Final Project of Multiclass Classification on Iris Dataset

Group Members: Hyuk Ahn, Chuiyang Kong, Jincheng Yang

Github Repo: https://github.com/hyukahn16/data2060-final-project

This report is divided into two main sections.

1. Background Information: the first section provides an overview of the foundational concepts behind multiclass classification, including the One-vs-All and All-Pairs algorithms. It also explains the stochastic gradient descent algorithm and the underlying mathematical principles.

2. Code Implementation and Evaluation: the second section focuses on the code. It is further divided into three parts:

- The implementation of the model.
- Unit tests designed to validate the model.
- A comparison of the reproduced results on the Iris dataset with those obtained using Scikit-learn models.

1. Overviews

1.1 Overview of Multiclass Classification: One-vs-All and All-Pairs

One-vs-All

This algorithm creates a classifier for each class (3 classifiers if there exists 3 classes). For each classifier, it is responsible for predicting whether an input belongs to its corresponding class or not.

Each classifier is trained on the entire dataset with modifications corresponding to each classifier.
The modification changes the dataset such that when you're training a classifier for class 3, labels for all the other classes are modified to 0 and labels for the classifier's class are modified to 1. (More details will be provided in the Representation section)

All-Pairs

This algorithm creates a classifier for each pair of classes. For each classifier, it is responsible for predicting whether a given input belongs to one class or the other.

Each classifier is trained on a portion of the dataset with modifications corresponding to each classifier.
First, each classifier is assigned portion of the dataset that contains the classes the classifier is predicting for. Then, the assigned data's classes are changed so that one class is assigned the label of 1 and the other is assigned the label of 0.

Advantages and Disadvantages of Multiclass Classification

Multiclass classification algorithm is an algorithm that classifies an input that can belong to one of the multiple classes (more than two classes).
For this project, we will be implementing One-vs-All and All-Pairs algorithms for the multiclass classification of the UCI Iris dataset.

Compared to a multiclass classification algorithm that inherently encompasses multiclass classification (output of model predicts multiclass), the main advantages of One-vs-All and All-Pairs stems from the use of binary classifiers.
Because of the binary classifiers to represent multiclass classification, these two algorithms have implementation simplicity and easy interpretability of the predictions.

Unfortunately, the disadvantages also stem from the use of binary classifiers.

- The binary classifiers do not have any knowledge that it is used for multiclass classification and therefore, does not have inherent understanding of the multiclass classification problem.
- Due to training classifier for each class, each classifier is trained on a class imbalanced dataset and may result in overfitting.
- Training multiple classifiers can be computationally expensive.

1.2 Representation

Binary Logistic Regression

Given sample's feature values $x \in \mathbb{R}^d$ and a label $y \in \{0, 1\}$, binary logistic regression classifier takes input x and calculates the logit of the input x through affine function:

$$y = \langle w, x \rangle$$

To predict the class of the input, the logit is passed through a sigmoid function to obtain a value between 0 and 1 (probability that input x belongs to class 1):

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

If the value is less than or equal to 0.5, it belongs to class 0. If the value is greater than 0.5, it belongs to class 1.

Therefore, our hypothesis function defined on weights w is

$$h_w(x) = \frac{1}{1 + e^{-\langle w, x \rangle}}$$

Multiclass Logistic Regression

Now, using the binary logistic regression defined above, we will define one-vs-all and all-pairs multiclass logistic regression algorithms that can classify input x into multiple classes.

Algorithm Pseudocode for One-vs-All (Shalev-Shwartz and Ben-David, 2014)

For one-vs-all, the algorithm utilizes *k* binary classifiers (logistic regression in our case), where each classifier predicts whether the true class of x belongs in their corresponding class or not.

```
Given Inputs:
  training set $S = (x_1, y_1), \ldots, (x_m, y_m)$
  class set $C$ = (0, 1, ..., $k$), where $k$ == number of classes
  binary classifier - logistic regression $L$

$\text{foreach } i \in C:$
  $\text{Create dataset } S_i = (x_1, 1_{[y_1 = i]}), \ldots, (x_m, 1_{[y_m = i]})$
  $\text{Train } h_i = L(S_i)$

Outputs:
  $h(x) \in argmax_{i \in Y}\ h_i(x)$
```

Algorithm Pseudocode for All-Pairs (Shalev-Shwartz and Ben-David, 2014)

Given inputs:

training set $S = (x_1, y_1), \ldots, (x_m, y_m)$
class set C = (0, 1, ..., k), where k == number of classes
binary classifier - logistic regression L

```
foreach $i, j \in C$ such that $i < j$
  Initialize empty dataset $S_{i,j}$
  for $t = 1, \ldots, m$
      If $y_t = i$, then add $(x_t, 1)$ to $S_{i,j}$
      If $y_t = j$, then add $(x_t, 0)$ to $S_{i,j}$
  Train $h_{i,j} = L(S_{i,j})$
```

Outputs:

$h(x) \in argmax_{i \in Y}\ (\Sigma_{j \in Y}\ \text{sign}(j - i) h_{i,j}(x))$

1.3 Loss: Logistic Loss

The loss function of a Logistic Regression classifier over k classes is the **log-loss**, also called **cross-entropy loss**. Since we will only use binary classifier, e.g. Binary Logistic Regression, in this project, only **Binary Log Loss** will be introduced in this section. To be noted, regularization is not used in our project.

The Binary Log Loss on a sample of m data points, also called the Binary Cross Entropy Loss, is:

$$L(h) = -\frac{1}{m} \sum_{i=1}^{m} (y_i \log h(x_i) + (1 - y_i) \log(1 - h(x_i)))$$

(Scikit-learn. (2024) *sklearn.metrics.log_loss)*

1.4 Optimizer

One-vs-All and **All-Pairs** are algorithms used to solve muticlass classification problems through binary classifiers.
To optimize the binary classifiers, we look to Stochastic Gradient Descent (Mini Batch) algorithm.

Given loss function L and learning rate α, the gradient descent equation for weight w_j update is

$$w_j = w_j - \alpha \cdot \frac{\partial L}{\partial w_j}$$

For each batch of size m, the gradient of the binary log loss L (given in the previous section) with respect to weight w_j is

$$\frac{\partial L}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} (h(x_i) - y_i) \cdot x_{ij}$$

Therefore, our stochastic gradient update is

$$w_j = w_j - \alpha \cdot \left(\frac{1}{m} \sum_{i=1}^{m} (h(x_i) - y_i) \cdot x_{ij} \right)$$

Pseudocode: Stochastic Gradient Descent for Logistic Regression (Shalev-Shwartz and Ben-David, 2014)

```
**Given inputs**:
  Traning examples $S$, step size $\alpha$, batch size $b < |S|$
**initialize:** $\mathbf{w} = 0$
**for** $t = 1, 2, \ldots, T$:
  Randomly shuffle $S$
  **for** $i = 0$ to $|S|/b - 1$:
    $S'$ = Extracted current batch using $i$
    $\mathbf{w} = \mathbf{w} - \alpha \cdot \nabla L_{S'}(h_w)$ + regularization
return
```

2. Codes

2.1 Model

In [1]:
```python
import numpy as np
from sklearn.linear_model import SGDClassifier
```

```
# Testing models using SGDClassifier from Scikit-learn (2024)

def sigmoid(x):
    '''
        Sigmoid function f(x) =  1/(1 + exp(-x))
        :param x: A scalar or Numpy array
        :return: Sigmoid function evaluated at x (applied element-wise if it is an array)
    '''
    return np.where(x > 0, 1 / (1 + np.exp(-x)), np.exp(x) / (np.exp(x) + np.exp(0)))

def get_estimator(train_epochs, lr):
    """
        helper function to get SGDClassifier from Sklearn.
        Other parameters except for the number of max iterations and learning rate
        have already been set in SGDClassifier.
    """
    """
        Since shuffling in SKlearn is different from that in our custom logistic regression
        model, here we turn the shuffling off.

        We also employ a log_loss and constant learning rate in our SGDClassfier.

        The reason that we choose SGDClassifier instead of the logistic regression model in sklearn
        is that it employs SGD as its optimizor while the logistic regression model does not.
    """
    estimator = SGDClassifier(
        loss='log_loss',
        tol=None,
        max_iter=train_epochs,
        shuffle=False,
        # shuffle=True,
        random_state=0,
        learning_rate='constant',
        eta0=lr,
        alpha=0)
    return estimator
```

2.1.1 Model: Representation - Logistic Regression

In [2]:
```
import numpy as np
#Modified Logistic regression taken from HW3 of data2060.

class MyLogisticRegression:
    '''
    Binary Logistic Regression that learns weights using
    stochastic gradient descent.
    '''
    def __init__(self, batch_size=1, num_epochs=1, lr=0.0001):
        '''
        Initializes a LogisticRegression classifer.
        @attrs:
            n_features: the number of features in the classification problem
            n_classes: the number of classes in the classification problem
            weights: The weights of the Logistic Regression model
            alpha: The learning rate used in stochastic gradient descent
        '''
        self.learning_rate = lr
        self.num_epochs = num_epochs
        self.batch_size = batch_size
        self.weights = None

    def train(self, X, Y):
        '''
        Train the model, using batch stochastic gradient descent
        @params:
            X: a 2D Numpy array where each row contains an example, padded by 1 column for the bias
            Y: a 1D Numpy array containing the corresponding labels for each example
        @return:
            None
        '''
        num_samples, num_features = X.shape

        self.weights = np.zeros((1, num_features))

        for epoch in range(self.num_epochs):

            shuffled_X = X
            shuffled_Y = Y
            """ if you want to turn on the shuffling, just uncomment codes below to replace the codes above
            shuffled_inds = np.random.permutation(num_samples)
            shuffled_X = X[shuffled_inds]
            shuffled_Y = Y[shuffled_inds]
            """

            for start in range(0, num_samples, self.batch_size):
                end = start + self.batch_size
                X_batch = shuffled_X [start: min(end, num_samples)]
                Y_batch = shuffled_Y [start: min(end, num_samples)]

                predictions = sigmoid(np.dot(X_batch, self.weights.T)) # num_samples * 1 (num_classes)
                Y_batch = np.reshape(Y_batch,(len(Y_batch),1)) # num_samples * 1, reshape Y to same dimens
                error = predictions - Y_batch
                loss_grad = np.dot(error.T, X_batch)/len(X_batch)

                self.weights -= self.learning_rate * loss_grad


    def loss(self, X, Y):
        '''
        Computes the logistic loss (binary cross-entropy loss) for binary classification
        @params:
            X: 2D Numpy array where each row contains an example, padded by 1 column for the bias
            Y: 1D Numpy array containing the corresponding labels for each example
        @return:
            A float number which is the average loss of the model on the dataset
        '''
        # Clip predictions to prevent log(0)
        y_pred = self.predict(X)
        y_pred = np.clip(y_pred, 1e-15, 1 - 1e-15)

        left_half = Y.T @ np.log(y_pred)
        right_half = (1-Y).T @ np.log(1-y_pred)

        # Calculate the logistic loss
        loss = -np.mean(left_half + right_half)
        return loss


    def predict(self, X):
        '''
        Compute predictions based on the learned parameters and examples X
        @params:
            X: a 2D Numpy array where each row contains an example, padded by 1 column for the bias
        @return:
            A 1D Numpy array with one element for each row in X containing the predicted class.
        '''
        # X.shape: (batch size, num features)
        # self.weights.shape: (1, num features)
        dot_product = np.dot(self.weights, X.T) # n_classes * n_samples
        probs = sigmoid(dot_product)
        probsall = np.vstack((1-probs, probs)) # probs are for class 2
        y_predict = np.argmax(probsall, axis=0) #finding the index of the max value in a column
        return y_predict


    def accuracy(self, X, Y):
        '''
        Output the accuracy of the trained model on a given testing dataset X and labels Y.
        @params:
            X: a 2D Numpy array where each row contains an example, padded by 1 column for the bias
            Y: a 1D Numpy array containing the corresponding labels for each example
        @return:
            a float number indicating accuracy (between 0 and 1)
        '''
        predicted_classes = self.predict(X)
        return np.mean(predicted_classes == Y)

    def predict_proba(self, X):
        '''
        Compute probabilities for the input data X.
        @params:
        X: A 2D Numpy array where each row contains an example
        @return:
        Probabilities of each example being in class 1
        '''
        dot_product = np.dot(self.weights, X.T) # n_classes * n_samples
        probs = sigmoid(dot_product)

        return probs
```

2.1.2 Model: one-vs-all

In [3]:
```python
import numpy as np

class OnevsAll:
    def __init__(self, n_classes, batch_size=1, epochs=1, lr=0.01):
        self.n_classes = n_classes
        self.lr = lr
        self.batch_size = batch_size
        self.epochs = epochs
        # self.conv_threshold = conv_threshold

    def train(self, X, Y):
        # Split data and train each representation
        self.S_Y = np.array([np.array(Y) for _ in range(self.n_classes)])
        self.h = np.array([
            MyLogisticRegression(
                batch_size=self.batch_size,
                num_epochs=self.epochs,
                lr=self.lr
                ) for _ in range(self.n_classes)])

        self.conv_epochs = [0] * self.n_classes
        for cls in range(self.n_classes):
            # Create S_i for each class i
            S_Y_i = self.S_Y[cls]
            cls_idx = S_Y_i == cls
            S_Y_i[cls_idx] = 1
            non_cls_idx = np.logical_not(cls_idx)
            S_Y_i[non_cls_idx] = 0

            # Train h_i for each class i on S_i
            h_i = self.h[cls]
            conv_epoch = h_i.train(X, S_Y_i)
            self.conv_epochs.append(conv_epoch)

    """
    Loss function is similar to accuracy function.
    As a result, we choose to only keep accuracy function.
    """
    # def loss(self, X, Y):
    #     preds = self.predict(X)
    #     # L1-loss
    #     losses = np.abs(Y-preds)
    #     losses = np.sum(losses)
    #     return losses

    def predict(self, X):
        # h_i in argmax h_i(x)
        predictions = [0] * X.shape[0]
        for i, x in enumerate(X):
            preds = [0] * self.n_classes
            # Get predictions from all hypotheses
            for c in range(self.n_classes):
                preds[c] = self.h[c].predict_proba(x)
            # Select max prediction
            predictions[i] = np.argmax(preds)

        return predictions

    def accuracy(self, X, Y):
        predictions = self.predict(X)
        return np.mean(predictions == Y)
```

2.1.3 Model: all-pairs

In [4]:
```
# You can only use python and numpy in this section.
import numpy as np

class AllPairs:
    def __init__(self, n_classes, batch_size, epochs, lr):
        self.n_classes = n_classes
        self.batch_size = batch_size
        self.epochs = epochs

        self.models = {}
        self.lr = lr

    def train(self, X, Y):
        if X.shape[0] == 0 or Y.shape[0] == 0:
            raise ValueError("No data provided")

        for i in range(self.n_classes):
            for j in range(i + 1, self.n_classes):
                selected_indices = []
                for index, label in enumerate(Y):
                    if label == i or label == j:
                        selected_indices.append(index)

                X_selected = X[selected_indices]
                Y_selected = Y[selected_indices]

                for idx in range(len(Y_selected)):
                    if Y_selected[idx] == i:
                        Y_selected[idx] = 0
                    else:
                        Y_selected[idx] = 1

                model = MyLogisticRegression(batch_size=self.batch_size, num_epochs=self.epochs, lr=self.l
                model.train(X_selected, Y_selected)
                key = i, j
                self.models[key] = model

    def loss(self, X, Y):
        prediction = self.predict(X)
        losses = np.abs(Y - prediction)
        return np.sum(losses)

    def predict(self, X):
        votes = np.zeros((X.shape[0], self.n_classes))
        confidence_scores = np.zeros((X.shape[0], self.n_classes))

        for (i, j), model in self.models.items():
            probabilities = model.predict_proba(X)
            predictions = (probabilities >= 0.5).astype(int).flatten()


            votes[:, i] += (1 - predictions)
            votes[:, j] += predictions


            confidence_scores[:, i] += 1 - probabilities.flatten()
            confidence_scores[:, j] += probabilities.flatten()
        max_votes = np.max(votes, axis=1, keepdims=True)
        candidates = (votes == max_votes).astype(int)
        final_scores = confidence_scores * candidates
        return np.argmax(final_scores, axis=1)


    def accuracy(self, X, Y):
        predictions = self.predict(X)
        return np.mean(predictions == Y)
```

2.2 Unit Tests

We develop a series of unit tests to evaluate our models and provide helper functions for visualizing the results, which are compared against those of sklearn models. The unit tests are categorized based on their corresponding test models. For each model, we include the following components:

- A helper function for visualization.
- Unit tests on toy datasets to validate the model's internal functions.
- A unit test to compare the results with sklearn models on a linearly separable dataset.
- A unit test to compare the results with sklearn models on a more complex, non-linear dataset.

2.2.1 Check Logistic Regression

2.2.1.1 A helper function for visualization

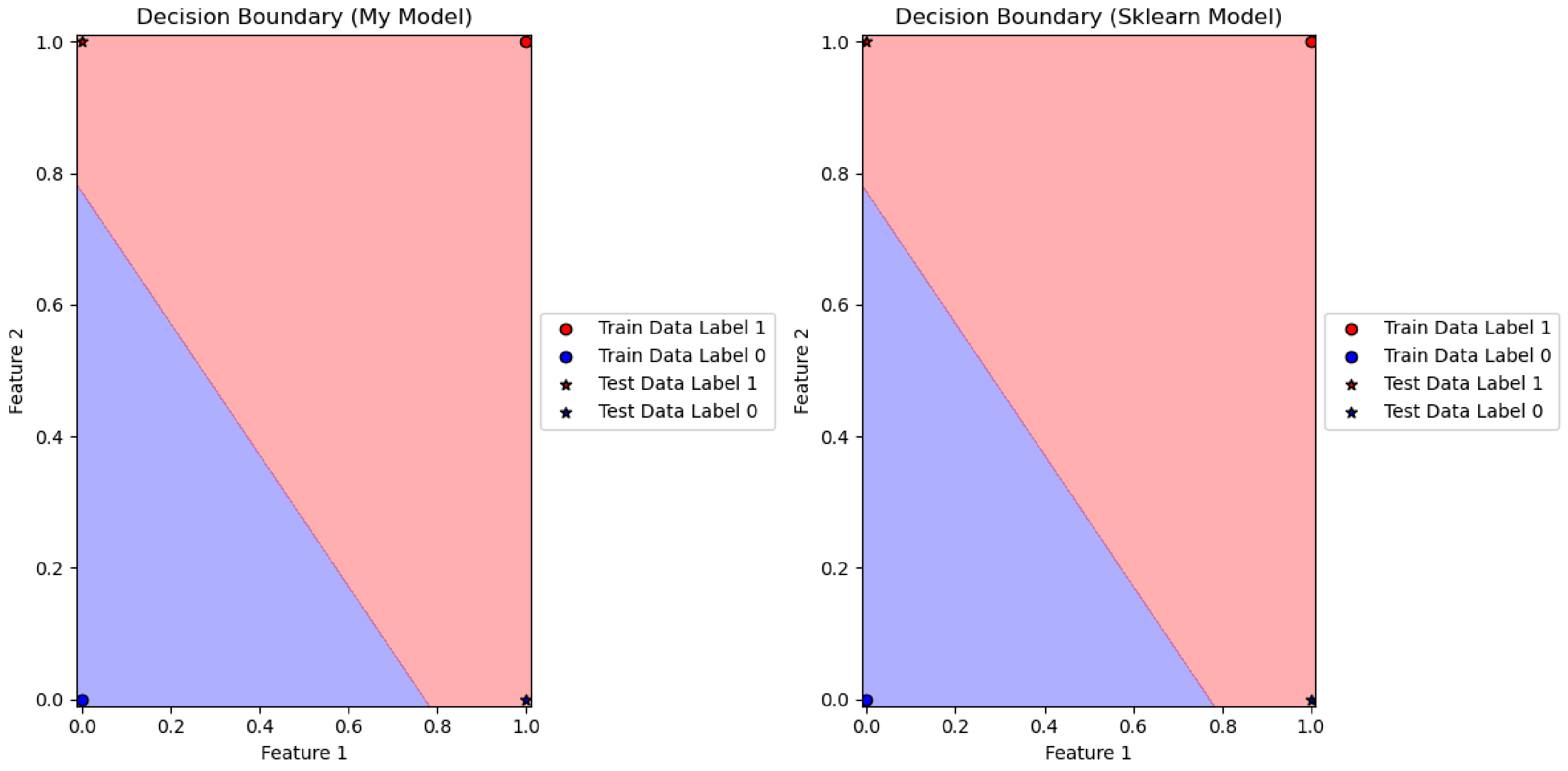
In [5]:
```
import pytest
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDClassifier
# Testing models using SGDClassifier from Scikit-learn (2024)


def CheckPlot_LR(X, Y, X_test, Y_test, lr, num_epochs):
    '''
    helper function for visualizing the differences of our logistic regression model
    and the SGD classifier from sklearn.
    '''
    # train my models
    X_bias = np.c_[X, np.ones(X.shape[0])]
    my_model = MyLogisticRegression(lr=lr, batch_size=1, num_epochs=num_epochs)
    my_model.train(X_bias, Y)

    # train sklearn model
    # SGDClassifier is always batch_size == 1
    sklearn_model = get_estimator(num_epochs, lr)
    sklearn_model.fit(X, Y)

    weights = my_model.weights
    assert isinstance(weights, np.ndarray)
    assert weights.ndim==2 and weights.shape == (1,X.shape[1]+1)
    # FIXME: relative tolerance might be not strict enough
    print('my weight', weights[0])
    print('sklearn weight', np.append(sklearn_model.coef_[0], sklearn_model.intercept_[0]))
    assert weights[0][:-1] == pytest.approx(sklearn_model.coef_[0], 0.01)
    assert weights[0][-1] == pytest.approx(sklearn_model.intercept_[0], 0.01)

    # Create a meshgrid and predict on the grid points
    x_min = min(X[:, 0].min(), X_test[:, 0].min())
    x_max = max(X[:, 0].max(), X_test[:, 0].max())
    y_min = min(X[:, 1].min(), X_test[:, 1].min())
    y_max = max(X[:, 1].max(), X_test[:, 1].max())
    x_gap = x_max - x_min
    y_gap = y_max - y_min

    xx, yy = np.meshgrid(np.linspace(x_min -x_gap/100, x_max + x_gap/100, 200), np.linspace(y_min-y_gap/10
    bias = np.ones(xx.ravel().shape)
    features_with_bias = np.c_[xx.ravel(), yy.ravel(), bias]

    my_Z = my_model.predict(features_with_bias)
    my_Z = my_Z.reshape(xx.shape)

    sklearn_Z = sklearn_model.predict(np.c_[xx.ravel(), yy.ravel()])
    sklearn_Z = sklearn_Z.reshape(xx.shape)

    # check test data predictions
    X_test_bias = np.c_[X_test, np.ones(X_test.shape[0])]
    my_preds = my_model.predict(X_test_bias)
    sklearn_preds = sklearn_model.predict(X_test)
    assert (my_preds == sklearn_preds).all()

    # Create subplots
    fig, axes = plt.subplots(1, 2, figsize=(12, 6))

    # Plot my model's results
    axes[0].contourf(xx, yy, my_Z, alpha=0.3, colors=['blue', 'red'], levels=[0, 0.5, 1])
    train_data_label1 = X[np.where(Y == 1)]
    train_data_label0 = X[np.where(Y == 0)]
    test_data_label1 = X_test[np.where(Y_test == 1)]
    test_data_label0 = X_test[np.where(Y_test == 0)]
    axes[0].scatter(train_data_label1[:, 0], train_data_label1[:, 1], c='red', edgecolor='k', label='Train
    axes[0].scatter(train_data_label0[:, 0], train_data_label0[:, 1], c='blue', edgecolor='k', label='Trai
    axes[0].scatter(test_data_label1[:, 0], test_data_label1[:, 1], c='red', edgecolor='k', label='Test Da
    axes[0].scatter(test_data_label0[:, 0], test_data_label0[:, 1], c='blue', edgecolor='k', label='Test D
    axes[0].set_title("Decision Boundary (My Model)")
    axes[0].set_xlabel("Feature 1")
    axes[0].set_ylabel("Feature 2")
    axes[0].legend(loc='center left', bbox_to_anchor=(1, 0.5))

    # Plot sklearn model's results
    axes[1].contourf(xx, yy, sklearn_Z, alpha=0.3, colors=['blue', 'red'], levels=[0, 0.5, 1])
    axes[1].scatter(train_data_label1[:, 0], train_data_label1[:, 1], c='red', edgecolor='k', label='Train
    axes[1].scatter(train_data_label0[:, 0], train_data_label0[:, 1], c='blue', edgecolor='k', label='Trai
    axes[1].scatter(test_data_label1[:, 0], test_data_label1[:, 1], c='red', edgecolor='k', label='Test Da
    axes[1].scatter(test_data_label0[:, 0], test_data_label0[:, 1], c='blue', edgecolor='k', label='Test D
    axes[1].set_title("Decision Boundary (Sklearn Model)")
    axes[1].set_xlabel("Feature 1")
    axes[1].set_ylabel("Feature 2")
    axes[1].legend(loc='center left', bbox_to_anchor=(1, 0.5))

    # Adjust layout and show the plot
    plt.tight_layout()
    plt.show()

    my_acc = my_model.accuracy(X=X_test_bias, Y=Y_test)
    sklearn_acc = sklearn_model.score(X=X_test,y=Y_test)
    print('The test Accuracy of My Logistic Regression Model is ', my_acc)
    print('The test Accuracy of Sklearn SGD Classifier is ', sklearn_acc)
```

2.2.1.2 2-Point Toy Dataset

To evaluate the performance of different models when the test points are on the decision boundary.

In [6]:
```
# Set random seed for testing purposes
np.random.seed(0)

# Create Data
X1_train = np.array([[0,0], [1,1]])
Y1_train = np.array([0,1])

X1_test = np.array([[0,1], [1,0]])
Y1_test = np.array([1, 0])

CheckPlot_LR(X=X1_train, Y=Y1_train, X_test=X1_test, Y_test=Y1_test, lr=0.01, num_epochs=1000)
```

```
my weight [ 1.85177289  1.85177289 -1.42881498]
sklearn weight [ 1.85177289  1.85177289 -1.42881498]
```



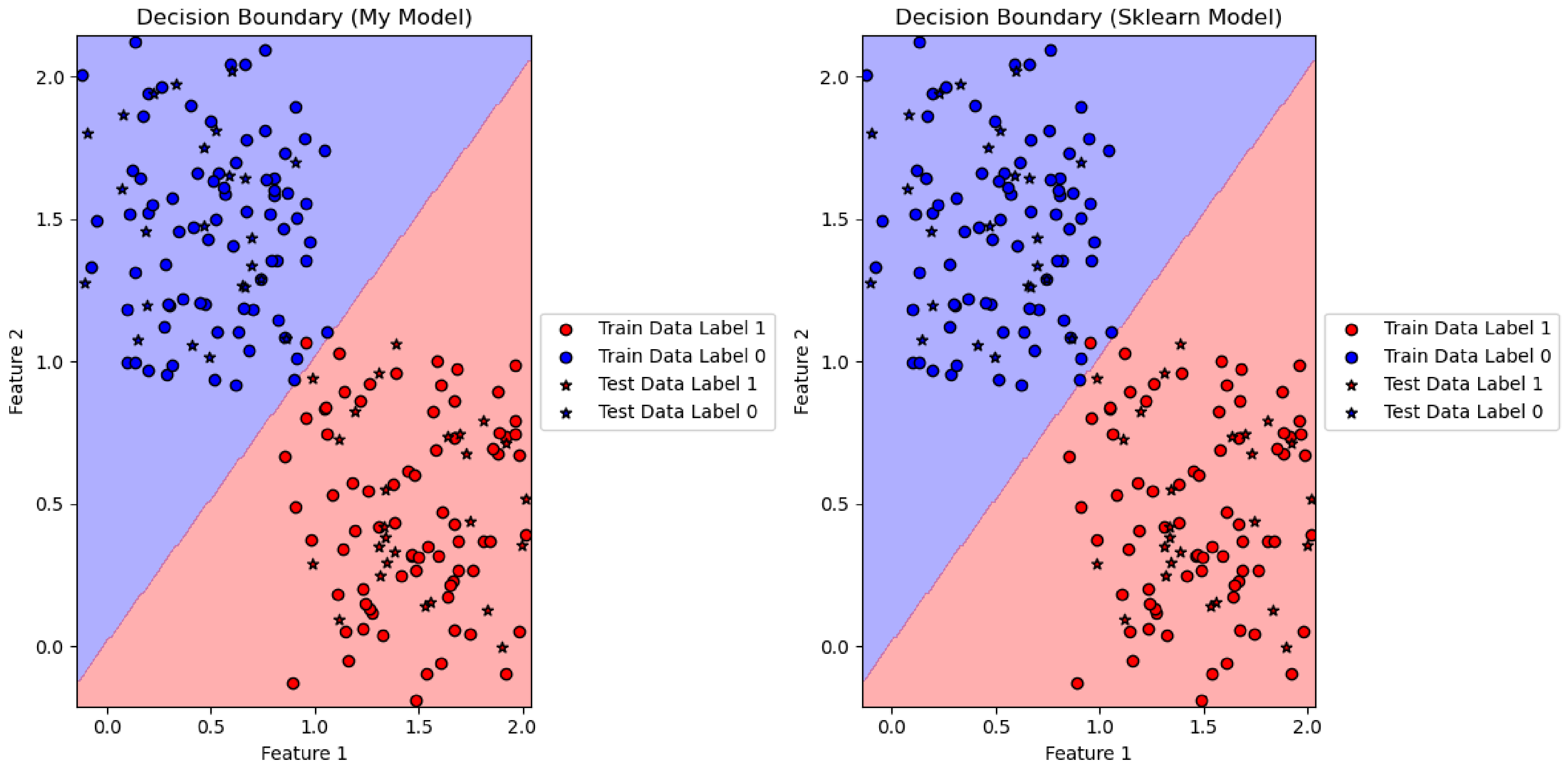
```
The test Accuracy of My Logistic Regression Model is  0.5
The test Accuracy of Sklearn SGD Classifier is  0.5
```

2.2.1.3 Linearly separated dataset

To evaluate performance of different models on linearly separated dataset

In [7]:
```
np.random.seed(0)

# generate two linear seperated datasets
class_0 = np.random.rand(100, 2) + [0, 1]
class_1 = np.random.rand(100, 2) + [1, 0]

# add noises
noise_0 = np.random.normal(0, 0.1, class_0.shape)
noise_1 = np.random.normal(0, 0.1, class_1.shape)
class_0 += noise_0
class_1 += noise_1

X2 = np.vstack((class_0, class_1))
Y2 = np.hstack((np.zeros(100), np.ones(100)))

# Split train and test datasets
indices = np.arange(X2.shape[0])
shuffled_inds = np.random.permutation(indices)
X2 = X2[shuffled_inds]
Y2 = Y2[shuffled_inds]
X2_train = X2[indices[:150]]
Y2_train = Y2[indices[:150]]
X2_test = X2[indices[-51:-1]]
Y2_test = Y2[indices[-51:-1]]

# check
CheckPlot_LR(X=X2_train, Y=Y2_train, X_test=X2_test, Y_test=Y2_test,
             lr=0.01, num_epochs=1000)
```

```
my weight [ 8.62864918 -8.61004358  0.14766954]
sklearn weight [ 8.62864918 -8.61004358  0.14766954]
```



```
The test Accuracy of My Logistic Regression Model is  1.0
The test Accuracy of Sklearn SGD Classifier is  1.0
```

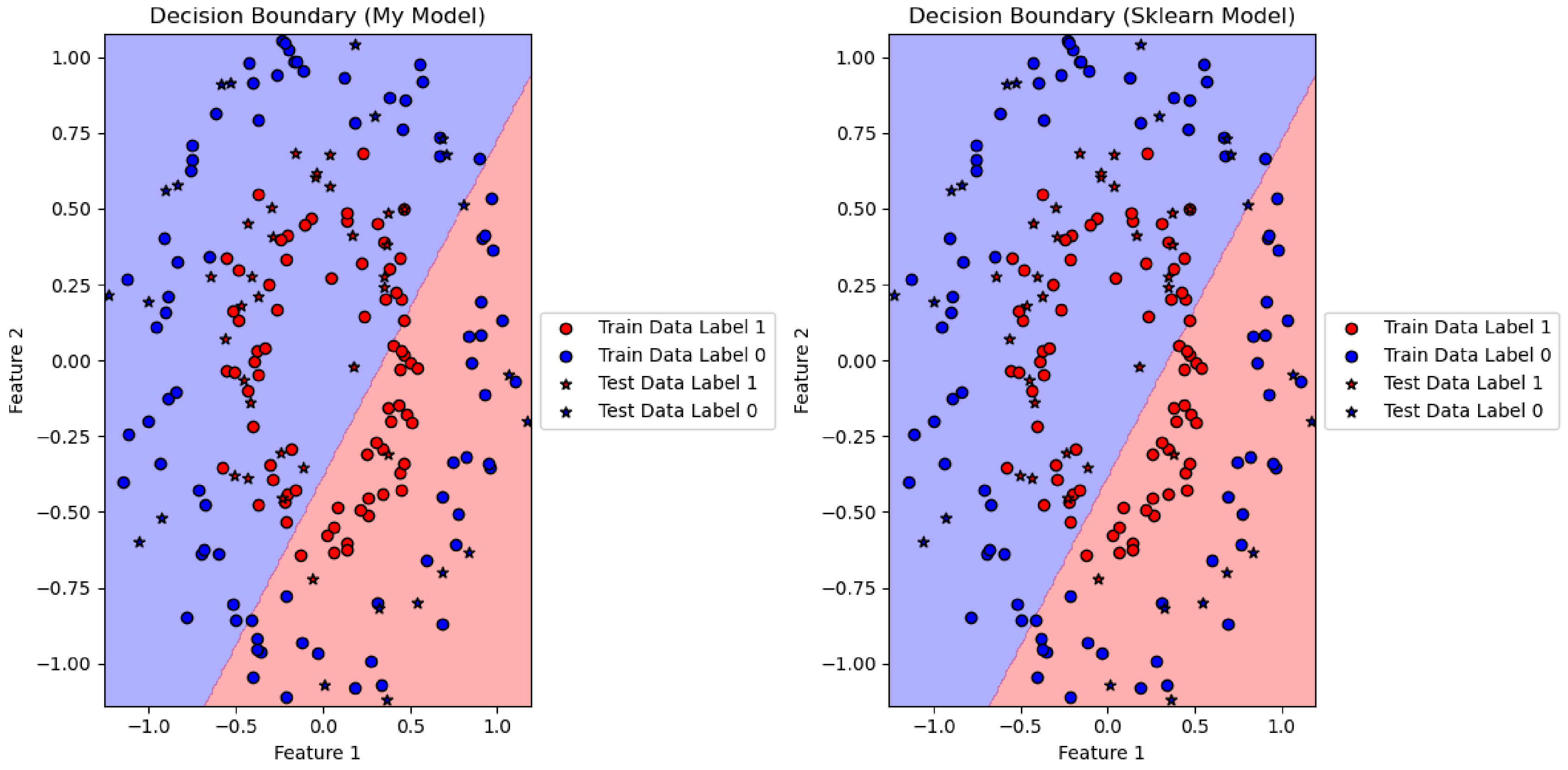
2.2.1.4 Non-Linearly separated datasets

To evaluate the models' performance on a more non-trivial case -- on non-linearly separated dataset

In [8]:
```python
from sklearn.datasets import make_circles
np.random.seed(0)

# Generate circle-like datasets
X3, Y3 = make_circles(n_samples=200, noise=0.1, factor=0.5, random_state=0)
indices = np.arange(X3.shape[0])
shuffled_inds = np.random.permutation(indices)
X3 = X3[shuffled_inds]
Y3 = Y3[shuffled_inds]
```

```
X3_train = X3[indices[:150]]
Y3_train = Y3[indices[:150]]
X3_test = X3[indices[-51:-1]]
Y3_test = Y3[indices[-51:-1]]

# check
CheckPlot_LR(X=X3_train, Y=Y3_train, X_test=X3_test, Y_test=Y3_test,
             lr=0.01, num_epochs=1000)
```

```
my weight [ 0.24860853 -0.22378149 -0.08607428]
sklearn weight [ 0.24860853 -0.22378149 -0.08607428]
```



```
The test Accuracy of My Logistic Regression Model is  0.3
The test Accuracy of Sklearn SGD Classifier is  0.3
```

2.2.2 Check Model: one-vs-all

2.2.2.1 A helper function for visualization

In [9]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.multiclass import OneVsRestClassifier
# Testing models using OneVsRestClassifier from Scikit-learn (2024)

def CheckPlot_OnevsAll(X_train, Y_train, X_test, Y_test, lr, num_epochs):
    '''
    helper function for visualizing the differences of our OnevsAll model
    and OneVsRestClassifier from sklearn.
    '''
    X_train_bias = np.c_[X_train, np.ones(X_train.shape[0])]
    X_test_bias = np.c_[X_test, np.ones(X_test.shape[0])]

    # initialize model
    n_classes = len(np.unique(Y_train))
    my_model = OnevsAll(n_classes, epochs=num_epochs, lr=lr)
    estimator = get_estimator(num_epochs, lr)
    sklearn_model = OneVsRestClassifier(estimator)

    my_model.train(X_train_bias, Y_train)
    sklearn_model.fit(X_train, Y_train)

    # generate meshgrids and predict on it
    x_min = min(X_train[:, 0].min(), X_test[:, 0].min())
    x_max = max(X_train[:, 0].max(), X_test[:, 0].max())
    y_min = min(X_train[:, 1].min(), X_test[:, 1].min())
    y_max = max(X_train[:, 1].max(), X_test[:, 1].max())
    x_gap = x_max - x_min
    y_gap = y_max - y_min

    xx, yy = np.meshgrid(np.linspace(x_min -x_gap/100, x_max + x_gap/100, 200), np.linspace(y_min-y_gap/10
    bias = np.ones(xx.ravel().shape)
    features_with_bias = np.c_[xx.ravel(), yy.ravel(), bias]

    my_Z = np.array(my_model.predict(features_with_bias))
    my_Z = my_Z.reshape(xx.shape)

    sklearn_Z = sklearn_model.predict(np.c_[xx.ravel(), yy.ravel()])
    sklearn_Z = sklearn_Z.reshape(xx.shape)

    # check test data predictions
    my_preds = my_model.predict(X_test_bias)
    sklearn_preds = sklearn_model.predict(X_test)
    assert (my_preds == sklearn_preds).all()

    # Step 6: Visualize the decision boundaries
    fig, axes = plt.subplots(1, 2, figsize=(14, 6))

    # Custom OvA model
    axes[0].contourf(xx, yy, my_Z.reshape(xx.shape), alpha=0.3, cmap="coolwarm")
    axes[0].scatter(X_train[:, 0], X_train[:, 1],
                    c=Y_train, edgecolor="k",  cmap="coolwarm", label='Train Data')
    axes[0].scatter(X_test[:, 0], X_test[:, 1],
                    c=Y_test, edgecolor="k",  cmap="coolwarm", marker='*', label='Test Data')
    axes[0].set_title("My One-vs-All Model")
    axes[0].set_xlabel("Feature 1")
    axes[0].set_ylabel("Feature 2")
    axes[0].legend()

    # Sklearn OvA model
    axes[1].contourf(xx, yy, sklearn_Z.reshape(xx.shape), alpha=0.3, cmap="coolwarm")
    axes[1].scatter(X_train[:, 0], X_train[:, 1],
                    c=Y_train, edgecolor="k",  cmap="coolwarm", label='Train Data')
    axes[1].scatter(X_test[:, 0], X_test[:, 1],
                    c=Y_test, edgecolor="k",  cmap="coolwarm", marker='*', label='Test Data')
    axes[1].set_title("Sklearn One-vs-Rest Model")
    axes[1].set_xlabel("Feature 1")
    axes[1].set_ylabel("Feature 2")
    axes[1].legend()

    plt.tight_layout()
    plt.show()

    my_acc = my_model.accuracy(X=X_test_bias, Y=Y_test)
    sk_acc = sklearn_model.score(X_test, y=Y_test)
    print('The test accuracy of my One-vs-All model is', my_acc)
    print('The test accuracy of sklearn One-vs-All model is', sk_acc)
```

2.2.2.2 10-Point Toy Model

A toy dataset to train on to compare the weights of Binary classifiers for different classes.

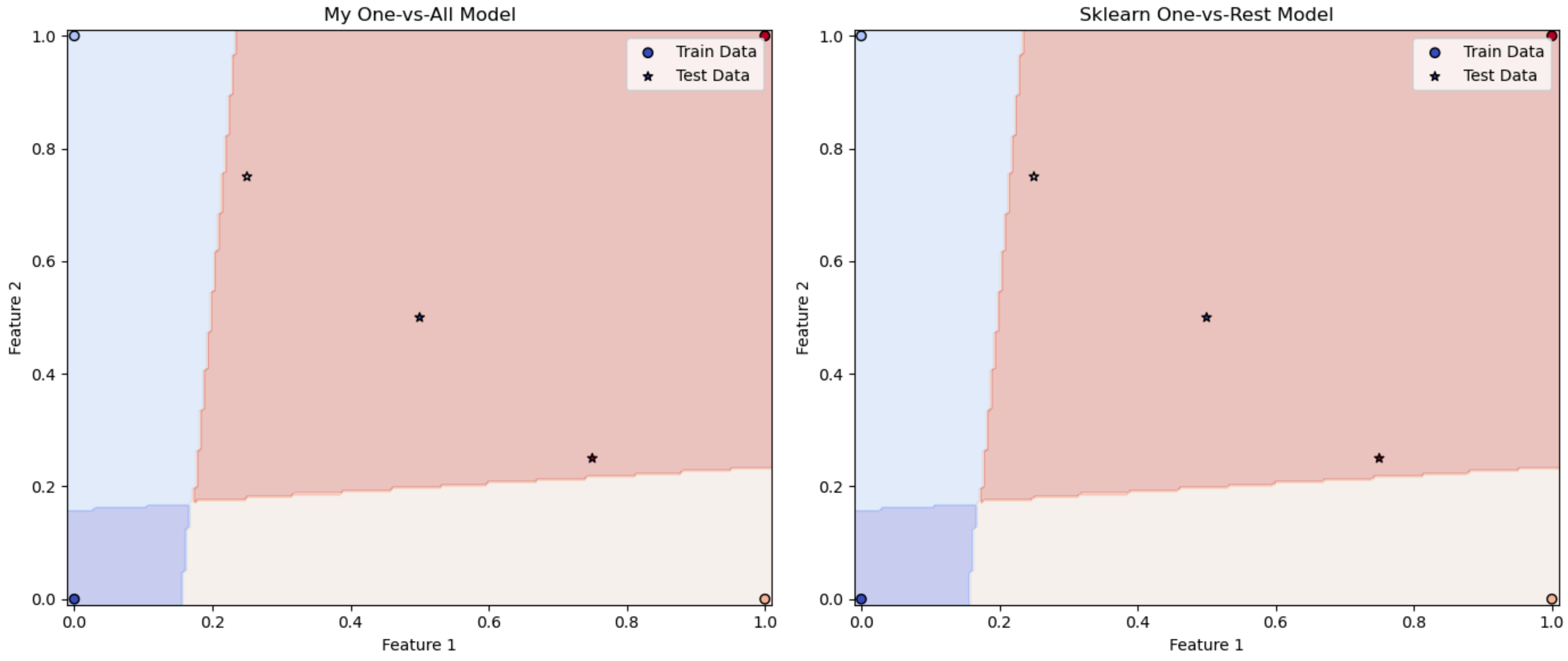
In [10]:
```
import numpy as np
import pytest
from sklearn.multiclass import OneVsRestClassifier
from sklearn.linear_model import SGDClassifier
# Testing models using OneVsRestClassifier from Scikit-learn (2024)
# Testing models using SGDClassifier from Scikit-learn (2024)

np.random.seed(0)

X = np.array([[0,4], [0,3], [5,0], [4,1], [0,5], [1,0], [2,1], [3,2], [4,3], [5,4]])
X_bias = np.array([[0,4,1], [0,3,1], [5,0,1], [4,1,1], [0,5,1], [1,0,1], [2,1,1], [3,2,1], [4,3,1], [5,4,1
Y = np.array([0,0,0,1,1,1,1,2,2,2])
n_classes = len(np.unique(Y))

# Initialize models:
train_epochs = 100
lr = 0.01
my_model = OnevsAll(n_classes, epochs=train_epochs, lr=lr)

estimator = get_estimator(train_epochs, lr)
sklearn_model = OneVsRestClassifier(estimator)

my_model.train(X_bias, Y)
sklearn_model.fit(X, Y)

# Check that S is populated correctly
assert my_model.S_Y.shape[0] == n_classes
assert my_model.S_Y.shape[1] == Y.shape[0]

# Check h (individual classifiers)
assert len(my_model.h) == n_classes
assert len(sklearn_model.estimators_) == n_classes
for h in my_model.h:
    assert isinstance(h, MyLogisticRegression)
for i in range(n_classes):
    my_weights = my_model.h[i].weights[0][:-1]
    my_bias = my_model.h[i].weights[0][-1]
    sklearn_weights = sklearn_model.estimators_[i].coef_[0]
    sklearn_bias = sklearn_model.estimators_[i].intercept_[0]
    print(" === ", i)
    print(my_weights)
    print(sklearn_weights)
    print(my_bias)
    print(sklearn_bias)

    # assert my_weights == pytest.approx(sklearn_weights, 0.01)
    # assert my_bias == pytest.approx(sklearn_bias, 0.01)

# Check predictions
predictions = my_model.predict(X_bias)
print("My Predictions:", np.array(predictions))

sklearn_predictions = sklearn_model.predict(X)
print("sklearn Predictions:", sklearn_predictions)

print('num_samples', X.shape[0])
print('Differences', np.sum(np.abs(np.array(predictions)-sklearn_predictions)))
```

```
 ===  0
[-0.31197662 -0.10912839]
[-0.31197662 -0.10912839]
-0.026783970677386276
-0.02678397067738624
 ===  1
[-0.25499991 -0.25585737]
[-0.25499991 -0.25585737]
0.533835614573224
0.533835614573224
 ===  2
[0.29462046 0.07319256]
[0.29462046 0.07319256]
-1.1883256927760357
-1.1883256927760357
My Predictions: [0 1 2 2 0 1 1 2 2 2]
sklearn Predictions: [0 1 2 2 0 1 1 2 2 2]
num_samples 10
Differences 0
```

2.2.2.3 4-Point Toy Dataset

To evaluate the performance of models when test points are on decision boundary.

In [11]:
```
import numpy as np

np.random.seed(0)

X7_train = np.array([[0,0],[0,1],[1,0],[1,1]])
Y7_train = np.array([0,1,2,3])
X7_test = np.array([[0.5,0.5],[0.75,0.25],[0.25,0.75]])
Y7_test = np.array([0, 2, 1])

#check
CheckPlot_OnevsAll(X7_train, Y7_train, X7_test, Y7_test, lr=0.01, num_epochs=100)
```

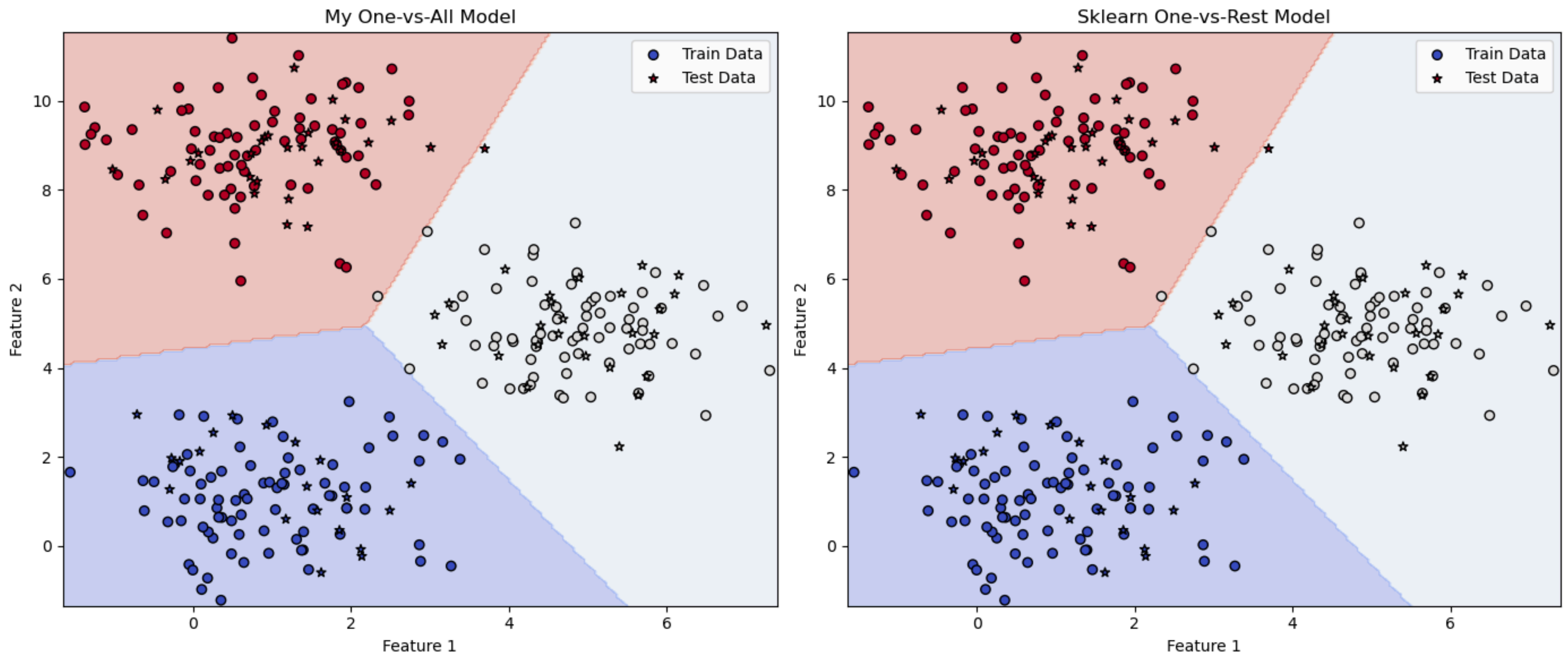


```
The test accuracy of my One-vs-All model is 0.0
The test accuracy of sklearn One-vs-All model is 0.0
```

2.2.2.4 Linearly-separated Datasets

To evaluate performance of different models on linearly separated dataset

In [12]:
```
import numpy as np

np.random.seed(0)

# Generate 100 samples for each class
n_samples = 100

# class 0 centered on [1,1]
class_1 = np.random.randn(n_samples, 2) + [1, 1]

# class 1 centered on [6,6]
class_2 = np.random.randn(n_samples, 2) + [5, 5]

# class 2 centered on [3,4]
class_3 = np.random.randn(n_samples, 2) + [1, 9]


X5 = np.vstack([class_1, class_2, class_3])
Y5 = np.hstack([np.zeros(n_samples), np.ones(n_samples), np.full(n_samples, 2)])

indices = np.arange(X5.shape[0])
shuffled_inds = np.random.permutation(indices)
X5 = X5[shuffled_inds]
Y5 = Y5[shuffled_inds]
X5_train = X5[indices[:225]]
Y5_train = Y5[indices[:225]]
X5_test = X5[indices[-76:-1]]
Y5_test = Y5[indices[-76:-1]]

#check
CheckPlot_OnevsAll(X5_train, Y5_train, X5_test, Y5_test, lr=0.01, num_epochs=1000)
```



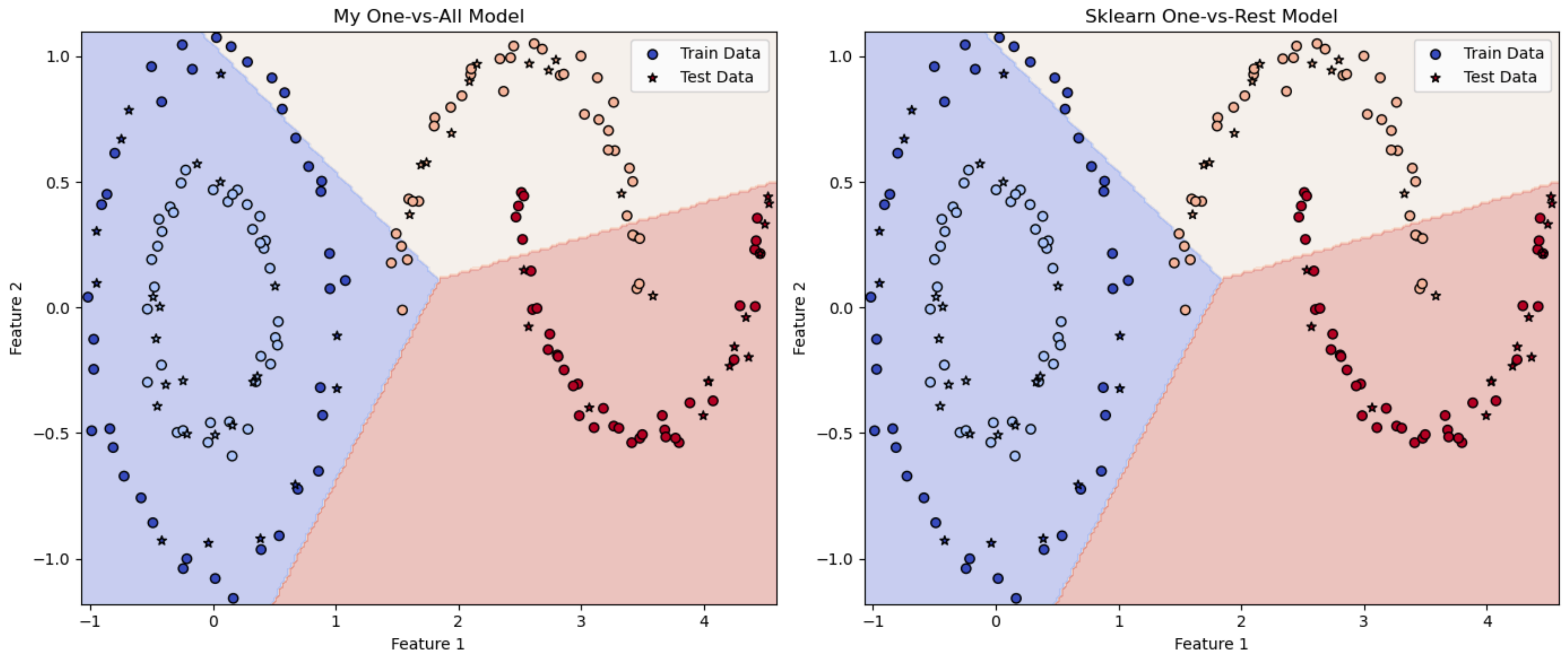
```
The test accuracy of my One-vs-All model is 0.9866666666666667
The test accuracy of sklearn One-vs-All model is 0.9866666666666667
```

2.2.2.5 Non-linearly Separated Data

To evaluate the models' performance on a more non-trivial case -- on non-linearly separated dataset

In [13]:
```python
from sklearn.datasets import make_circles, make_moons
np.random.seed(0)

x1, y1 = make_circles(n_samples=100, noise=0.05, factor=0.5, random_state=0)
x2, y2 = make_moons(n_samples=100, noise=0.05, random_state=0)
x2 += np.array([2.5, 0])
y2 += 2

X6 = np.vstack([x1, x2])
Y6 = np.hstack([y1, y2])

indices = np.arange(X6.shape[0])
```

```
shuffled_inds = np.random.permutation(indices)
X6 = X6[shuffled_inds]
Y6 = Y6[shuffled_inds]
X6_train = X6[indices[:150]]
Y6_train = Y6[indices[:150]]
X6_test = X6[indices[-51:-1]]
Y6_test = Y6[indices[-51:-1]]

#check
CheckPlot_OnevsAll(X6_train, Y6_train, X6_test, Y6_test, lr=0.01, num_epochs=1000)
```



```
The test accuracy of my One-vs-All model is 0.7
The test accuracy of sklearn One-vs-All model is 0.7
```

2.2.3 Check Model: all-pairs

2.2.3.1 A helper function for visualization

In [14]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.multiclass import OneVsOneClassifier
# Testing models using OneVsOneClassifier from Scikit-learn (2024)

def CheckPlot_AllPairs(X_train, Y_train, X_test, Y_test, lr, num_epochs):
    '''
    a helper function for visualizing the differences of our AllPairs model
    and the OneVsOneClassifier from sklearn.
    '''
    X_train_bias = np.c_[X_train, np.ones(X_train.shape[0])]
    X_test_bias = np.c_[X_test, np.ones(X_test.shape[0])]

    # initialize model
    n_classes = len(np.unique(Y_train))
    my_model = AllPairs(n_classes=n_classes, batch_size=1, epochs=num_epochs, lr=lr)

    estimator = get_estimator(num_epochs, lr)
    sklearn_model = OneVsOneClassifier(estimator)

    my_model.train(X_train_bias, Y_train)
    sklearn_model.fit(X_train, Y_train)

    # generate meshgrids and predict on it
    x_min = min(X_train[:, 0].min(), X_test[:, 0].min())
    x_max = max(X_train[:, 0].max(), X_test[:, 0].max())
    y_min = min(X_train[:, 1].min(), X_test[:, 1].min())
    y_max = max(X_train[:, 1].max(), X_test[:, 1].max())
    x_gap = x_max - x_min
    y_gap = y_max - y_min

    xx, yy = np.meshgrid(np.linspace(x_min -x_gap/100, x_max + x_gap/100, 200), np.linspace(y_min-y_gap/10
    bias = np.ones(xx.ravel().shape)
    features_with_bias = np.c_[xx.ravel(), yy.ravel(), bias]

    my_Z = np.array(my_model.predict(features_with_bias))
    my_Z = my_Z.reshape(xx.shape)

    sklearn_Z = sklearn_model.predict(np.c_[xx.ravel(), yy.ravel()])
    sklearn_Z = sklearn_Z.reshape(xx.shape)
```

```
# check test data predictions
my_preds = my_model.predict(X_test_bias)
sklearn_preds = sklearn_model.predict(X_test)
assert (my_preds == sklearn_preds).all()

# Step 6: Visualize the decision boundaries
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# Custom OvA model
axes[0].contourf(xx, yy, my_Z.reshape(xx.shape), alpha=0.3, cmap="coolwarm")
axes[0].scatter(X_train[:, 0], X_train[:, 1],
                c=Y_train, edgecolor="k",  cmap="coolwarm", label='Train Data')
axes[0].scatter(X_test[:, 0], X_test[:, 1],
                c=Y_test, edgecolor="k",  cmap="coolwarm", marker='*', label='Test Data')
axes[0].set_title("My AllPair Model")
axes[0].set_xlabel("Feature 1")
axes[0].set_ylabel("Feature 2")
axes[0].legend()

# Sklearn OvA model
axes[1].contourf(xx, yy, sklearn_Z.reshape(xx.shape), alpha=0.3, cmap="coolwarm")
axes[1].scatter(X_train[:, 0], X_train[:, 1],
                c=Y_train, edgecolor="k",  cmap="coolwarm", label='Train Data')
axes[1].scatter(X_test[:, 0], X_test[:, 1],
                c=Y_test, edgecolor="k",  cmap="coolwarm", marker='*', label='Test Data')
axes[1].set_title("Sklearn Allpair Model")
axes[1].set_xlabel("Feature 1")
axes[1].set_ylabel("Feature 2")
axes[1].legend()

plt.tight_layout()
plt.show()

my_acc = my_model.accuracy(X=X_test_bias, Y=Y_test)
sk_acc = sklearn_model.score(X_test, y=Y_test)
print('The test accuracy of my One-vs-All model is', my_acc)
print('The test accuracy of sklearn One-vs-All model is', sk_acc)
```

2.2.3.2 10-Point Toy Model

A toy dataset to train on to compare the weights of Binary Classifiers for different pairs.

In [15]:
```
#Does not work when shuffle is enabled
import numpy as np
import pytest
from sklearn.multiclass import OneVsOneClassifier
from sklearn.linear_model import SGDClassifier
# Testing models using OneVsOneClassifier from Scikit-learn (2024)
# Testing models using SGDClassifier from Scikit-learn (2024)

X = np.array([[0,4], [0,3], [5,0], [4,1], [0,5], [1,0], [2,1], [3,2], [4,3], [5,4]])
X_bias = np.array([[0,4,1], [0,3,1], [5,0,1], [4,1,1], [0,5,1], [1,0,1], [2,1,1], [3,2,1], [4,3,1], [5,4,1
Y = np.array([0,0,0,1,1,1,1,2,2,2])
n_classes = len(np.unique(Y))


train_epochs = 100
lr = 0.01
my_model = AllPairs(n_classes, batch_size=1, epochs=train_epochs, lr=lr)
my_model.train(X_bias, Y)

estimator = get_estimator(train_epochs, lr)
sklearn_model = OneVsOneClassifier(estimator)
sklearn_model.fit(X, Y)


def test_train_basic():
    """
    Test if the model trains the correct number of binary classifiers
    """
    my_model = AllPairs(n_classes, batch_size=1, epochs=train_epochs, lr=lr)
    my_model.train(X_bias, Y)
    assert len(my_model.models) == (n_classes * (n_classes - 1)) // 2, "Incorrect number of models trained


def test_train_weights():
    """
    Compare weights and biases of trained models against Scikit-learn's OneVsOneClassifier.
    """
    my_model = AllPairs(n_classes, batch_size=1, epochs=train_epochs, lr=lr)
    my_model.train(X_bias, Y)
    for i in range(n_classes):
        for j in range(i + 1, n_classes):
            key = (i, j)
            model = my_model.models[key]
            my_weights = model.weights[0][:-1]
            my_bias = model.weights[0][-1]

            model_index = int(i * n_classes - i * (i + 1) / 2 + j - i - 1)
            sklearn_weights = sklearn_model.estimators_[model_index].coef_[0]
            sklearn_bias = sklearn_model.estimators_[model_index].intercept_[0]

            np.testing.assert_allclose(my_weights, sklearn_weights, atol=1e-1, err_msg=f"Weights mismatch 
            np.testing.assert_allclose(my_bias, sklearn_bias, atol=1e-1, err_msg=f"Bias mismatch for class


def test_predict_basic():
    """
    Compare final prediction result with Scikit-learn's OneVsOneClassifier.
    """
    my_model = AllPairs(n_classes, batch_size=1, epochs=train_epochs, lr=lr)
    my_model.train(X_bias, Y)
    predictions = my_model.predict(X_bias)
    sklearn_predictions = sklearn_model.predict(X)

    assert len(predictions) == len(sklearn_predictions), "Prediction length mismatch."
    np.testing.assert_array_equal(predictions, sklearn_predictions, "Predictions do not match Scikit-learn


def test_edge_cases():
    """
    Testing for
    - Empty dataset
    - Single class dataset
    - Dataset with unbalanced classes
    """
    my_model = AllPairs(n_classes, batch_size=1, epochs=train_epochs, lr=lr)


    X_empty = np.empty((0, X_bias.shape[1]))
    Y_empty = np.empty((0,))
    with pytest.raises(ValueError, match="No data provided"):
        my_model.train(X_empty, Y_empty)


    X_single_class = X_bias[:3]
    Y_single_class = np.zeros((3,))
    my_model.train(X_single_class, Y_single_class)
    predictions = my_model.predict(X_single_class)
    assert all(predictions == 0), "All predictions should match the single class."


    X_unbalanced = np.vstack([X_bias[:8], X_bias[:2]])
    Y_unbalanced = np.hstack([Y[:8], Y[:2]])
    my_model.train(X_unbalanced, Y_unbalanced)
    predictions = my_model.predict(X_unbalanced)
    assert len(predictions) == len(Y_unbalanced), "Prediction length mismatch for unbalanced dataset."


def test_accuracy():
    """
    Compare accuracy with Scikit-learn's OneVsOneClassifier.
    """
    my_model = AllPairs(n_classes, batch_size=1, epochs=train_epochs, lr=lr)
    my_model.train(X_bias, Y)
    my_accuracy = my_model.accuracy(X_bias, Y)
    sklearn_accuracy = sklearn_model.score(X, Y)

    assert np.isclose(my_accuracy, sklearn_accuracy, atol=1e-2), "Accuracy mismatch between models."


print("Running test_train_basic...")
test_train_basic()
print("test_train_basic passed.")

print("Running test_train_weights...")
test_train_weights()
print("test_train_weights passed.")

print("Running test_predict_basic...")
test_predict_basic()
print("test_predict_basic passed.")

print("Running test_edge_cases...")
```

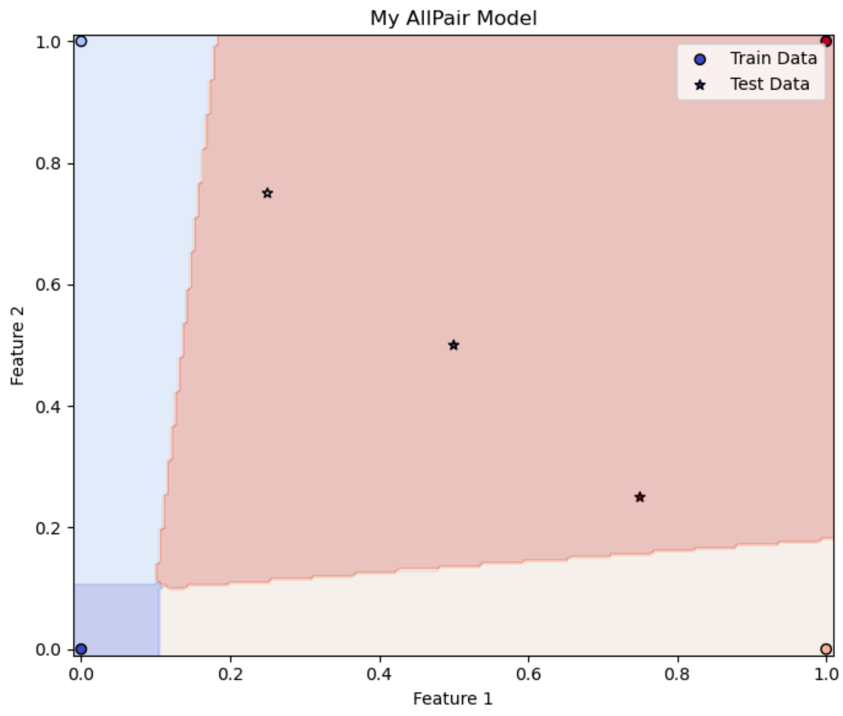
```
test_edge_cases()
print("test_edge_cases passed.")

print("Running test_accuracy...")
test_accuracy()
print("test_accuracy passed.")

print("All tests passed successfully!")
```

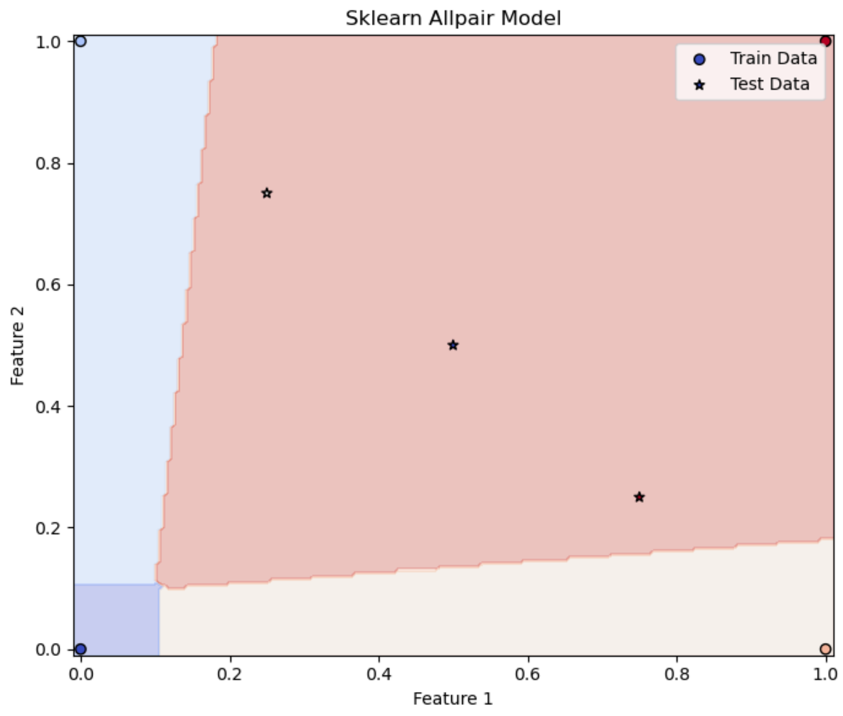
```
Running test_train_basic...
test_train_basic passed.
Running test_train_weights...
test_train_weights passed.
Running test_predict_basic...
test_predict_basic passed.
Running test_edge_cases...
test_edge_cases passed.
Running test_accuracy...
test_accuracy passed.
All tests passed successfully!
```

2.2.3.3 4-Point Toy Dataset

To evaluate the performance of models when test points are on decision boundary.

In [16]:
```
np.random.seed(0)
#check
CheckPlot_AllPairs(X7_train, Y7_train, X7_test, Y7_test, 0.01, 100)
```



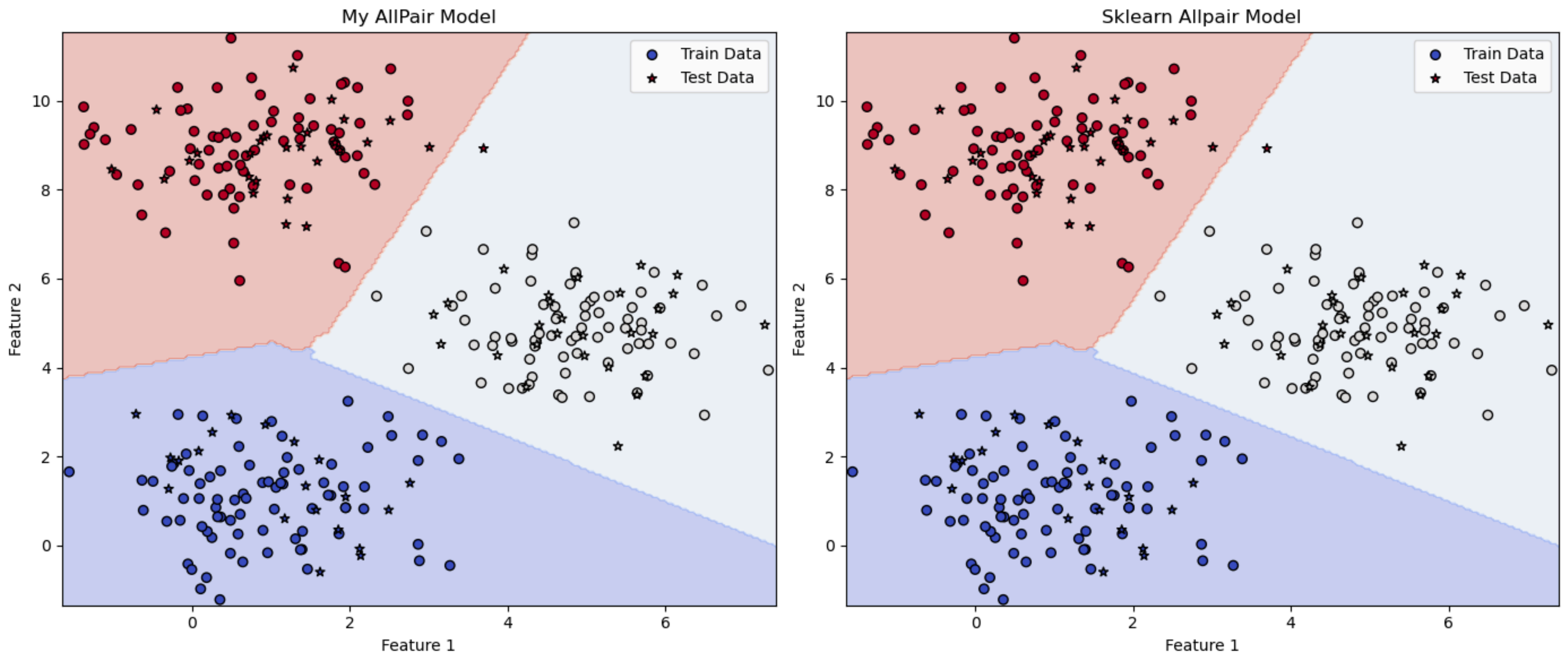


```
The test accuracy of my One-vs-All model is 0.0
The test accuracy of sklearn One-vs-All model is 0.0
```

2.2.3.4 Linearly Separated Datasets

To evaluate performance of different models on linearly separated dataset

In [17]:
```
np.random.seed(0)
CheckPlot_AllPairs(X5_train, Y5_train, X5_test, Y5_test, 0.01, 1000)
```

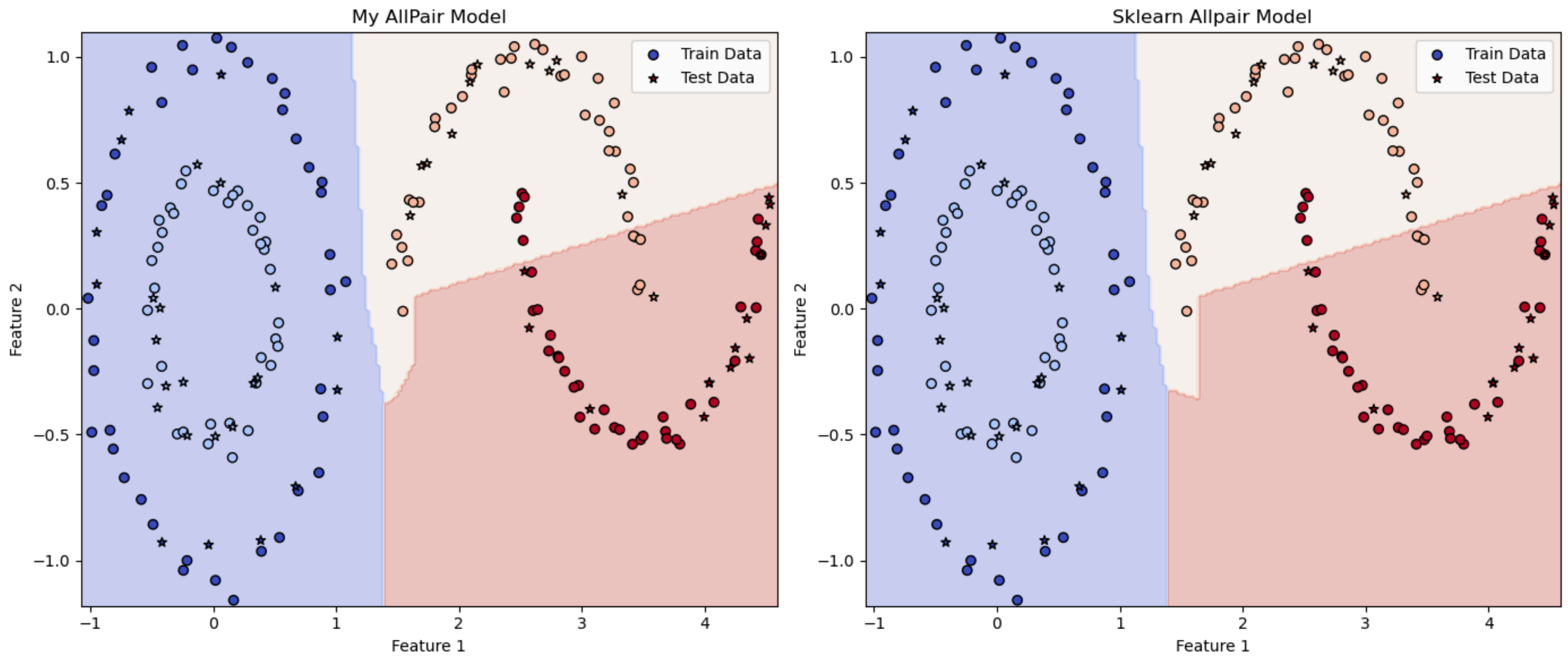


```
The test accuracy of my One-vs-All model is 0.9866666666666667
The test accuracy of sklearn One-vs-All model is 0.9866666666666667
```

2.2.3.5 Non-linearly Separated Datasets

To evaluate the models' performance on a more non-trivial case -- on non-linearly separated dataset

In [18]:
```
np.random.seed(0)
CheckPlot_AllPairs(X6_train, Y6_train, X6_test, Y6_test, 0.01, 1000)
```



```
The test accuracy of my One-vs-All model is 0.7
The test accuracy of sklearn One-vs-All model is 0.7
```

2.3 Performance on Iris Dataset

In [19]:
```python
from sklearn.datasets import load_iris
#Iris dataset for testing, Fisher (1936), accessed via Scikit-learn's load_iris
"""
This block is to split training and testing datasets from Iris dataset
"""

data = load_iris()

X_iris = data.data
Y_iris = data.target
n_classes = len(np.unique(Y_iris))

# split datasets
indices = np.arange(X_iris.shape[0])
```

```
shuffled_inds = np.random.permutation(indices)
X_iris = X_iris[shuffled_inds]
Y_iris = Y_iris[shuffled_inds]

X_iris_train = X_iris[indices[:125]]
X_iris_train_biased = np.hstack((X_iris_train, np.ones((X_iris_train.shape[0], 1))))
Y_iris_train = Y_iris[indices[:125]]

X_iris_test = X_iris[indices[-26:-1]]
X_iris_test_biased = np.hstack((X_iris_test, np.ones((X_iris_test.shape[0], 1))))
Y_iris_test = Y_iris[indices[-26:-1]]
```

2.3.1 One-vs-all Model

To compare the results of our custom One-vs-All model and results of One-vs-Rest Classifier from Sklearn

In [20]:
```
import numpy as np
from sklearn.multiclass import OneVsRestClassifier
# Testing models using OneVsRestClassifier from Scikit-learn (2024)

np.random.seed(0)

# Initialize models:
np.random.seed(0)
train_epochs = 1000
lr = 0.01
batch_size = 1

my_model = OnevsAll(n_classes, epochs=train_epochs, lr=lr)
my_model.train(X_iris_train_biased, Y_iris_train)

estimator = get_estimator(train_epochs, lr)
sklearn_model = OneVsRestClassifier(estimator)
sklearn_model.fit(X_iris_train, Y_iris_train)

my_preds = my_model.predict(X_iris_test_biased)
sklearn_preds = sklearn_model.predict(X_iris_test)
my_acc = my_model.accuracy(X_iris_test_biased, Y_iris_test)
sk_acc = sklearn_model.score(X_iris_test, Y_iris_test)
```

```python
print("Predictions of our One-vs-All model:", np.array(my_preds))
print("Predictions of sklearn OneVsRest Classifier:", sklearn_preds)

print('num_samples of training dataset', X_iris_train.shape[0])
print('num_samples of testing dataset', X_iris_test.shape[0])
print('Differences', np.sum(np.abs(np.array(my_preds)-sklearn_preds)))

print("Accuracy of our One-vs-All model:", my_acc)
print("Accuracy of sklearn OneVsRest Classifier:", sk_acc)
```

```
Predictions of our One-vs-All model: [2 0 2 1 1 1 2 2 2 2 0 1 2 2 0 1 1 2 1 0 0 0 2 1 2]
Predictions of sklearn OneVsRest Classifier: [2 0 2 1 1 1 2 2 2 2 0 1 2 2 0 1 1 2 1 0 0 0 2 1 2]
num_samples of training dataset 125
num_samples of testing dataset 25
Differences 0
Accuracy of our One-vs-All model: 0.88
Accuracy of sklearn OneVsRest Classifier: 0.88
```

2.3.2 All-Pairs Model

To compare results of our All-Pairs model and results of One-vs-One Classifier from sklearn.

In [21]:
```python
import numpy as np
from sklearn.multiclass import OneVsOneClassifier
# Testing models using OneVsOneClassifier from Scikit-learn (2024)

np.random.seed(0)
lr = 0.01
train_epochs = 1000
batch_size = 1

model = AllPairs(n_classes=n_classes, batch_size=1, epochs=train_epochs, lr=lr)
model.train(X_iris_train_biased, Y_iris_train)
predictions = model.predict(X_iris_test_biased)
my_acc = model.accuracy(X_iris_test_biased, Y_iris_test)

estimator = get_estimator(train_epochs, lr)
sklearn_model_ap = OneVsOneClassifier(estimator)
sklearn_model_ap.fit(X_iris_train, Y_iris_train)
sklearn_predictions = sklearn_model_ap.predict(X_iris_test)
sk_acc = sklearn_model_ap.score(X_iris_test, Y_iris_test)
```

```python
print("Predictions of our All-Pairs Model:", np.array(predictions))
print("Predictions of sklearn OneVsOne Classifier:", sklearn_predictions)
print('num_samples of training dataset', X_iris_train.shape[0])
print('num_samples of testing dataset', X_iris_test.shape[0])
print('Differences', np.sum(np.abs(np.array(predictions)-sklearn_predictions)))
print("Accuracy of our All-Pairs model:", my_acc)
print("Accuracy of sklearn OneVsOne Classifier:", sk_acc)
```

```
Predictions of our All-Pairs Model: [2 0 2 1 1 1 2 2 2 1 0 1 2 2 0 1 1 2 1 0 0 0 2 1 2]
Predictions of sklearn OneVsOne Classifier: [2 0 2 1 1 1 2 2 2 1 0 1 2 2 0 1 1 2 1 0 0 0 2 1 2]
num_samples of training dataset 125
num_samples of testing dataset 25
Differences 0
Accuracy of our All-Pairs model: 0.92
Accuracy of sklearn OneVsOne Classifier: 0.92
```

References

Fisher, R.A. (1936) 'Iris.' Available at: https://archive.ics.uci.edu/dataset/53/iris (Accessed November 2024).

Scikit-learn. (2024) *1.12 Multiclass and multioutput algorithms* [Online]. Available at: https://scikit-learn.org/1.5/modules/multiclass.html#multiclass-classification (Accessed November 2024).

Scikit-learn. (2024) *sklearn.metrics.log_loss* [Online]. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.log_loss.html (Accessed November 2024).

Scikit-learn. (2024) *sklearn.multiclass.OneVsRestClassifier* [Online]. Available at: https://scikit-learn.org/1.5/modules/generated/sklearn.multiclass.OneVsRestClassifier.html (Accessed November 2024).

Scikit-learn. (2024) *sklearn.multiclass.OneVsOneClassifier* [Online]. Available at: https://scikit-learn.org/1.5/modules/generated/sklearn.multiclass.OneVsOneClassifier.html (Accessed November 2024).

Scikit-learn. (2024) *sklearn.linear_model.SGDClassifier* [Online]. Available at: https://scikit-learn.org/1.5/modules/generated/sklearn.linear_model.SGDClassifier.html (Accessed November 2024).

Shalev-Shwartz, S. and Ben-David, S. (2014) *Understanding Machine Learning: From Theory to Algorithms*. Cambridge: Cambridge University Press.