Part 1

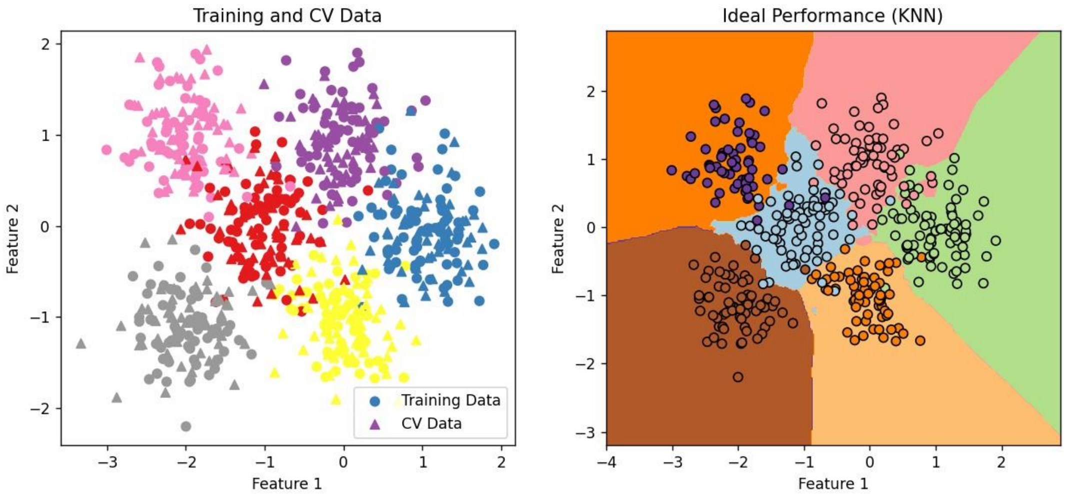
- We compare, “training data”, “cross-validation” and “ideal data”

```python
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

classes = 6
m = 800
std = 0.4
centers = np.array([[-1, 0], [1, 0], [0, 1], [0, -1], [-2, 1], [-2, -1]])
X, y = make_blobs(n_samples=m, centers=centers, cluster_std=std,
random_state=2, n_features=2)

X_train, X_, y_train, y_ = train_test_split(X, y, test_size=0.50,
random_state=1)
X_cv, X_test, y_cv, y_test = train_test_split(X_, y_, test_size=0.20,
random_state=1)
# Function to plot data and decision boundaries
def plot_decision_boundary(X, y, model, title):
    h = 0.02  # step size in the mesh
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)

    # Plot also the training points
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap=plt.cm.Paired)
    plt.title(title)
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.show()

# Training and cross-validation plot
plt.figure(figsize=(12, 5))

# Plotting training data
plt.subplot(1, 2, 1)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.Set1, label='Training Data')
plt.scatter(X_cv[:, 0], X_cv[:, 1], c=y_cv, cmap=plt.cm.Set1, marker='^', label='CV Data')
plt.title('Training and CV Data')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()

# Ideal performance plot
plt.subplot(1, 2, 2)
# Fit a model on the entire training data
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, y_train)

# Plot decision boundaries
plot_decision_boundary(X_train, y_train, model, 'Ideal Performance (KNN)')

# Calculate accuracy on cross-validation data
cv_accuracy = accuracy_score(y_cv, model.predict(X_cv))
print(f'Cross-validation accuracy: {cv_accuracy:.2f}')

plt.tight_layout()
plt.show()
```

- Here is the training accuracy:

```
17    X_cv, X_test, y_cv, y_test = train_test_split(X_, y_, test_size=0.20,
18    random_state=1)

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\ardah\UCM> python -u "c:\Users\ardah\UCM\p6_b\p6_b.py"
Cross-validation accuracy: 0.94
PS C:\Users\ardah\UCM>
```

- Here are the graphs:



Part 2

- We develop a complex model

```python
classes = 6
m = 800
std = 0.4
centers = np.array([[-1, 0], [1, 0], [0, 1], [0, -1], [-2, 1], [-2, -1]])
X, y = make_blobs(n_samples=m, centers=centers, cluster_std=std,
                  random_state=2, n_features=2)
# Split dataset into training, cross validation, and test sets
X_train, X_tmp, y_train, y_tmp = train_test_split(X, y, test_size=0.50,
                                                  random_state=1)
X_cv, X_test, y_cv, y_test = train_test_split(X_tmp, y_tmp, test_size=0.20,
                                              random_state=1)
# Convert numpy arrays to PyTorch tensors
X_train = torch.FloatTensor(X_train)
y_train = torch.LongTensor(y_train)
X_cv = torch.FloatTensor(X_cv)
y_cv = torch.LongTensor(y_cv)
# Define the complex neural network model
class ComplexModel(nn.Module):
    def __init__(self):
        super(ComplexModel, self).__init__()
        self.fc1 = nn.Linear(2, 120)
        self.fc2 = nn.Linear(120, 40)
        self.fc3 = nn.Linear(40, 6)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
# Initialize the complex model
model_complex = ComplexModel()

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_complex.parameters(), lr=0.001)
# Training loop
epochs = 1000
for epoch in range(epochs):
    optimizer.zero_grad()
    outputs = model_complex(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()

    if (epoch+1) % 100 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
# Evaluate the complex model on cross-validation data
with torch.no_grad():
    outputs_cv = model_complex(X_cv)
    _, predicted_cv = torch.max(outputs_cv, 1)
    cv_accuracy = accuracy_score(y_cv, predicted_cv)

print(f'Complex Model - Cross-validation accuracy: {cv_accuracy:.2%}')
# Function to plot decision boundaries of the model
def plot_decision_boundary(model, X, y, title):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                         np.arange(y_min, y_max, 0.1))
    # Create input for the model
    grid_tensor = torch.FloatTensor(np.c_[xx.ravel(), yy.ravel()])
    Z = model(grid_tensor)
    _, Z = torch.max(Z, 1)
    Z = Z.reshape(xx.shape)
    # Create a color plot
    cmap = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF', '#FFD700', '#8A2BE2', '#FF6984'])
    plt.contourf(xx, yy, Z, cmap=cmap, alpha=0.8)
    # Plot training points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap, edgecolors='k')
    plt.title(title)
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.show()
# Convert the model to evaluation mode
model_complex.eval()
# Plot decision boundaries of the complex model on the training + CV data
X_combined = torch.FloatTensor(np.vstack((X_train.numpy(), X_cv.numpy())))
y_combined = torch.LongTensor(np.concatenate((y_train.numpy(), y_cv.numpy())))
plot_decision_boundary(model_complex, X_combined, y_combined, 'Complex Model Decision Boundaries')
```

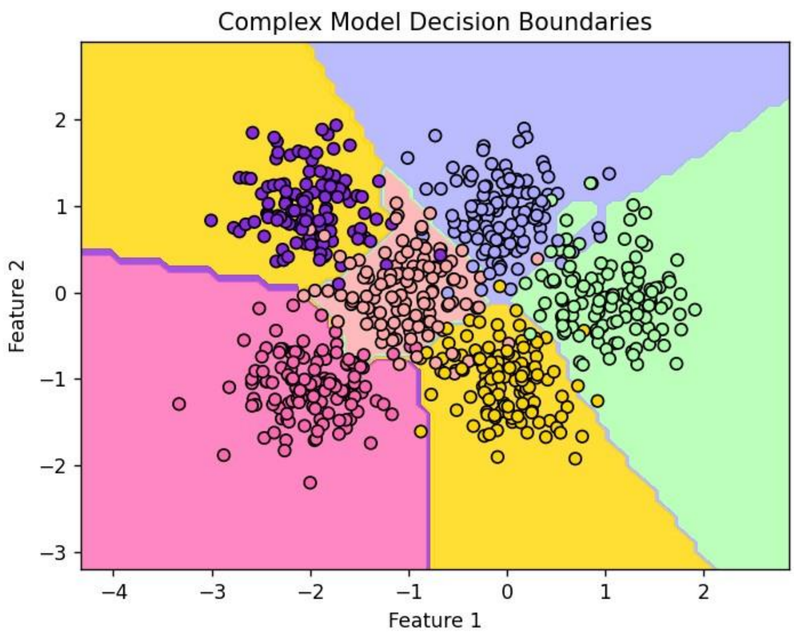
- Here is the cross-validation accuracy

```python
# Function to plot decision boundaries of the model
def plot_decision_boundary(model, X, y, title):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                         np.arange(y_min, y_max, 0.1))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\ardah\UCM> python -u "c:\Users\ardah\UCM\p6_b\p6_b.py"
Cross-validation accuracy: 0.94
PS C:\Users\ardah\UCM> python -u "c:\Users\ardah\UCM\p6_b\p6_b2.py"
Epoch [100/1000], Loss: 0.3264
Epoch [200/1000], Loss: 0.2036
Epoch [300/1000], Loss: 0.1867
Epoch [400/1000], Loss: 0.1777
Epoch [500/1000], Loss: 0.1699
Epoch [600/1000], Loss: 0.1624
Epoch [700/1000], Loss: 0.1541
Epoch [800/1000], Loss: 0.1453
Epoch [900/1000], Loss: 0.1362
Epoch [1000/1000], Loss: 0.1266
Complex Model - Cross-validation accuracy: 90.62%
PS C:\Users\ardah\UCM>
```

- Here is the plotted graph:



Part 3

- We will now try to develop a simple model

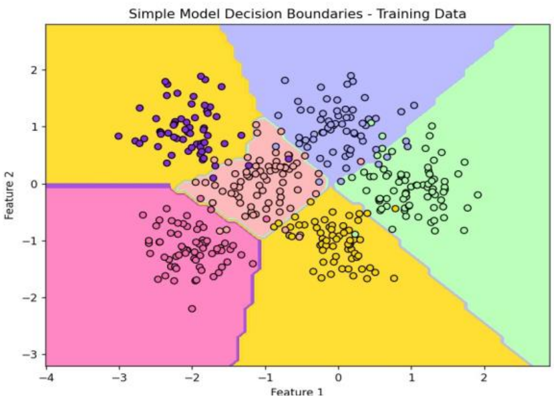
```
classes = 6
m = 800
std = 0.4
centers = np.array([[-1, 0], [1, 0], [0, 1], [0, -1], [-2, 1], [-2, -1]])
X, y = make_blobs(n_samples=m, centers=centers, cluster_std=std,
                  random_state=2, n_features=2)
# Split dataset into training, cross-validation, and test sets
X_train, X_tmp, y_train, y_tmp = train_test_split(X, y, test_size=0.50,
                                                  random_state=1)
X_cv, X_test, y_cv, y_test = train_test_split(X_tmp, y_tmp, test_size=0.20,
                                              random_state=1)
# Convert numpy arrays to PyTorch tensors
X_train = torch.FloatTensor(X_train)
y_train = torch.LongTensor(y_train)
X_cv = torch.FloatTensor(X_cv)
y_cv = torch.LongTensor(y_cv)
# Define the simple neural network model
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc1 = nn.Linear(2, 6)
        self.fc2 = nn.Linear(6, 6)
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
# Initialize the simple model
model_simple = SimpleModel()
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_simple.parameters(), lr=0.01)
# Training loop
epochs = 1000
for epoch in range(epochs):
    optimizer.zero_grad()
    outputs = model_simple(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()
    if (epoch+1) % 100 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
# Evaluate the simple model on training and cross-validation data
with torch.no_grad():
    outputs_train = model_simple(X_train)
    _, predicted_train = torch.max(outputs_train, 1)
    train_accuracy = accuracy_score(y_train, predicted_train)
    outputs_cv = model_simple(X_cv)
    _, predicted_cv = torch.max(outputs_cv, 1)
    cv_accuracy = accuracy_score(y_cv, predicted_cv)
print(f'Simple Model - Training accuracy: {train_accuracy:.2%}')
print(f'Simple Model - Cross-validation accuracy: {cv_accuracy:.2%}')
# Function to plot decision boundaries of the model on specific data
def plot_decision_boundary(model, X, y, title):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                         np.arange(y_min, y_max, 0.1))
    # Create input for the model
    grid_tensor = torch.FloatTensor(np.c_[xx.ravel(), yy.ravel()])
    Z = model(grid_tensor)
    _, Z = torch.max(Z, 1)
    Z = Z.reshape(xx.shape)

    # Create a color plot
    cmap = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF', '#FFD700', '#8A2BE2', '#FF69B4'])
    plt.contourf(xx, yy, Z, cmap=cmap, alpha=0.8)

    # Plot data points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap, edgecolors='k')
    plt.title(title)
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.show()
# Convert the model to evaluation mode
model_simple.eval()
# Plot decision boundaries of the simple model on the training data
plt.figure(figsize=(8, 6))
plot_decision_boundary(model_simple, X_train.numpy(), y_train.numpy(), 'Simple Model Decision Boundaries - Training Data')
# Plot decision boundaries of the simple model on the cross-validation data
plt.figure(figsize=(8, 6))
plot_decision_boundary(model_simple, X_cv.numpy(), y_cv.numpy(), 'Simple Model Decision Boundaries - Cross-Validation Data')
```

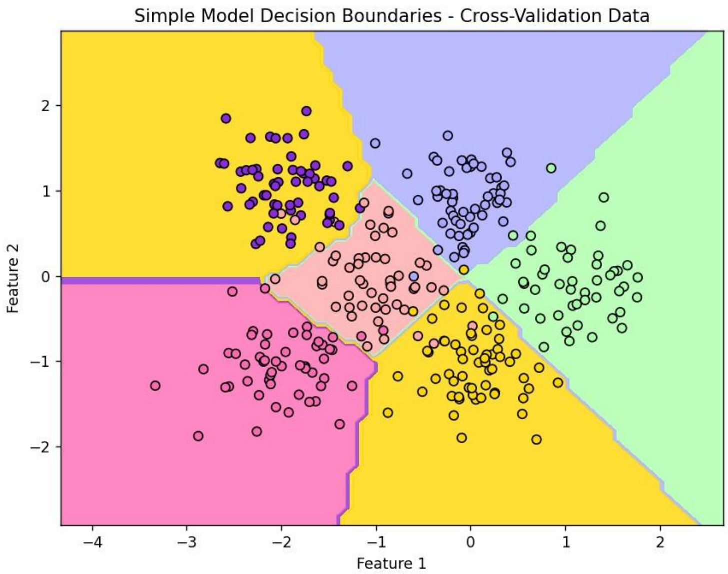
- Here is the output ad accuracy rate

```
print(f'Simple Model - Training accuracy: {train_accuracy:.2%}')
```

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

```
PS C:\Users\ardah\UCM> python -u "c:\Users\ardah\UCM\p6_b\p6_b3_1.py"
Epoch [100/1000], Loss: 0.4075
Epoch [200/1000], Loss: 0.2248
Epoch [300/1000], Loss: 0.2019
Epoch [400/1000], Loss: 0.1962
Epoch [500/1000], Loss: 0.1936
Epoch [600/1000], Loss: 0.1919
Epoch [700/1000], Loss: 0.1905
Epoch [800/1000], Loss: 0.1888
Epoch [900/1000], Loss: 0.1876
Epoch [1000/1000], Loss: 0.1866
Simple Model - Training accuracy: 92.50%
Simple Model - Cross-validation accuracy: 94.06%
PS C:\Users\ardah\UCM>
```





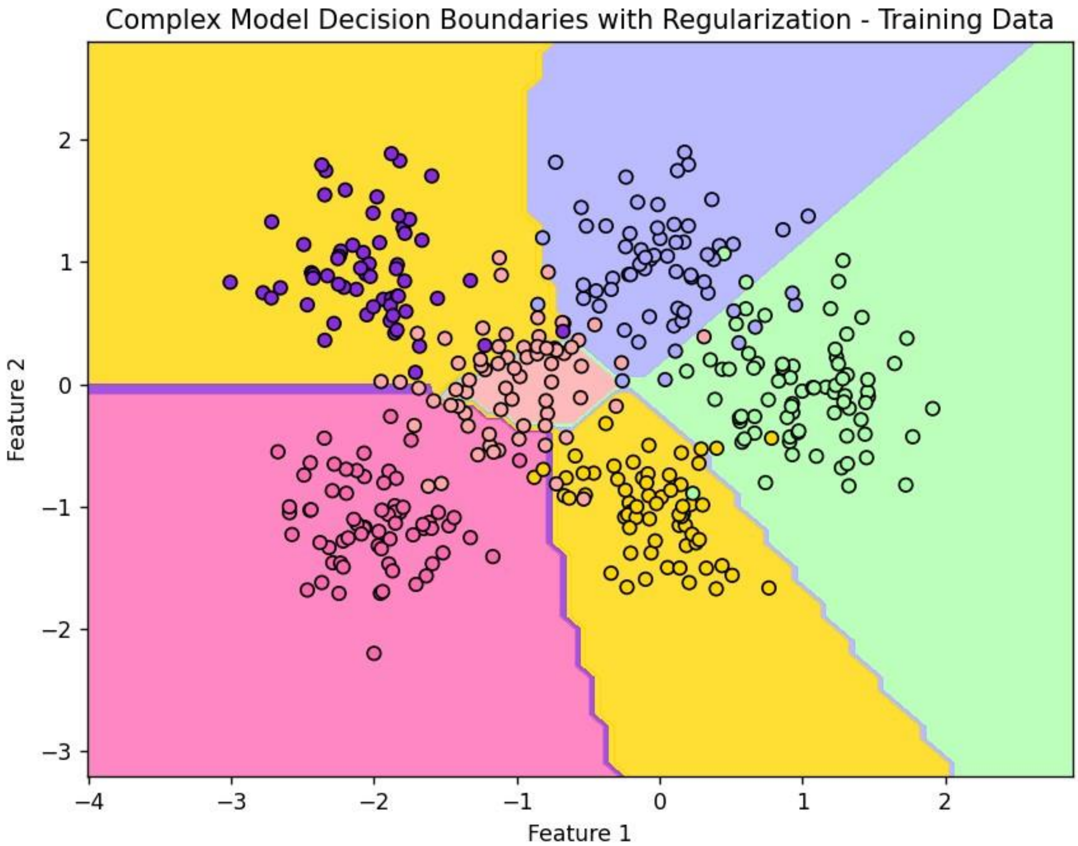
Part 4

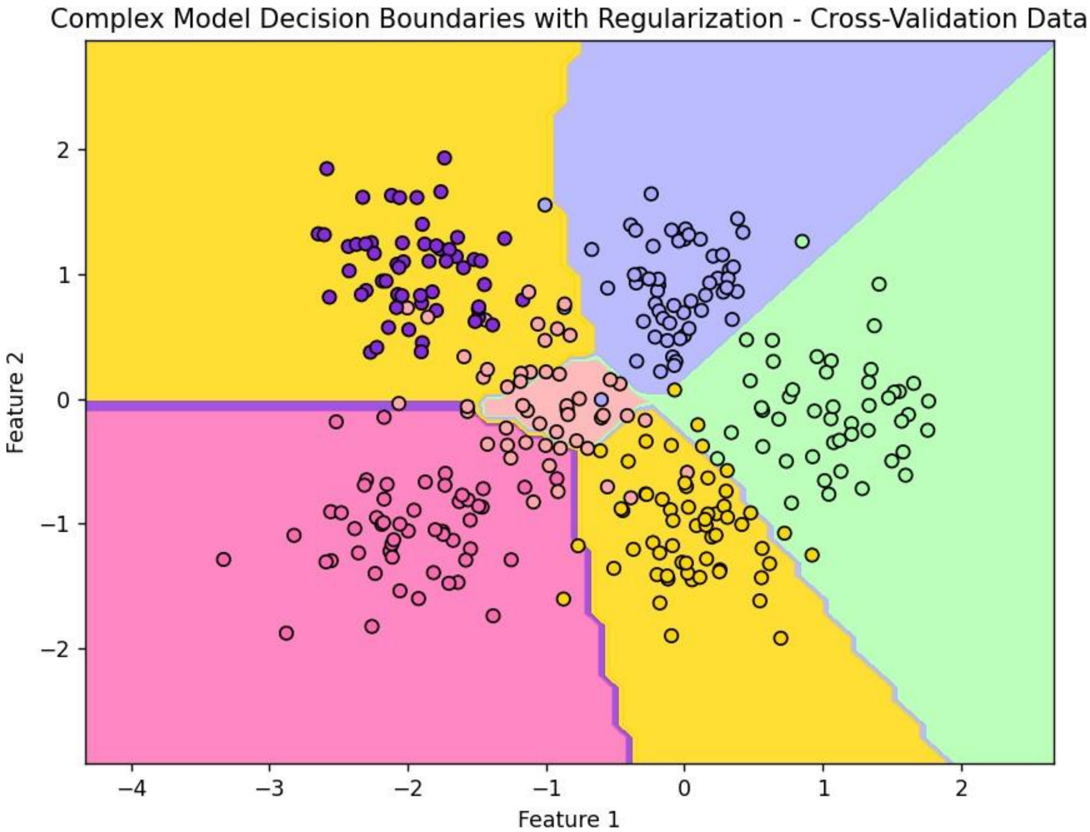
- We will regularize the complex model

```python
# Generate artificial dataset
classes = 6
m = 800
std = 0.4
centers = np.array([[-1, 0], [1, 0], [0, 1], [0, -1], [-2, 1], [-2, -1]])
X, y = make_blobs(n_samples=m, centers=centers, cluster_std=std,
                  random_state=2, n_features=2)
# Split dataset into training, cross-validation, and test sets
X_train, X_tmp, y_train, y_tmp = train_test_split(X, y, test_size=0.50,
                                                  random_state=1)
X_cv, X_test, y_cv, y_test = train_test_split(X_tmp, y_tmp, test_size=0.20,
                                              random_state=1)
# Convert numpy arrays to PyTorch tensors
X_train = torch.FloatTensor(X_train)
y_train = torch.LongTensor(y_train)
X_cv = torch.FloatTensor(X_cv)
y_cv = torch.LongTensor(y_cv)
# Define the complex neural network model with regularization
class ComplexModelRegularized(nn.Module):
    def __init__(self):
        super(ComplexModelRegularized, self).__init__()
        self.fc1 = nn.Linear(2, 120)
        self.fc2 = nn.Linear(120, 40)
        self.fc3 = nn.Linear(40, 6)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
# Initialize the complex model with regularization
model_complex_regularized = ComplexModelRegularized()
# Define loss function and optimizer with L2 regularization (weight decay)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_complex_regularized.parameters(), lr=0.001, weight_decay=0.1)
# Training loop
epochs = 1000
for epoch in range(epochs):
    optimizer.zero_grad()
    outputs = model_complex_regularized(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()
    if (epoch+1) % 100 == 0:
        print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')
# Evaluate the complex model with regularization on cross-validation data
with torch.no_grad():
    outputs_cv = model_complex_regularized(X_cv)
    _, predicted_cv = torch.max(outputs_cv, 1)
    cv_accuracy = accuracy_score(y_cv, predicted_cv)
print(f'Complex Model with Regularization - Cross-validation accuracy: {cv_accuracy:.2%}')
# Function to plot decision boundaries of the model on specific data
def plot_decision_boundary(model, X, y, title):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                         np.arange(y_min, y_max, 0.1))
    # Create input for the model
    grid_tensor = torch.FloatTensor(np.c_[xx.ravel(), yy.ravel()])
    Z = model(grid_tensor)
    _, Z = torch.max(Z, 1)
    Z = Z.reshape(xx.shape)
    # Create a color plot
    cmap = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF', '#FFD700', '#8A2BE2', '#FF69B4'])
    plt.contourf(xx, yy, Z, cmap=cmap, alpha=0.8)
    # Plot data points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap, edgecolors='k')
    plt.title(title)
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.show()
# Convert the model to evaluation mode
model_complex_regularized.eval()
# Plot decision boundaries of the complex model on the training data
plt.figure(figsize=(8, 6))
plot_decision_boundary(model_complex_regularized, X_train.numpy(), y_train.numpy(), 'Complex Model Decision Boundaries with Regularization - Training Data')
# Plot decison boundaries of the complex model on the cross-validation data
plt.figure(figsize=(8, 6))
plot_decision_boundary(model_complex_regularized, X_cv.numpy(), y_cv.numpy(), 'Complex Model Decision Boundaries with Regularization - Cross-Validation Data')
```

- Accuracy rate seems way lower than it should be… I wasnt able to solve this issue

```
Regularization Value: 0.3, Cross-validation Error: 0.8531, Training Error: 0.812
PS C:\Users\ardah\UCM> python -u "c:\Users\ardah\UCM\p6_b\p6_b3_2.py"
Epoch [100/1000], Loss: 0.7264
Epoch [200/1000], Loss: 0.6461
Epoch [300/1000], Loss: 0.6676
Epoch [400/1000], Loss: 0.7018
Epoch [500/1000], Loss: 0.7309
Epoch [600/1000], Loss: 0.7505
Epoch [700/1000], Loss: 0.7605
Epoch [800/1000], Loss: 0.7655
Epoch [900/1000], Loss: 0.7677
Epoch [1000/1000], Loss: 0.7687
Complex Model with Regularization - Cross-validation accuracy: 85.62%
PS C:\Users\ardah\UCM>
```




Complex Model Decision Boundaries with Regularization - Cross-Validation Data
Feature 2
Feature 1
2
1
0
−1
−2
−4
−3
−2
−1
0
1
2

Part 5

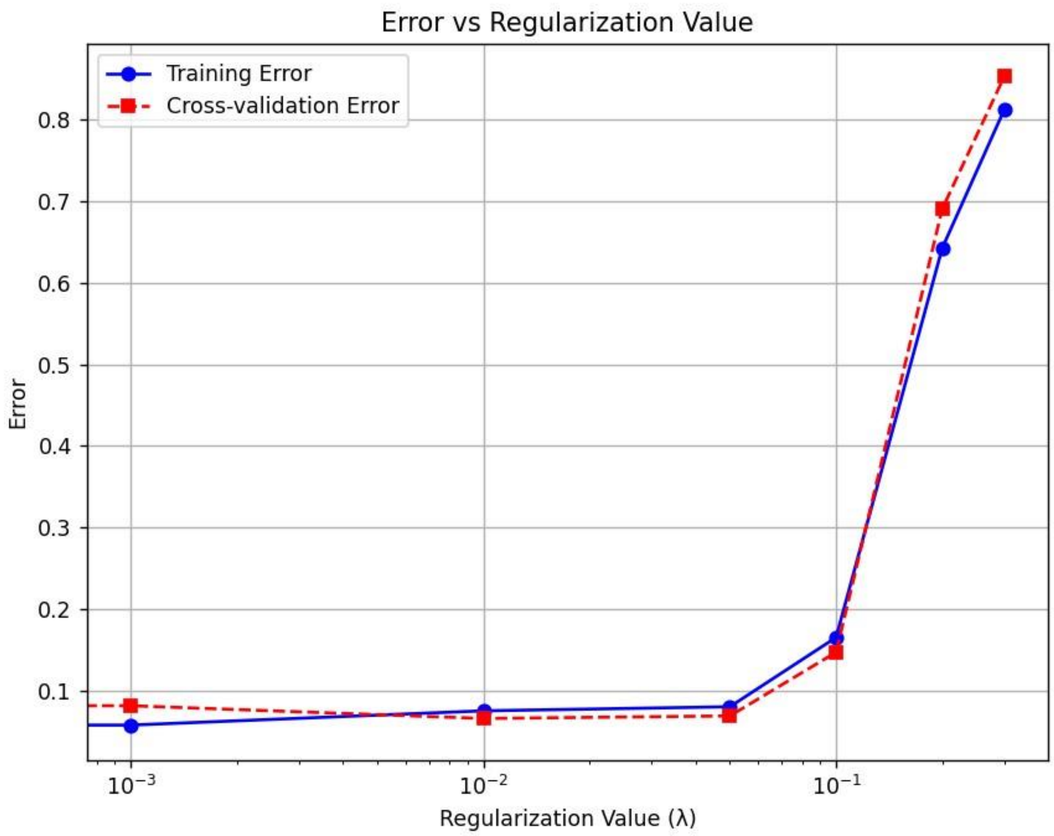
- We need to find the most optimal lambda by trying different lambdas

```python
classes = 6
m = 800
std = 0.4
centers = np.array([[-1, 0], [1, 0], [0, 1], [0, -1], [-2, 1], [-2, -1]])
X, y = make_blobs(n_samples=m, centers=centers, cluster_std=std,
                  random_state=2, n_features=2)
# Split dataset into training, cross-validation, and test sets
X_train, X_tmp, y_train, y_tmp = train_test_split(X, y, test_size=0.50,
                                                  random_state=1)
X_cv, X_test, y_cv, y_test = train_test_split(X_tmp, y_tmp, test_size=0.20,
                                              random_state=1)
# Convert numpy arrays to PyTorch tensors
X_train = torch.FloatTensor(X_train)
y_train = torch.LongTensor(y_train)
X_cv = torch.FloatTensor(X_cv)
y_cv = torch.LongTensor(y_cv)
# Define the complex neural network model
class ComplexModel(nn.Module):
    def __init__(self):
        super(ComplexModel, self).__init__()
        self.fc1 = nn.Linear(2, 120)
        self.fc2 = nn.Linear(120, 40)
        self.fc3 = nn.Linear(40, 6)
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
# Function to train and evaluate the model with specified regularization value
def train_and_evaluate_model(regularization_value):
    # Initialize the complex model
    model = ComplexModel()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=regularization_value)  # L2 regularization
    # Training loop
    epochs = 1000
    train_errors = []
    cv_errors = []
    for epoch in range(epochs):
        optimizer.zero_grad()
        outputs = model(X_train)
        loss = criterion(outputs, y_train)
        loss.backward()
        optimizer.step()
        # Evaluate cross-validation error
        with torch.no_grad():
            outputs_cv = model(X_cv)
            _, predicted_cv = torch.max(outputs_cv, 1)
            cv_accuracy = accuracy_score(y_cv, predicted_cv)
            cv_error = 1.0 - cv_accuracy  # Error rate
            cv_errors.append(cv_error)
    # Final cross-validation error after all epochs
    final_cv_error = cv_errors[-1]
    # Evaluate final training error after all epochs
    with torch.no_grad():
        outputs_train = model(X_train)
        _, predicted_train = torch.max(outputs_train, 1)
        train_accuracy = accuracy_score(y_train, predicted_train)
        train_error = 1.0 - train_accuracy  # Error rate
    return final_cv_error, train_error
# List of regularization values to try
regularization_values = [0.0, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3]
# Lists to store cross-validation errors and training errors for each regularization value
cv_errors = []
train_errors = []
# Iterate over regularization values and evaluate model performance
for reg_value in regularization_values:
    cv_error, train_error = train_and_evaluate_model(reg_value)
    cv_errors.append(cv_error)
    train_errors.append(train_error)
    print(f'Regularization Value: {reg_value}, Cross-validation Error: {cv_error:.4f}, Training Error: {train_error:.4f}')
# Plotting "Training Error" and "Cross-validation Error" vs "Regularization Value (λ)"
plt.figure(figsize=(8, 6))
plt.plot(regularization_values, train_errors, marker='o', linestyle='-', color='b', label='Training Error')
plt.plot(regularization_values, cv_errors, marker='s', linestyle='--', color='r', label='Cross-validation Error')
plt.title('Error vs Regularization Value')
plt.xlabel('Regularization Value (λ)')
plt.ylabel('Error')
plt.xscale('log')  # Use logarithmic scale for the x-axis
plt.grid(True)
plt.legend()
plt.show()
```

- For me “0.1” seemed to be the most optimal, which contradicts with the actual results

```
PS C:\Users\ardah\UCM> python -u "c:\Users\ardah\UCM\p6_b\p6_b5.py"
Regularization Value: 0.0, Cross-validation Error: 0.0813, Training Error: 0.0550
Regularization Value: 0.001, Cross-validation Error: 0.0813, Training Error: 0.0575
Regularization Value: 0.01, Cross-validation Error: 0.0656, Training Error: 0.0750
Regularization Value: 0.05, Cross-validation Error: 0.0687, Training Error: 0.0800
Regularization Value: 0.1, Cross-validation Error: 0.1469, Training Error: 0.1650
Regularization Value: 0.2, Cross-validation Error: 0.6906, Training Error: 0.6425
Regularization Value: 0.3, Cross-validation Error: 0.8531, Training Error: 0.8125
PS C:\Users\ardah\UCM>
```

- This is the graph I obtained:



Part 6

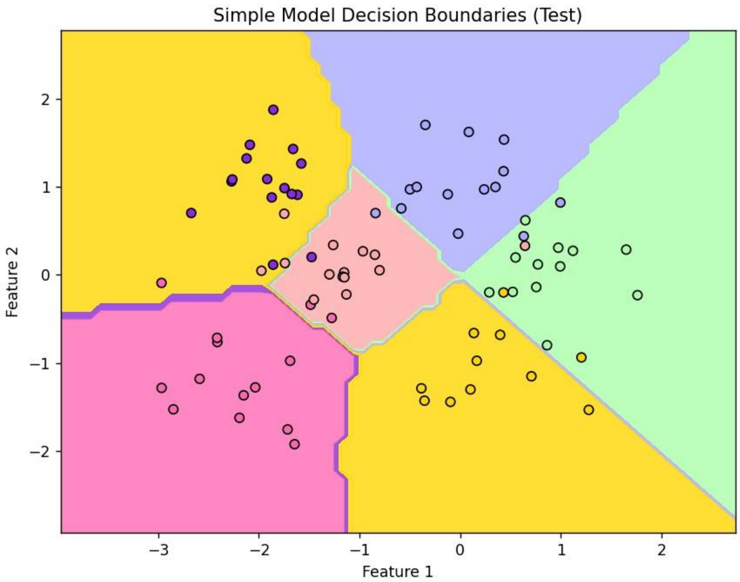
- We will use the X_test to test the models…

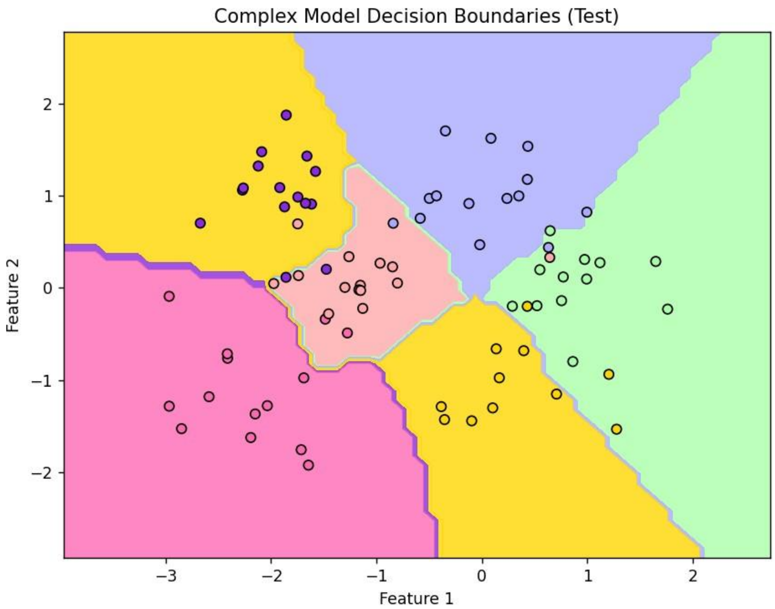
```python
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from matplotlib.colors import ListedColormap
from sklearn.neighbors import KNeighborsClassifier

# Generate artificial dataset
classes = 6
m = 800
std = 0.4
centers = np.array([[-1, 0], [1, 0], [0, 1], [0, -1], [-2, 1], [-2, -1]])
X, y = make_blobs(n_samples=m, centers=centers, cluster_std=std,
                  random_state=2, n_features=2)
# Split dataset into training, cross-validation, and test sets
X_train, X_tmp, y_train, y_tmp = train_test_split(X, y, test_size=0.50,
                                                  random_state=1)
X_cv, X_test, y_cv, y_test = train_test_split(X_tmp, y_tmp, test_size=0.20,
                                              random_state=1)
# Convert numpy arrays to PyTorch tensors
X_train = torch.FloatTensor(X_train)
y_train = torch.LongTensor(y_train)
X_cv = torch.FloatTensor(X_cv)
y_cv = torch.LongTensor(y_cv)
X_test = torch.FloatTensor(X_test)
y_test = torch.LongTensor(y_test)
# Define the simple neural network model
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc1 = nn.Linear(2, 6)
        self.fc2 = nn.Linear(6, 6)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
# Define the complex neural network model
class ComplexModel(nn.Module):
    def __init__(self):
        super(ComplexModel, self).__init__()
        self.fc1 = nn.Linear(2, 120)
        self.fc2 = nn.Linear(120, 40)
        self.fc3 = nn.Linear(40, 6)
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
# Function to train the model with specified regularization value
def train_model(model, X_train, y_train):
    # Define loss function and optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    # Training loop
    epochs = 1000
    for epoch in range(epochs):
        optimizer.zero_grad()
        outputs = model(X_train)
        loss = criterion(outputs, y_train)
        loss.backward()
        optimizer.step()
# Train the simple model
model_simple = SimpleModel()
train_model(model_simple, X_train, y_train)
```

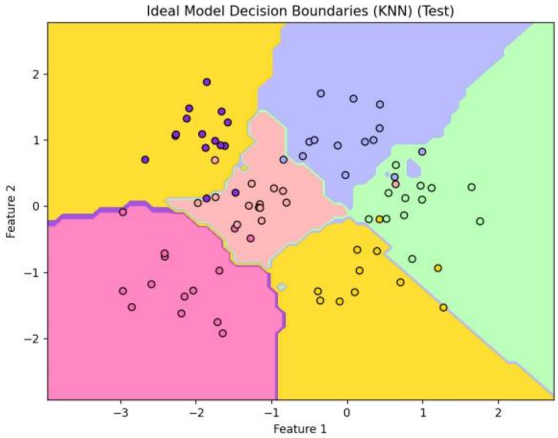
```python
        optimizer.step()
# Train the simple model
model_simple = SimpleModel()
train_model(model_simple, X_train, y_train)
# Train the complex model
model_complex = ComplexModel()
train_model(model_complex, X_train, y_train)
# Evaluate model performance
def evaluate_model(model, X, y, dataset_name):
    with torch.no_grad():
        outputs = model(X)
        _, predicted = torch.max(outputs, 1)
        accuracy = accuracy_score(y.numpy(), predicted.numpy())
        print(f"{dataset_name} Accuracy: {accuracy:.2%}")
        return accuracy
# Evaluate simple model on test set
test_accuracy_simple = evaluate_model(model_simple, X_test, y_test, "Simple Model (Test)")
# Evaluate complex model on test set
test_accuracy_complex = evaluate_model(model_complex, X_test, y_test, "Complex Model (Test)")
# Evaluate ideal model (KNN-based) on test data
model_ideal = KNeighborsClassifier(n_neighbors=5)
model_ideal.fit(X_train.numpy(), y_train.numpy())
test_accuracy_ideal = accuracy_score(y_test, model_ideal.predict(X_test.numpy()))
print(f"Ideal Model (Test) Accuracy: {test_accuracy_ideal:.2%}")
# Plotting decision boundaries for simple model
def plot_decision_boundary(model, X, y, title):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                         np.arange(y_min, y_max, 0.1))
    # Create input for the model
    grid_tensor = torch.FloatTensor(np.c_[xx.ravel(), yy.ravel()])
    Z = model(grid_tensor)
    _, Z = torch.max(Z, 1)
    Z = Z.reshape(xx.shape)
    # Create a color plot
    cmap = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF', '#FFD700', '#8A2BE2', '#FF69B4'])
    plt.contourf(xx, yy, Z, cmap=cmap, alpha=0.8)
    # Plot test points
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap, edgecolors='k')
    plt.title(title)
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.show()
# Plotting decision boundaries for simple model on test set
plt.figure(figsize=(8, 6))
plot_decision_boundary(model_simple, X_test, y_test, 'Simple Model Decision Boundaries (Test)')
# Plotting decision boundaries for complex model on test set
plt.figure(figsize=(8, 6))
plot_decision_boundary(model_complex, X_test, y_test, 'Complex Model Decision Boundaries (Test)')
# Plotting decision boundaries for ideal model (KNN-based) on test data
plt.figure(figsize=(8, 6))
cmap = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF', '#FFD700', '#8A2BE2', '#FF69B4'])
x_min, x_max = X_test[:, 0].min() - 1, X_test[:, 0].max() + 1
y_min, y_max = X_test[:, 1].min() - 1, X_test[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))
Z = model_ideal.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=cmap, alpha=0.8)
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cmap, edgecolors='k')
plt.title('Ideal Model Decision Boundaries (KNN) (Test)')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

- These are the accuracy rates…

```
PS C:\Users\ardah\UCM> python -u "c:\Users\ardah\UCM\p6_b\p6_b6.py"
Simple Model (Test) Accuracy: 82.50%
Complex Model (Test) Accuracy: 86.25%
Ideal Model (Test) Accuracy: 86.25%
```


Simple Model Decision Boundaries (Test)
Feature 2
Feature 1
2
1
0
−1
−2
−3
−2
−1
0
1
2


Complex Model Decision Boundaries (Test)
Feature 2
Feature 1
2
1
0
−1
−2
−3
−2
−1
0
1
2


Ideal Model Decision Boundaries (KNN) (Test)
Feature 2
Feature 1
2
1
0
−1
−2
−3
−2
−1
0
1
2