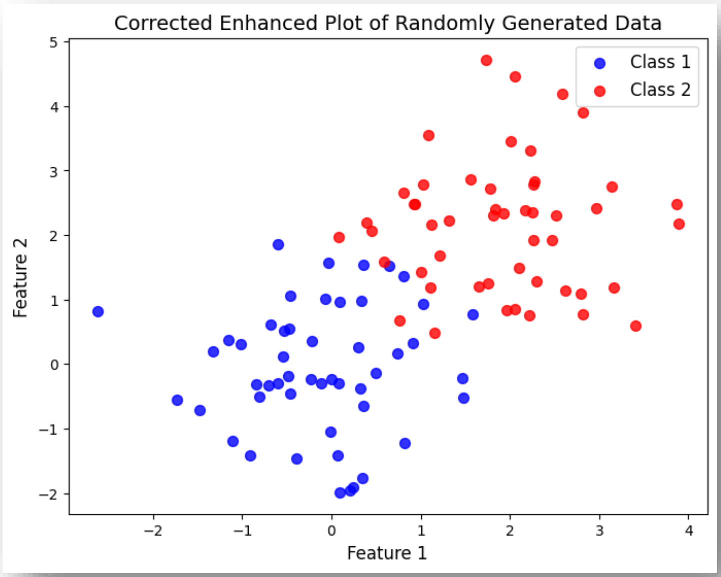
Method 1: Using sklearn and numpy packages

Section 1: Data Generation and Plotting

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score

np.random.seed(42)   # for reproducibility

# Parameters
num_samples = 50
mean_class1 = 0
mean_class2 = 2
covariance_matrix = np.eye(2)

# Generate random vectors
class1_samples = np.random.multivariate_normal([mean_class1, mean_class1],
covariance_matrix, num_samples)
class2_samples = np.random.multivariate_normal([mean_class2, mean_class2],
covariance_matrix, num_samples)

plt.style.use('default')

# Recreate the plot with the correct style
plt.figure(figsize=(8, 6))
plt.scatter(class1_samples[:, 0], class1_samples[:, 1], alpha=0.8, la-
bel='Class 1', color='blue', s=50)
plt.scatter(class2_samples[:, 0], class2_samples[:, 1], alpha=0.8, la-
bel='Class 2', color='red', s=50)
plt.title('Corrected Enhanced Plot of Randomly Generated Data', fontsize=14)
plt.xlabel('Feature 1', fontsize=12)
plt.ylabel('Feature 2', fontsize=12)
plt.legend(fontsize=12)
plt.show()
```

Explanation:

- This section generates synthetic data for two classes with different means and the same covariance matrix.
- The data is then plotted using Matplotlib, where each class is represented by a scatter plot with different colors.



Section 2: Data Preprocessing

```
# Concatenate and shuffle data
all_samples = np.concatenate((class1_samples, class2_samples), axis=0)
labels = np.concatenate((np.zeros(num_samples), np.ones(num_samples)))

# Shuffle data
shuffled_indices = np.random.permutation(2 * num_samples)
all_samples = all_samples[shuffled_indices]
labels = labels[shuffled_indices]
```

Explanation:

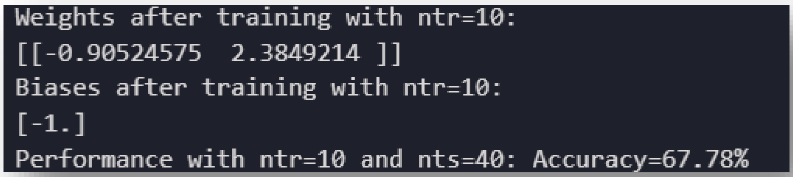
- The data from the two classes is concatenated, and corresponding labels are created.
- The data is shuffled to ensure a random order, which is important for later training and evaluation.

Section 3: Training and Evaluation

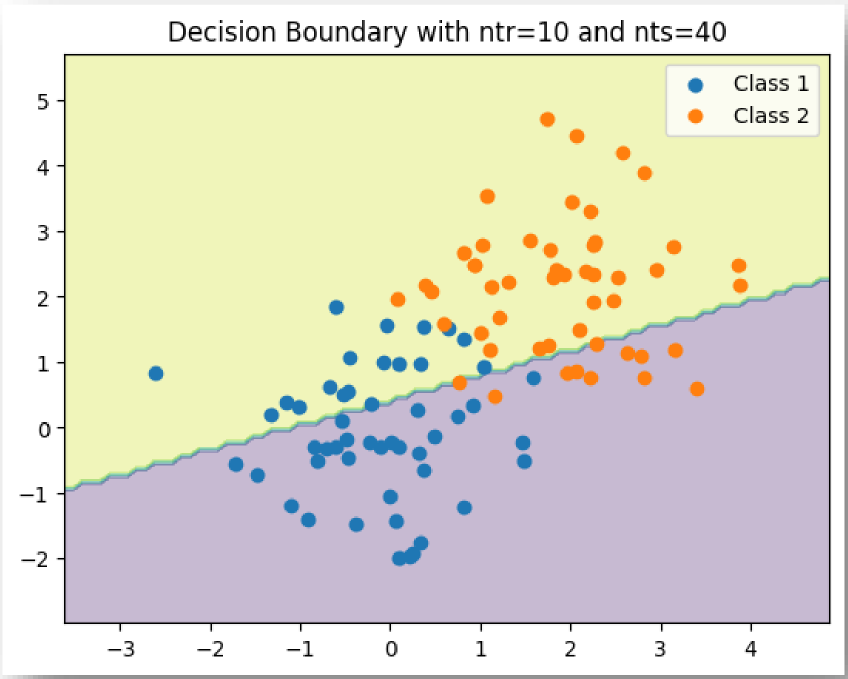
```
# Split into training and testing sets
ntr_values = [10, 20, 30]
nts_values = list(np.array(num_samples) - ntr_values)

# Function to train and evaluate the perceptron
def train_and_evaluate(ntr_value, nts_value):
    best_accuracy = 0
    best_model = None

    # Create Perceptron model
    model = Perceptron(random_state=42)

    for _ in range(5):
        # Train the perceptron with learning data
        model.fit(all_samples[:ntr_value], labels[:ntr_value])

        # Evaluate on test data
        predictions = model.predict(all_samples[ntr_value:])
        accuracy = accuracy_score(labels[ntr_value:], predictions)

        # Keep track of the best model
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_model = model

    print(f"Weights after training with
ntr={ntr_value}:\n{best_model.coef_}")
    print(f"Biases after training with
ntr={ntr_value}:\n{best_model.intercept_}")
    print(f"Performance with ntr={ntr_value} and nts={nts_value}:
Accuracy={best_accuracy:.2%}")

    # Plot decision boundary
    x_min, x_max = all_samples[:, 0].min() - 1, all_samples[:, 0].max() + 1
    y_min, y_max = all_samples[:, 1].min() - 1, all_samples[:, 1].max() + 1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min,
y_max, 0.1))
    Z = best_model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.3)
    plt.scatter(class1_samples[:, 0], class1_samples[:, 1], label='Class 1')
    plt.scatter(class2_samples[:, 0], class2_samples[:, 1], label='Class 2')
    plt.title(f'Decision Boundary with ntr={ntr_value} and nts={nts_value}')
    plt.legend()
    plt.show()

# Train and evaluate for each ntr value
for ntr, nts in zip(ntr_values, nts_values):
    train_and_evaluate(ntr, nts)
```

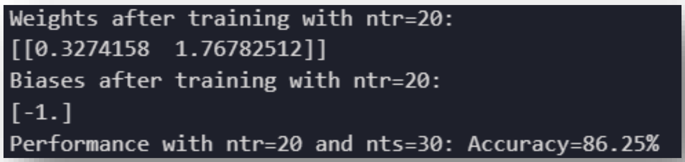
Explanation:

- The code defines a function **train_and_evaluate** to train a Perceptron model and evaluate its performance.
- The function is called for different training set sizes (**ntr_values**) and corresponding testing set sizes (**nts_values**).
- The best-performing model is printed along with its weights, biases, and accuracy.
- Decision boundaries are plotted to visualize the model's classification.

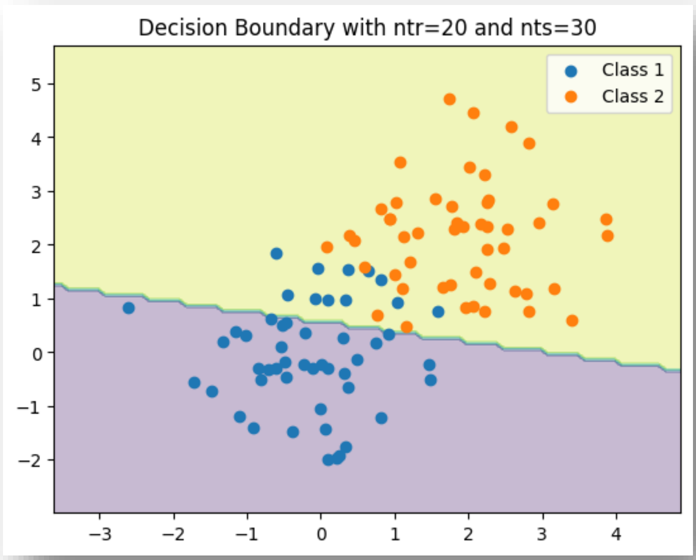


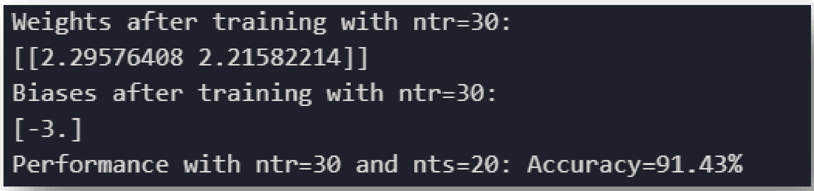
```
Weights after training with ntr=10:
[[-0.90524575  2.3849214 ]]
Biases after training with ntr=10:
[-1.]
Performance with ntr=10 and nts=40: Accuracy=67.78%
```



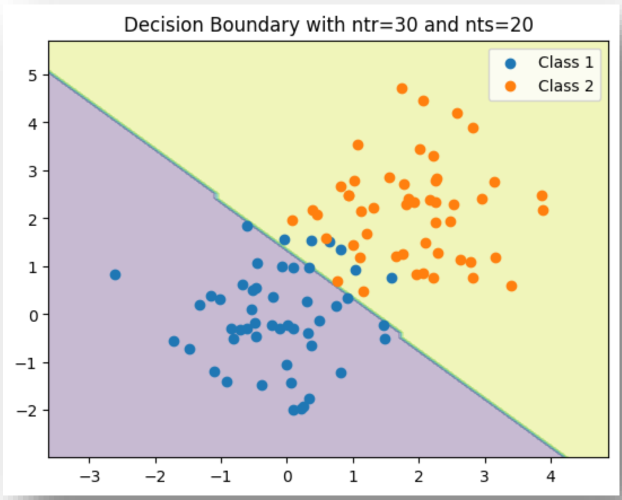


```
Weights after training with ntr=20:
[[0.3274158  1.76782512]]
Biases after training with ntr=20:
[-1.]
Performance with ntr=20 and nts=30: Accuracy=86.25%
```





```
Weights after training with ntr=30:
[[2.29576408 2.21582214]]
Biases after training with ntr=30:
[-3.]
Performance with ntr=30 and nts=20: Accuracy=91.43%
```



Report Summary:

1- Data Generation and Plotting:
- Random data for two classes generated with different means.
- A corrected plot is created using Matplotlib.

2- Data Preprocessing:
- Data from both classes concatenated.
- Shuffling done for randomization.

3- Training and Evaluation:
- Perceptron model trained and evaluated.
- Performance metrics (accuracy, weights, biases) printed.
- Decision boundaries plotted for visualization.

Method 2: Using only numpy packages

Section 1: Data Generation

```
import numpy as np
import matplotlib.pyplot as plt

# Set the parameters for the distributions
mean_1, variance_1 = 0, 1
mean_2, variance_2 = 2, 1

# Generate random numbers from normal distributions
random_number_1 = np.random.normal(mean_1, np.sqrt(variance_1), 50)
random_number_2 = np.random.normal(mean_2, np.sqrt(variance_2), 50)

# Create labels for the two classes
labels_1 = np.zeros(50, dtype=int)
labels_2 = np.ones(50, dtype=int)

# Combine data and labels for each class
combined_data_1 = np.column_stack((random_number_1, labels_1))
combined_data_2 = np.column_stack((random_number_2, labels_2))

# Concatenate the two classes to create a single dataset
combined_dataset = np.concatenate((combined_data_1, combined_data_2))
```

Explanation:

This section uses NumPy to generate random numbers from normal distributions with specified means and variances for two classes. The generated data is then labeled, creating two sets of 50 samples each. The final dataset, **'combined_dataset'**, is a concatenation of these two classes.

Section 2: Data Splitting Function

```
def split_data_for_training_and_testing(dataset, num_per_class_for_training):
    """
    Split the dataset into training and testing sets based on the specified
number per class for training.

    :param dataset: Combined dataset with labels.
    :param num_per_class_for_training: Number of samples per class for the
training set.
    :return: A tuple of (training_data, testing_data).
    """
    # Shuffle the dataset to ensure randomness
    np.random.shuffle(dataset)

    # Extract the subsets for each class
    data_class_0 = dataset[dataset[:, 1] == 0]
    data_class_1 = dataset[dataset[:, 1] == 1]

    # Split the data for each class into training and testing
    training_data_class_0 = data_class_0[:num_per_class_for_training]
    training_data_class_1 = data_class_1[:num_per_class_for_training]

    testing_data_class_0 = data_class_0[num_per_class_for_training:]
    testing_data_class_1 = data_class_1[num_per_class_for_training:]

    # Combine the training data from both classes
    training_data = np.concatenate((training_data_class_0,
training_data_class_1))

    # Combine the testing data from both classes
    testing_data = np.concatenate((testing_data_class_0,
testing_data_class_1))

    return training_data, testing_data
```

Explanation:

The **split_data_for_training_and_testing** function shuffles the dataset to ensure randomness. It then splits the data for each class into training and testing sets based on the specified number of samples per class for training. The function returns a tuple containing the training and testing data.

Section 3: Data Splitting

```
# Split the data for different sizes of the training set: 10, 20, and 30 per
class
training_data_10, testing_data_10 =
split_data_for_training_and_testing(combined_dataset, 10)
training_data_20, testing_data_20 =
split_data_for_training_and_testing(combined_dataset, 20)
training_data_30, testing_data_30 =
split_data_for_training_and_testing(combined_dataset, 30)
```

Explanation:

Here, the dataset is split into training and testing sets for three different sizes (10, 20, and 30 samples per class) using the previously defined function.

Section 4: Single Layer Perceptron Class

```
class SingleLayerPerceptron:
    def __init__(self, input_size, learning_rate=0.01, epochs=5,
initial_weights=None, initial_bias=None):
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = np.array(initial_weights) if initial_weights is not
None else np.random.rand(input_size)
        self.bias = initial_bias if initial_bias is not None else
np.random.rand()

    def activation(self, x):
        # Sigmoid function
        return 1 / (1 + np.exp(-x))

    def predict(self, inputs):
        # Weighted sum of inputs + bias (where inputs includes a '1' for the
bias weight)
        weighted_sum = np.dot(inputs, self.weights) + self.bias
        return self.activation(weighted_sum)

    def train(self, training_inputs, labels):
        # Ensure training_inputs is 2D
        training_inputs = training_inputs.reshape(-1, 1)
        # Add a '1' for the bias weight to each set of inputs
        training_inputs_with_bias = np.hstack((training_inputs,
np.ones((training_inputs.shape[0], 1))))

        for epoch in range(self.epochs):
            for inputs, label in zip(training_inputs_with_bias, labels):
                prediction = self.predict(inputs)
                # Update weights - the last weight is the bias weight
                # Gradient of the sigmoid function for error calculation
                error = label - prediction
                self.weights += self.learning_rate * error * inputs
                self.bias += self.learning_rate * error

            # Calculate accuracy and error for the epoch
            epoch_accuracy, epoch_error =
self.calculate_accuracy_and_error(training_inputs_with_bias, labels)
            print(f'Epoch {epoch+1}/{self.epochs} - Weights: {self.weights},
Bias: {self.bias}, Accuracy: {epoch_accuracy}, Error: {epoch_error}')

    def calculate_accuracy_and_error(self, test_inputs_with_bias,
test_labels):
        predictions = [self.predict(inputs) for inputs in
test_inputs_with_bias]
        # Binarize predictions for accuracy calculation
        binarized_predictions = [1 if p >= 0.5 else 0 for p in predictions]
        errors = test_labels - binarized_predictions
        accuracy = np.mean(np.array(binarized_predictions) == test_labels)

        return accuracy, np.sum(errors ** 2)
```

Explanation:

The **SingleLayerPerceptron** class is introduced, representing a basic single-layer perceptron. It has methods for initialization, activation, prediction, training, and accuracy calculation. This class will be used to create perceptron objects for different training set sizes.

Section 5: Perceptron Training

```
# Now, create a perceptron object with the same initial weights and bias for
different training set sizes
# Training data size of 10
perceptron_10 = SingleLayerPerceptron(input_size=2, learning_rate=0.01,
epochs=5, initial_weights=initial_weights, initial_bias=initial_bias)
# Training data size of 20
perceptron_20 = SingleLayerPerceptron(input_size=2, learning_rate=0.01,
epochs=5, initial_weights=initial_weights, initial_bias=initial_bias)
# Training data size of 30
perceptron_30 = SingleLayerPerceptron(input_size=2, learning_rate=0.01,
epochs=5, initial_weights=initial_weights, initial_bias=initial_bias)
```

Explanation:

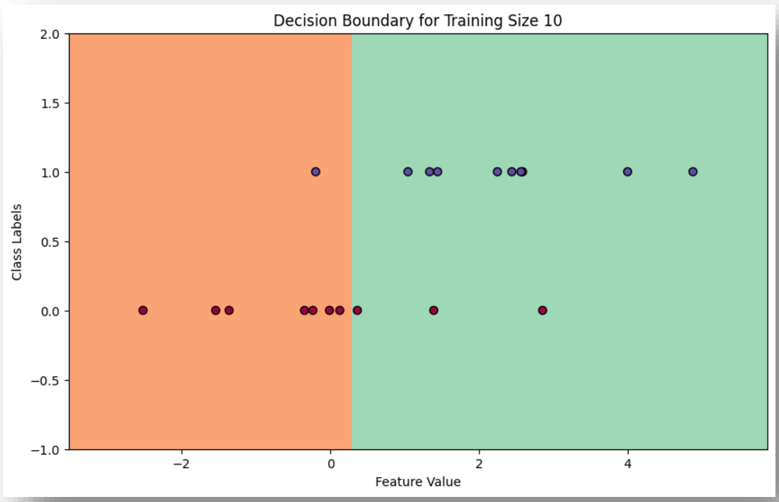
Perceptron objects are created for three different training set sizes (10, 20, and 30 samples per class). Each perceptron is initialized with specified parameters, and the train method is called to train the model on its respective training dataset.

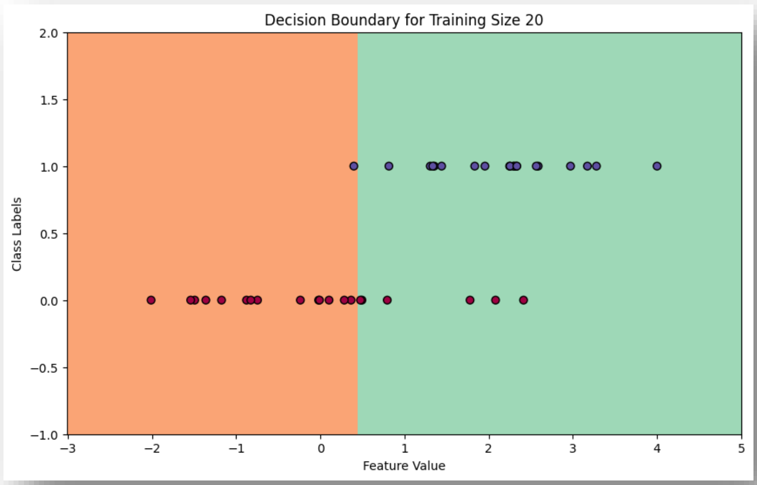
Section 6: Decision Boundary Plotting

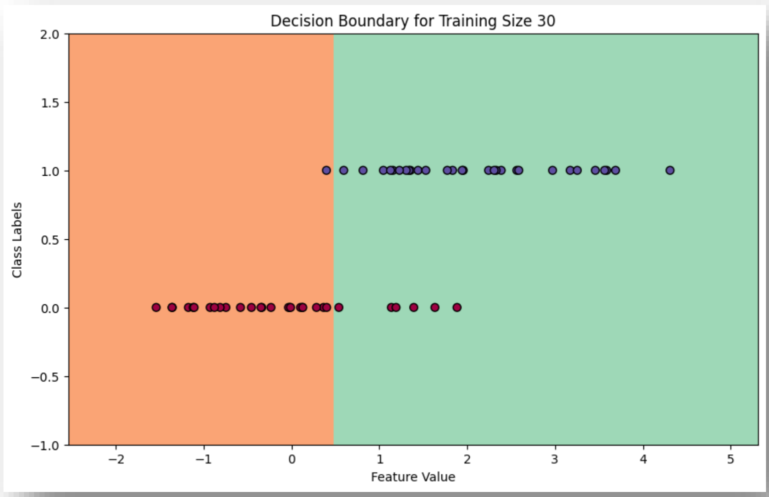
```
def plot_decision_boundary(perceptron, training_data, labels, title):
    # Set min and max values and give it some padding
    x_min, x_max = training_data[:, 0].min() - 1, training_data[:, 0].max() +
1
    y_min, y_max = labels.min() - 1, labels.max() + 1

    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 500), np.linspace(y_min,
y_max, 2))
    # Flatten the grid to pass into model
    grid = np.c_[xx.ravel(), np.ones_like(xx.ravel())]  # We only use the
first input feature and a bias term of 1

    # Predict the function value for the whole grid
    Z = perceptron.predict(grid)
    Z = Z.reshape(xx.shape)

    # Plot the contour and training examples
    plt.figure(figsize=(10, 6))
    plt.contourf(xx, yy, Z, alpha=0.8, levels=np.linspace(0, 1, 3),
cmap=plt.cm.Spectral)
    plt.scatter(training_data[:, 0], labels, c=labels, s=40,
cmap=plt.cm.Spectral, edgecolors='k')
    plt.title(title)
    plt.xlabel("Feature Value")
    plt.ylabel("Class Labels")
    plt.show()

# Plot the decision boundary for each perceptron
plot_decision_boundary(perceptron_10, training_data_10[:, :1],
training_data_10[:, -1], "Decision Boundary for Training Size 10")
plot_decision_boundary(perceptron_20, training_data_20[:, :1],
training_data_20[:, -1], "Decision Boundary for Training Size 20")
plot_decision_boundary(perceptron_30, training_data_30[:, :1],
training_data_30[:, -1], "Decision Boundary for Training Size 30")
```

Explanation:

A function, **plot_decision_boundary**, is defined to visualize the decision boundary of a perceptron. The decision boundary is plotted using Matplotlib, showing how the perceptron classifies different regions. This function is then applied to each perceptron with their corresponding training data, providing insights into the learned decision boundaries.


Decision Boundary for Training Size 10
Class Labels
Feature Value


Decision Boundary for Training Size 20
Class Labels
Feature Value


Decision Boundary for Training Size 30
Class Labels
Feature Value