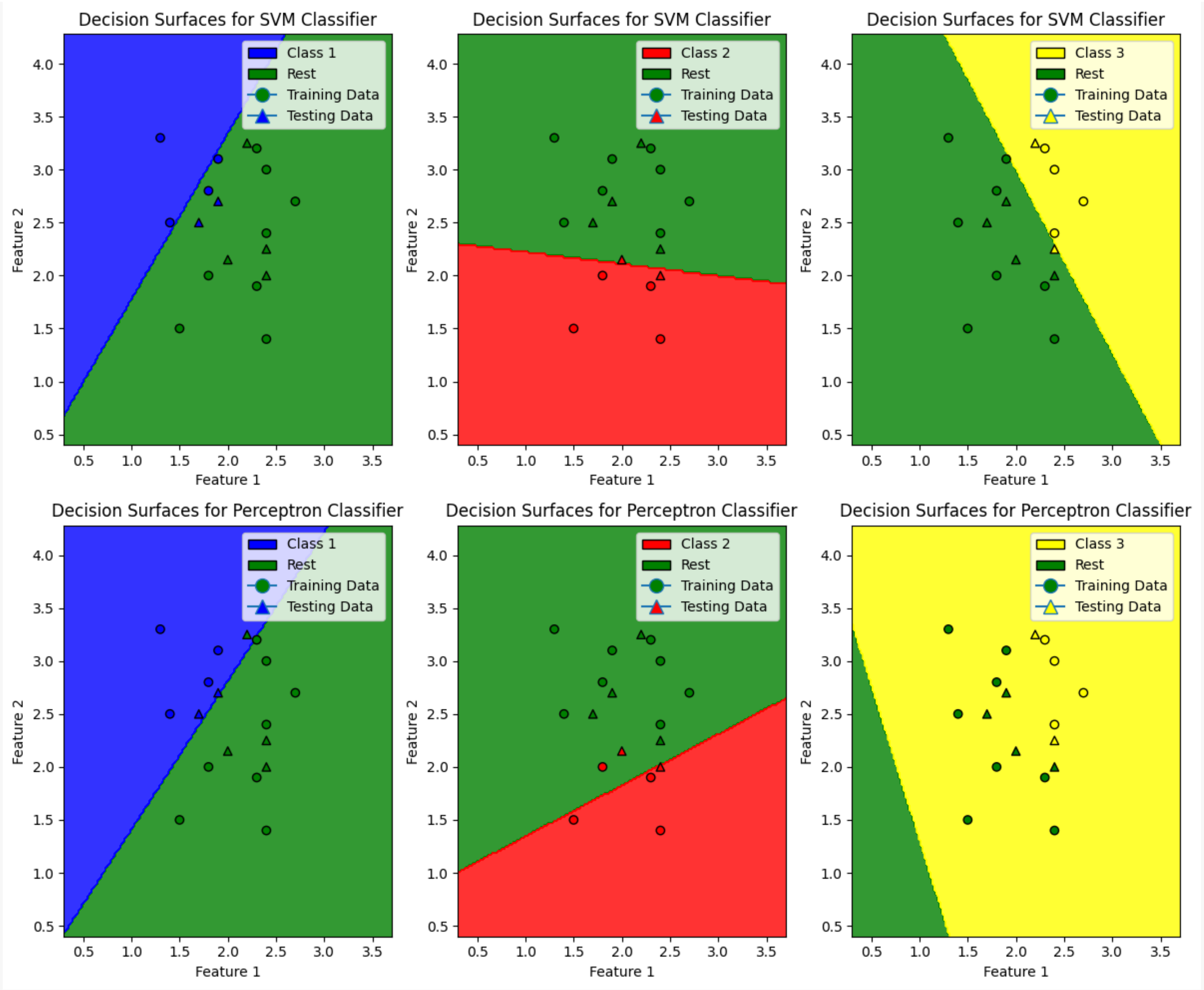
Obtain confusion matrices for testing datasets.

```
# Plot the confusion matrix for Perceptron classifier (third row)
perceptron_cm3 = ConfusionMatrixDisplay.from_estimator(perceptron_classifier3, X_Test, Y_Test_ovr3, ax=axs[1, 2])
perceptron_cm3.ax_.set_title('Perceptron Confusion Matrix (Class 3)')
```



Visualize decision surfaces using the color scheme mentioned earlier.

```
# Make predictions for each model

svm_predictions1 = svm_classifier1.predict(np.c_[xx.ravel(), yy.ravel()])
svm_predictions1 = svm_predictions1.reshape(xx.shape)
.
perceptron_predictions3 = perceptron_classifier3.predict(np.c_[xx.ravel(), yy.ravel()])
perceptron_predictions3 = perceptron_predictions3.reshape(xx.shape)
# Create a custom colormap for visualization others will be green
cmap11 = ListedColormap(['green', 'blue'])
cmap21 = ListedColormap(['green', 'red'])
cmap31 = ListedColormap(['green', 'yellow'])
.
.
# Visualize decision surfaces for Perceptron classifier
axes[1,1].contourf(xx, yy, perceptron_predictions2, alpha=0.8, cmap=cmap21)
axes[1,1].scatter(X_Train[:, 0], X_Train[:, 1], c=Y_Train_ovr2, cmap=cmap21, edgecolors='k',
marker='o', label='Training Data')
axes[1,1].scatter(X_Test[:, 0], X_Test[:, 1], c=Y_Test_ovr2, cmap=cmap21, edgecolors='k',
marker='^', label='Test Data')
axes[1,1].set_xlabel('Feature 1')
axes[1,1].set_ylabel('Feature 2')
axes[1,1].set_title('Decision Surfaces for Perceptron Classifier')

.

.
 # Show the plot
 plt.show()
```



Compare and analyze SVM and Perceptron results.

```
# Predict labels for each classifier using the test data

svm_pred1 = svm_classifier1.predict(X_Test)
# Calculate accuracy scores for each classifier
svm_acc1 = accuracy_score(Y_Test_ovr1, svm_pred1)
print("SVM Results:", svm_results)
print("Perceptron Results:", perceptron_results)
```

```
SVM Results: [0.6667, 0.8334, 0.8334]
Perceptron Results: [0.8334, 0.8334, 0.3333]
```

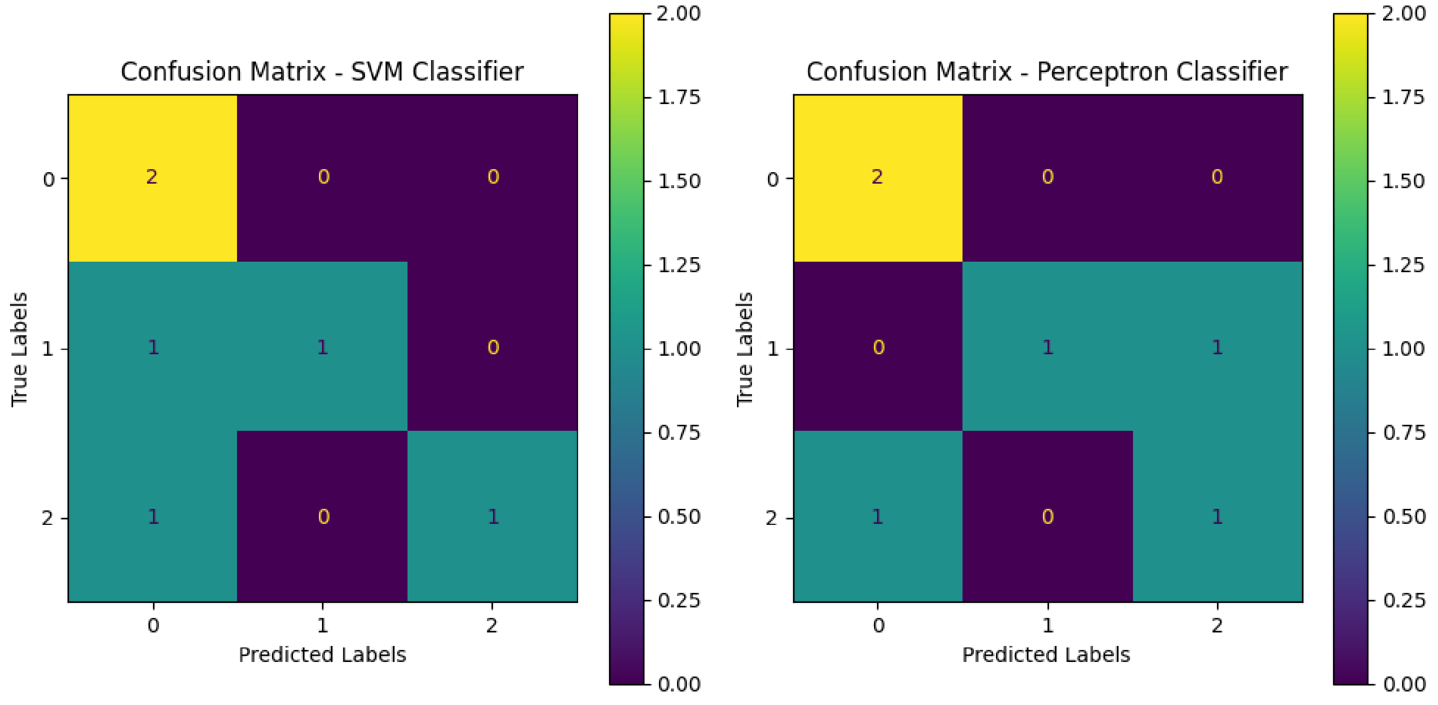
c) (c) Results

Aggregate results from the one-vs-rest strategy for SVM and Perceptron.

```
# Combine the predictions of the three models
def combine_predictions(predictions):
    combined = np.zeros((len(predictions[0]), len(predictions)))
    for i, prediction in enumerate(predictions):
        combined[:, i] = prediction
    combined = np.argmax(combined, axis=1)
    return combined
# Make predictions for each model
svm_predictions1 = svm_classifier1.predict(X_Test)
.
# Make predictions for each model
.
predictions3 = perceptron_classifier3.predict(X_Test)
# Combine the predictions
combined_svm_predictions = combine_predictions([svm_predictions1, svm_predictions2, svm_predictions3])
combined_perceptron_predictions = combine_predictions([predictions1, predictions2, predictions3])
```

Calculate the confusion matrix for the aggregated results.

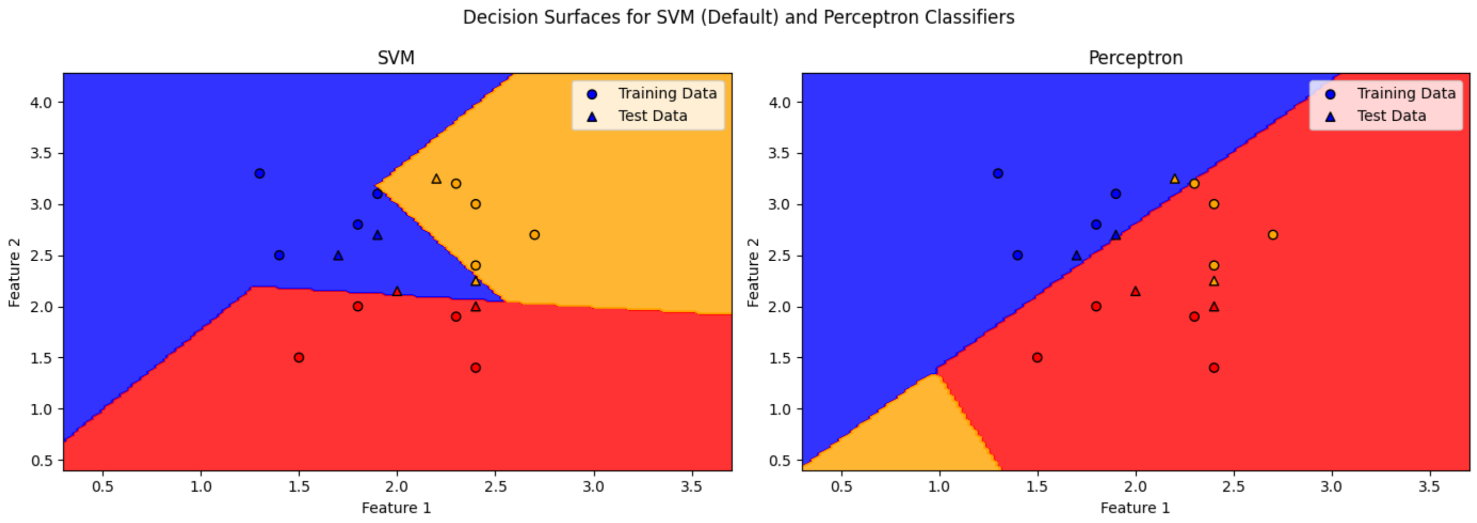
```
# Calculate confusion matrix
svm_confusion_matrix = confusion_matrix(Y_Test, combined_svm_predictions)
# Calculate confusion matrix
perceptron_confusion_matrix = confusion_matrix(Y_Test, combined_perceptron_predictions)

# Plot confusion matrix for Perceptron classifier
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(10, 5))

# Plot confusion matrix for SVM classifier
svm_display = ConfusionMatrixDisplay(confusion_matrix=svm_confusion_matrix)
svm_display.plot(ax=axs[0])
axs[0].set_title("Confusion Matrix - SVM Classifier")
axs[0].set_xlabel("Predicted Labels")
axs[0].set_ylabel("True Labels")

perceptron_display = ConfusionMatrixDisplay(confusion_matrix=perceptron_confusion_matrix)
perceptron_display.plot(ax=axs[1])
axs[1].set_title("Confusion Matrix - Perceptron Classifier")
axs[1].set_xlabel("Predicted Labels")
axs[1].set_ylabel("True Labels")
plt.tight_layout()
plt.show()
```



Visualize the decision surface for the aggregated results

```
# Make predictions for each point in the meshgrid
Zs1 = svm_classifier1.predict(np.c_[xx.ravel(), yy.ravel()])
Zs2 = svm_classifier2.predict(np.c_[xx.ravel(), yy.ravel()])
Zs3 = svm_classifier3.predict(np.c_[xx.ravel(), yy.ravel()])
Zp1 = perceptron_classifier1.predict(np.c_[xx.ravel(), yy.ravel()])
Zp2 = perceptron_classifier2.predict(np.c_[xx.ravel(), yy.ravel()])
Zp3 = perceptron_classifier3.predict(np.c_[xx.ravel(), yy.ravel()])
# combined_perceptron_predictions
combined_svm_predictions_mesh = combine_predictions([Zs1, Zs2, Zs3])
combined_svm_predictions_mesh = combined_svm_predictions_mesh.reshape(xx.shape)
# combined_perceptron_predictions
combined_perceptron_predictions_mesh = combine_predictions([Zp1, Zp3, Zp2])
combined_perceptron_predictions_mesh = combined_perceptron_predictions_mesh.reshape(xx.shape)
# Set up the subplots for SVM classifiers
fig, axs = plt.subplots(1,2, figsize=(14, 5))
# Plot decision surfaces for Perceptron classifier
axs[1].contourf(xx, yy, combined_perceptron_predictions_mesh, alpha=0.8, cmap=cmap)
# Plot decision surfaces for SVM classifier
axs[0].contourf(xx, yy, combined_svm_predictions_mesh, alpha=0.8, cmap=cmap)
```

```
# Scatter plot for training and test data
for i, ax in enumerate(axs.flat):
    ax.scatter(X_Train[:, 0], X_Train[:, 1], c=Y_Train, cmap=cmap, edgecolors='k', marker='o',
label='Training Data')
    ax.scatter(X_Test[:, 0], X_Test[:, 1], c=Y_Test, cmap=cmap, edgecolors='k', marker='^',
label='Test Data')
    ax.legend()
    ax.set_xlabel('Feature 1')
    ax.set_ylabel('Feature 2')
# Set titles for each subplot
axs[0].set_title('SVM')
axs[1].set_title('Perceptron')
# Set overall title for the figure
fig.suptitle('Decision Surfaces for SVM (Default) and Perceptron Classifiers')
# Adjust spacing between subplots
plt.tight_layout()
# Show the plot
plt.show()
```



Analyze performance.

```
# Calculate accuracy score
svm_accuracy = accuracy_score(Y_Test, combined_svm_predictions)
print("Combined SVM Accuracy:", svm_accuracy)
# Calculate accuracy score
perceptron_accuracy = accuracy_score(Y_Test, combined_perceptron_predictions)
print("Combined Perceptron Accuracy:", perceptron_accuracy)
```

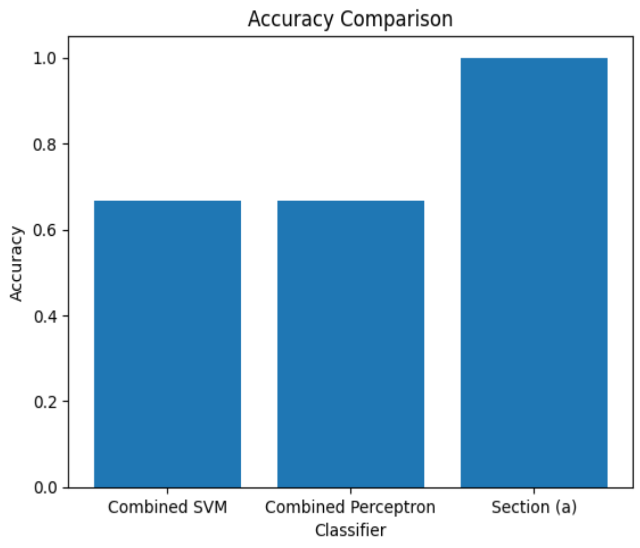
```
Combined SVM Accuracy: 0.6666666666666666
Combined Perceptron Accuracy: 0.6666666666666666
```

Compare performance with section (a) solution.

```
print("Combined SVM Accuracy:", svm_accuracy)
print("Combined Perceptron Accuracy:", perceptron_accuracy)

#section (a) accuracy
predictions = classifier.predict(X_Test)
svm_a_accuracy = accuracy_score(Y_Test, predictions)

print("Section (a)  svm Accuracy:", svm_a_accuracy)
```

```
Combined SVM Accuracy: 0.6666666666666666
Combined Perceptron Accuracy: 0.6666666666666666
Section (a) svm Accuracy: 1.0
```

```
# Bar plot
classifiers = ['Combined SVM', 'Combined Perceptron',
'Section (a)']
plt.bar(classifiers, accuracy_values)
plt.title('Accuracy Comparison')
plt.show()
```



The observation

While using svm with default kernel we got testing accuracies for classifier [1, 2, 3] as [1, 1, 0.83] respectively. We also tried to change the kernel value to 'linear' and we got testing accuracies for classifier [1,2,3] as [0.66,0.83,0.83] respectively. On the other hand, while using the perceptron classifier we got the testing accuracies for classifier [1,2,3] as [0.83, 0.83, 0.33]. So, it turns out that using the svm with the default kernel is the best choice for our case.

d) conclusion

Determine the reason why SVM performance in section (a) is different than aggregated performance of SVM in section (c)?

The default SVM implementation and the one-vs-all (OvA) strategy are different in terms of how they handle multiclass classification problems. In the default SVM implementation, the classifier directly learns a decision boundary to separate the classes, while the OvA strategy transforms a multiclass problem into multiple binary classification subproblems. The training approach involves creating binary datasets by relabeling the samples and training individual classifiers for each class and finally aggregate the results.

Refine the default SVM by selecting the appropriate parameter.

Train the SVM model with selected parameters and evaluate its performance.

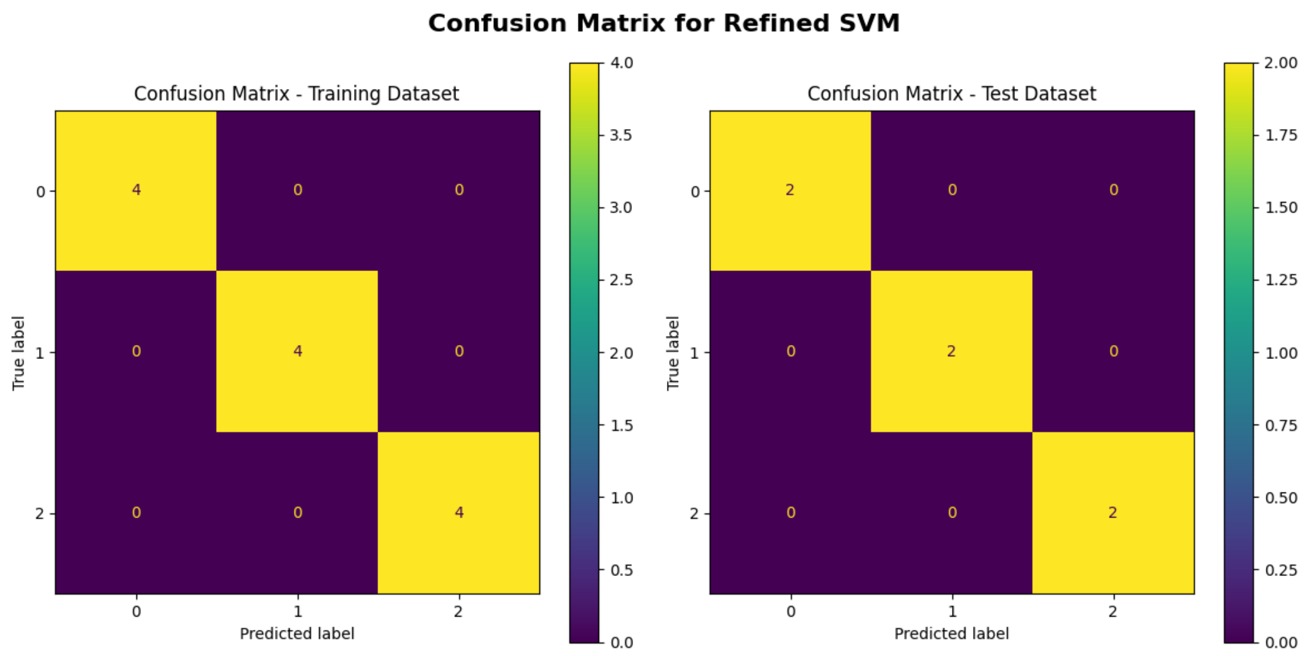
```
# Create the SVM model with desired parameters
svm_model = svm.SVC(C=0.1, kernel='linear', gamma='scale')
# Train the SVM model
svm_model.fit(X_Train, Y_Train)
# Make predictions on the test set
predictions = svm_model.predict(X_Test)
# Evaluate the model's accuracy
accuracy = accuracy_score(Y_Test, predictions)
print("Accuracy:", accuracy)
```

```
Accuracy: 1.0
```

Why we select those parameters.

As we tried some trials with different parameters but this was the least computing ones and giving the same results. The parameters were chosen to provide similar results while minimizing computational complexity. This balance between accuracy and computational complexity can be beneficial when working with large datasets or limited computational resources. The default parameter values provided by the SVM implementation may have been chosen to provide reasonable performance across a wide range of problems.

Obtain confusion matrices for multi-class classification.



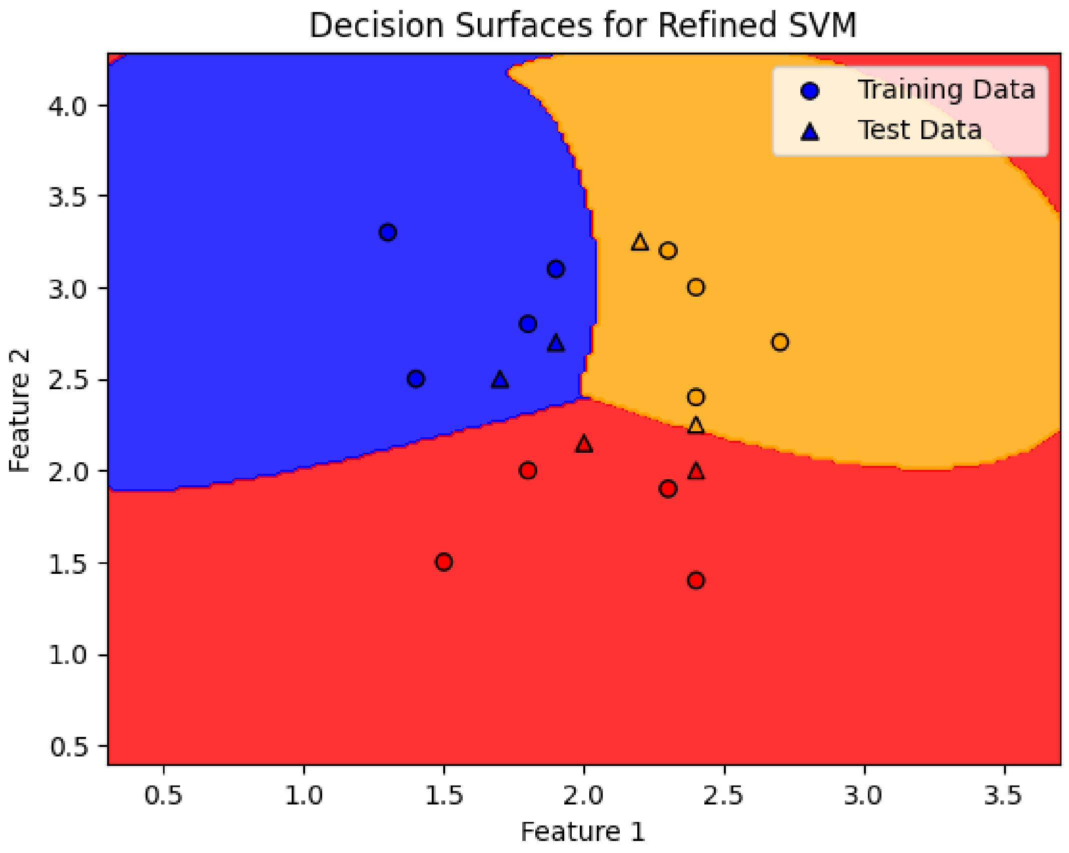
Training Dataset:

True \ Predicted	0	1	2
0	4	0	0
1	0	4	0
2	0	0	4

Test Dataset:

True \ Predicted	0	1	2
0	2	0	0
1	0	2	0
2	0	0	2

Obtain decision surfaces for multi-class classification.



Compare results with the default SVM and discuss the impact of parameter selection.

As a comparison with section (a)

After changing the parameters, we had a slightly better model performance, but both had 100% accuracy. But the main advantage is to use the optimal method as saving computing power and faster results. Optimizing the parameters of a machine learning model can lead to improved performance and efficiency. In this case, changing the parameters resulted in a slightly better model performance while maintaining 100% accuracy. The optimal method, such as the one-vs-rest approach for SVM, can benefit from its computational efficiency and faster results. However, it is important to consider the possibility of overfitting, especially if the model has a high complexity or as the dataset is small. Evaluating the model on additional unseen data or using cross-validation techniques can provide a more robust assessment of its performance. Overall, by selecting the optimal method and fine-tuning the parameters, you have improved the efficiency and performance of your model, leading to more effective and practical solutions.

2- KNN classifier

a) Data Preprocessing

Firstly, you need to shuffle the dataset and split the dataset into a training set with 1000 samples and a validation set with 300 samples and a testing set with 428 samples.

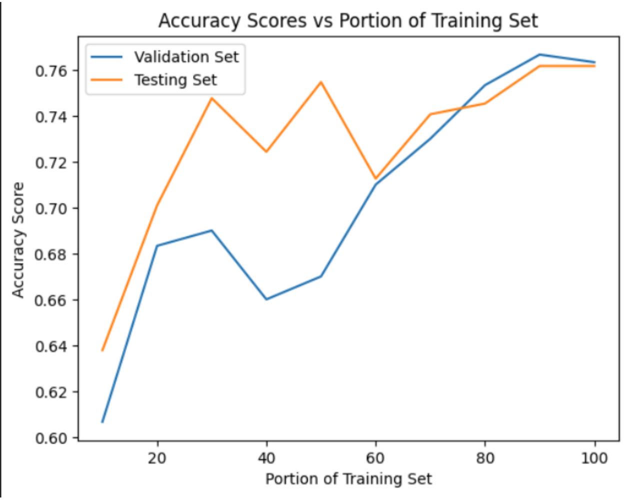
```
data = pd.read_csv('car.data',header=None)
header = pd.Index(['buying', 'maint', 'doors','persons','lug boot','safety','decision'])
shuffled_df = data.sample(frac=1).reset_index(drop=True)
x = shuffled_df.drop('decision',axis=1)
y=shuffled_df['decision']
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size=(428/1728),random_state=42)
x_train,x_val,y_train,y_val = train_test_split(x_train,y_train,test_size=(300/1300),random_state=42)
```

b) We need to transform the string values into numbers.

```
labeler = LabelEncoder()
x_train = x_train.apply(labeler.fit_transform)
y_train = labeler.fit_transform(y_train)
x_test = x_test.apply(labeler.fit_transform)
y_test = labeler.fit_transform(y_test)
x_val = x_val.apply(labeler.fit_transform)
y_val = labeler.fit_transform(y_val)
```

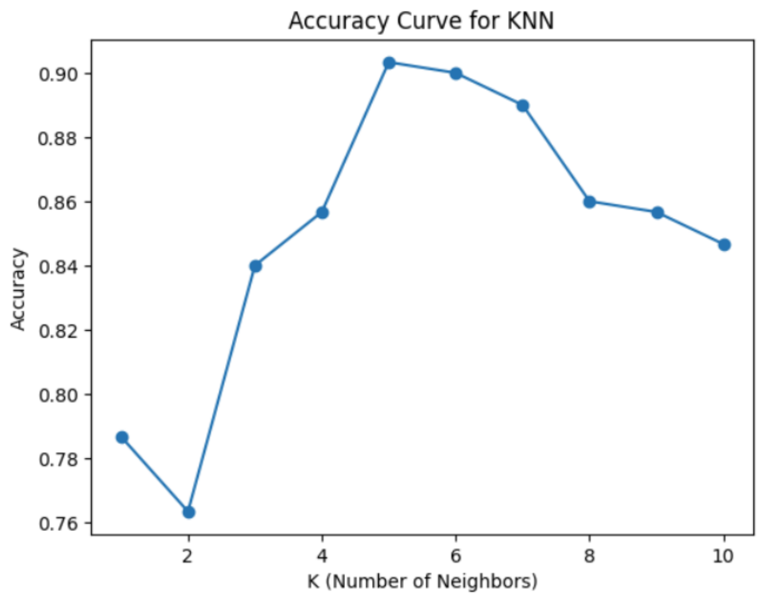
c) use different number of training samples to show the impact of number of training samples.

```
knn_models = ['knn1','knn2','knn3','knn4','knn5','knn6','knn7','knn8','knn9','knn10']
val_accuracy_scores = []
test_accuracy_scores = []
train_portions = []
for i in range(0,len(knn_models)-1):
  knn_models[i] = KNeighborsClassifier(n_neighbors=2)
  X,_,Y,_ = train_test_split(x_train,y_train,train_size =((i+1)/10),random_state=42)
  knn_models[i].fit(X,Y)
  y_pred_val = knn_models[i].predict(x_val)
  y_pred_test = knn_models[i].predict(x_test)
  val_accuracy_scores.append(accuracy_score(y_val,y_pred_val))
  test_accuracy_scores.append(accuracy_score(y_test,y_pred_test))
  train_portions.append(((i + 1) * 10))
knn_models[9] = KNeighborsClassifier(n_neighbors=2)
knn_models[9].fit(x_train,y_train)
y_pred_val = knn_models[9].predict(x_val)
y_pred_test = knn_models[9].predict(x_test)
val_accuracy_scores.append(accuracy_score(y_val,y_pred_val))
test_accuracy_scores.append(accuracy_score(y_test,y_pred_test))
train_portions.append(100)
plt.plot(train_portions, val_accuracy_scores, label='Validation Set')
plt.plot(train_portions, test_accuracy_scores, label='Testing Set')
plt.xlabel('Portion of Training Set')
plt.ylabel('Accuracy Score')
plt.title('Accuracy Scores vs Portion of Training Set')
plt.legend()
plt.show()
```



d) Best K

```
accuracy_scores = []
k_values = []
for i in range(1,11):
  knn = KNeighborsClassifier(n_neighbors=i)
  knn.fit(x_train, y_train)
  y_pred = knn.predict(x_val)
  accuracy = accuracy_score(y_val, y_pred)
  k_values.append(i)
  accuracy_scores.append(accuracy)

# Plot the accuracy curve
plt.plot(k_values, accuracy_scores, marker='o')
plt.xlabel('K (Number of Neighbors)')
plt.ylabel('Accuracy')
plt.title('Accuracy Curve for KNN')
plt.show()
```



e) Conclusion

Provide your conclusions from the experiments of question (c) and (d) in this question.

C)

- Increasing the portion of training set increases the performance as both the validation and testing accuracy scores increased.
- The validation set accuracy provided us with the estimation of our model generalization as the model will perform prediction on unseen data.
- The testing set accuracy confirms the model's performance as the model will perform prediction on completely unseen data.
- The plot shows that the testing set accuracy and the validation set accuracy follows the same trend.

D)

- The accuracy curve shows how the performance of the KNN models varies with different values of K. In this case, the highest accuracy is achieved when K is around 5.
- It was clear from this experiment that when K is even number and it always higher for odd number, so it was expected to have the best K to be odd.