DATA 2060 Final Project

Written by: Allison Gao, Jingxian Zhang

Link to the github repo: https://github.com/Jingxian2022/Multiclass-Classification-with-Logistic-Regression/tree/main

Overview of Multiclass Classification with Logistic Regression

Algorithm Overview

Multiclass classification with logistic regression extends the standard logistic regression approach, which traditionally handles binary classification, to address problems involving more than two classes.

Advantages

Both OvA and OvO break down the multiclass problem into smaller, simpler binary tasks. OvA is computationally efficient because it requires only k classifiers. It works particularly well for datasets with many classes, as it trains each classifier independently. On the other hand, OvO can do better when the inter-class boundaries are complex, as it compares classes pairwise and focuses on fine-grained distinctions. Another interesting aspect is their adaptability to imbalanced class distributions. OvA can prioritize improving the decision boundary for minority classes, while OvO ensures that each class pair is treated equally.

Disadvantages

While OvA and OvO are conceptually simple, implementing them comes with its own set of challenges. One major challenge is computational intensity. OvA requires k classifiers, which scales linearly with the number of classes. However, OvO requires k(k–1)/2 classifiers, which grows quadratically. For datasets with many classes, this becomes computationally expensive, especially when combined with large datasets. Another challenge is parameter tuning. Hyperparameters like the learning rate, batch size, and convergence threshold significantly impact the training process. A learning rate that's too high can cause the model to diverge, while one that's too low slows down convergence. Similarly, choosing the right batch size affects both training speed and stability. Incorrect tuning can lead to underfitting, where the model fails to learn meaningful patterns, or overfitting, where it performs well on training data but poorly on unseen data. Lastly, there are challenges in interpretation. In OvA, the decision boundaries of individual classifiers may overlap, leading to ambiguous predictions. In all-pairs, conflicting votes from pairwise classifiers can complicate the final decision. Aggregating these results into a single prediction requires careful consideration to avoid inconsistencies.

Multiclass Classification Basics

- Problem Definition
  Multiclass classification is the problem of classifying instances into one of three or more classes.

- Input and Output
  - Inputs X typically come from a feature space.
  - Outputs Y are from a finite set of labels $Y = \{1, 2, \ldots, k\}$, where k is the number of classes.

Multiclass Classification Strategies

In binary logistic regression, a linear function predicts the probability of the positive class using a logistic (sigmoid) function. Extending this to multiclass classification can be done using the following approaches:

One-vs-All Approach

One-vs-All involves training a single binary classifier for each class, with the samples of that class as positive samples and all other samples as negatives. The class with the highest probability score is selected for each input.

All-pairs Approach

All-pairs involves training $\binom{k}{2} = k(k-1)/2$ binary classifiers, each receives the samples of a pair of classes from the original training set, and learn to distinguish these two classes. For prediction, all $k(k-1)/2$ classifiers are applied to an unseen sample and the class that got the highest number of "+1" predictions gets predicted by the combined classifier.

Logistic Regression

Logistic Regression is a statistical method used for binary classification, predicting one of two possible outcomes based on input features. It estimates the parameters of a logistic model (the coefficients in the linear or non linear combinations) and transforms the linear combination of features using the sigmoid function, which maps any real-valued number into a value between 0 and 1. Logistic regression belongs to the family of generalized linear models and is widely used when the target variable is binary.

Loss Function

In logistic regression, the loss function quantifies the error between the predicted probabilities and the actual class labels. The most commonly used loss function for binary logistic regression is logistic loss(sometimes called cross-entropy loss). This function aims to minimize the log loss across all training observations. By penalizing incorrect predictions, the loss function encourages the model to produce probabilities that are closer to the true class labels.

Optimization

Gradient descent and its variants, like stochastic gradient descent (SGD), are common optimization techniques for logistic regression. Gradient descent works by computing the gradient (partial derivatives) of the loss function with respect to each parameter, and updating each parameter in the opposite direction of the gradient to minimize the loss.

Representation

Logistic regression

Logistic regression is common hypothesis class for classification

$$\mathcal{X} = \mathbb{R}^d \quad \mathcal{Y} = \{1, -1\}$$

Now we use a linear predictor that outputs a continuous value in [0, 1]

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

Where:

- $\mathbf{x} \in \mathcal{X}$ represents the input vector with dimension d
- $\mathbf{w}$ is the weight vector
- $\langle \mathbf{w}, \mathbf{x} \rangle$ denotes the dot product between $\mathbf{w}$ and $\mathbf{x}$

This linear predictor maps to:

$$h : \mathcal{X} \rightarrow [0, 1]$$

Loss

For binary classification, logistic regression uses the sigmoid function:

$$P(y = 1|x) = \sigma(w^T x + b)$$

Where:

- x is the input vector

- w is the weights
- b is the bias
- $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function

Binary Cross-Entropy Loss:

$$L(y, \hat{y}) = -(ylog(\hat{y}) + (1-y)log(1-\hat{y}))$$

Where:

- y is the true label (0 or 1)
- $\hat{y}$ is the predicted probability of the first class
- and $\hat{y} = \sigma(w^T x + b)$

One-vs-All: For one-vs-all, we have to train K different classifiers for each class so that each classifier k can learn to distinguish one class from all the others. The loss for the i-th example of classifier k is:

$$L_k(y^{(i)}, \hat{y}_k^{(i)}) = -[y_k^{(i)} log(\hat{y}_k^{(i)}) + (1 - y_k^{(i)}) log(1 - \hat{y}_k^{(i)})]$$

Where:

- $y_k^{(i)} = 1$ if the true class of the i-th example is class k, otherwise $y_k^{(i)} = 0$
- $\hat{y}_k^{(i)}$ is the predicted probability for class k

The overall class is determined by selecting the classifier that has the highest probability (or confidence).

All-Pairs: For All-Pairs, we have to train a classifier for every pair of classes instead of K classifiers in One-vs-All training. For K classes, we train $\frac{K(K-1)}{2}$ classifiers to distinguish between 2 classes for each classifier.

The loss function is still the binary cross-entropy loss and rewritten for the i-th example as:

$$L_{k,j}(y_{k,j}^{(i)}, \hat{y}_{k,j}^{(i)}) = -[y_{k,j}^{(i)} log(\hat{y}_{k,j}^{(i)}) + (1 - y_{k,j}^{(i)}) log(1 - \hat{y}_{k,j}^{(i)})]$$

Each classifier will vote for one of two classes and the overall class is the class that receives the most votes.

Optimizer

The optimizer used in this implementation is Stochastic Gradient Descent (SGD). SGD is an iterative optimization algorithm that updates the model parameters (weights and biases) by minimizing the loss function, specifically the cross-entropy loss for multiclass classification problems. Below are the psudo-codes for finding the optimizer for the one-vs-all and the all-pairs algorithms with SGD.

One-vs-All (OvR) with SGD

In One-vs-All, we train a separate binary classifier for each class. Each classifier learns to distinguish one class from all others.

```
Initialize parameters $\mathbf{w}$ for each class, learning rate $\alpha$, and batch size $b$
converge $=$ False

while not converge:
    epoch$+=1$
    Shuffle training examples

    for $i = 0, 1, \ldots, \left\lceil \frac{n_{\text{examples}}}{b} \right\rceil - 1$:     (iterate over batches)
        $X_{\text{batch}} = X[i \cdot b : (i+1) \cdot b]$    (select the $X$ in the current batch)
        $\mathbf{y}_{\text{batch}} = \mathbf{y}[i \cdot b : (i+1) \cdot b]$    (select the labels in the current batch)
        $\nabla L_{\mathbf{w}} = \mathbf{0}$    (initialize gradient matrix for each class)

        for each pair of training data $(x, y) \in (X_{\text{batch}}, \mathbf{y}_{\text{batch}})$:
            for $j = 0, 1, \ldots, n_{\text{classes}} - 1$:
                if $y = j$:
                    $\nabla L_{\mathbf{w}_j} += \Big(\sigma(\mathbf{w}_j^T x) - 1\Big) \cdot x$    (for correct class, reflects how much the p
                else:
                    $\nabla L_{\mathbf{w}_j} += \sigma(\mathbf{w}_j^T x) \cdot x$    (for other classes)
            $\mathbf{w}_j = \mathbf{w}_j - \alpha \cdot \frac{\nabla L_{\mathbf{w}_j}}{\text{len}(X_{\text{batch}})}$    (update weights for each class)

    Calculate this epoch loss
    if $|\text{Loss}(X, \mathbf{y})_{\text{this-epoch}} - \text{Loss}(X, \mathbf{y})_{\text{last-epoch}}| <$ CONV-THRESHOLD:
        converge $=$ True    (break the loop if loss converged)
```

Here, $\sigma(w_j^T x)$ gives the probability that x belongs to class j (treated as a binary classification for that specific class).

All Pairs (OvO) with SGD

In All Pairs, we train a separate binary classifier for each pair of classes, focusing only on the data points belonging to the two classes in each pair.

```
Initialize parameters $\mathbf{w}$ for each pair of classes, learning rate $\alpha$, and batch size $b$
converge $=$ False
```

```
while not converge:
    epoch$+=1$
    Shuffle training examples

    for $i = 0, 1, \ldots, \left\lceil \frac{n_{\text{examples}}}{b} \right\rceil - 1$:    (iterate over batches)
        $X_{\text{batch}} = X[i \cdot b : (i+1) \cdot b]$    (select the $X$ in the current batch)
        $\mathbf{y}_{\text{batch}} = \mathbf{y}[i \cdot b : (i+1) \cdot b]$    (select the labels in the current batch)
        for each unique pair of classes $(A, B)$:
            $\nabla L_{\mathbf{w}_{AB}} = \mathbf{0}$    (initialize gradient for each pair (A, B))
            for each $(x, y) \in (X_{\text{batch}}, \mathbf{y}_{\text{batch}})$:
                if $y = A$ or $y = B$:    (focus on examples for classes A and B)
                    if $y = A$:
                        $\nabla L_{\mathbf{w}_{AB}} += \left(\sigma(\mathbf{w}_{AB}^T x) - 1\right) \cdot x$    (for class A)
                else:
                        $\nabla L_{\mathbf{w}_{AB}} += \sigma(\mathbf{w}_{AB}^T x) \cdot x$    (for class B)

            $\mathbf{w}_{AB} = \mathbf{w}_{AB} - \alpha \cdot \frac{\nabla L_{\mathbf{w}_{AB}}}{\text{len}(X_{\text{batch}})}$    (update weights for the pair (A, B))

    Calculate this epoch loss
    if $|\text{Loss}(X, \mathbf{y})_{\text{this-epoch}} - \text{Loss}(X, \mathbf{y})_{\text{last-epoch}}| <$ CONV-THRESHOLD:
        converge = True    (break the loop if loss converged)
```

Run the environment test below and make sure all the requirements are met.

In [3]:
```python
from __future__ import print_function
from packaging.version import parse as Version
from platform import python_version

OK = '\x1b[42m[ OK ]\x1b[0m'
FAIL = "\x1b[41m[FAIL]\x1b[0m"

try:
    import importlib
except ImportError:
    print(FAIL, "Python version 3.12.5 is required,"
                " but %s is installed." % sys.version)

def import_version(pkg, min_ver, fail_msg=""):
    mod = None
    try:
        mod = importlib.import_module(pkg)
        if pkg in {'PIL'}:
            ver = mod.VERSION
        else:
            ver = mod.__version__
        if Version(ver) == Version(min_ver):
            print(OK, "%s version %s is installed."
                  % (lib, min_ver))
        else:
            print(FAIL, "%s version %s is required, but %s installed."
                  % (lib, min_ver, ver))
    except ImportError:
        print(FAIL, '%s not installed. %s' % (pkg, fail_msg))
    return mod


# first check the python version
pyversion = Version(python_version())

if pyversion >= Version("3.12.5"):
    print(OK, "Python version is %s" % pyversion)
elif pyversion < Version("3.12.5"):
    print(FAIL, "Python version 3.12.5 is required,"
                " but %s is installed." % pyversion)
else:
    print(FAIL, "Unknown Python version: %s" % pyversion)


print()
requirements = {'numpy': "2.0.1",'sklearn': "1.5.1",
                'pandas': "2.2.2",'pytest': "7.4.4",
                 'imblearn': "0.12.4"}

# now the dependencies
for lib, required_version in list(requirements.items()):
    import_version(lib, required_version)
```

```
[ OK ] Python version is 3.12.7

[ OK ] numpy version 2.0.1 is installed.
[ OK ] sklearn version 1.5.1 is installed.
[ OK ] pandas version 2.2.2 is installed.
[ OK ] pytest version 7.4.4 is installed.
[ OK ] imblearn version 0.12.4 is installed.
```

Model

In [4]:
```
import numpy as np

def sigmoid(x):
    '''
    Apply sigmoid function to an array.
    @params:
        x: The input array.
    @return:
        An array with the sigmoid function applied elementwise.
    '''
    return 1 / (1 + np.exp(-x))

class MulticlassLogisticRegression:
    '''
    Multiclass Logistic Regression with One-vs-All (OvA) and All-Pairs (OvO)
    trained using stochastic gradient descent.
    '''
    def __init__(self, n_features, n_classes, batch_size=32, conv_threshold=
        '''
        Initializes the Multiclass Logistic Regression classifier.
        @attrs:
            n_features: Number of features in the dataset.
            n_classes: Number of unique classes.
            weights: Model weights, initialized to zeros.
            strategy: Multiclass strategy ('one-vs-all' or 'all-pairs').
            alpha: Learning rate for SGD.
        '''
        self.n_classes = n_classes
        self.n_features = n_features
        self.strategy = strategy
        self.weights = None  # Initialize dynamically based on the strategy
        self.alpha = 0.1
        self.batch_size = batch_size
        self.conv_threshold = conv_threshold

    def train(self, X, Y):
        '''
        Trains the model using stochastic gradient descent.
        Supports both One-vs-All and All-Pairs strategies.
        @params:
            X: 2D Numpy array where each row is an example, padded with one
            Y: 1D Numpy array of labels for each example.
        @return:
            Number of epochs taken to converge.
        '''
        if self.strategy == 'one-vs-all':
            self._train_one_vs_all(X, Y)
        elif self.strategy == 'all-pairs':
            self._train_all_pairs(X, Y)
        else:
            raise ValueError(f"Invalid strategy: {self.strategy}. Use 'one-v

    def _train_one_vs_all(self, X, Y):
        '''
        Trains the model using the One-vs-All (OvA) strategy.
        Each class is treated as a binary classification problem against all
        and a separate weight vector is trained for each class.

        @params:
            X: A 2D Numpy array where each row is a feature vector of an exa
               padded with one column for the bias term.
            Y: A 1D Numpy array of class labels for each example in X.

            Labels are converted into binary format for each class during tr
        '''
        self.weights = np.zeros((self.n_classes, self.n_features + 1))
        for class_label in range(self.n_classes):
            binary_Y = (Y == class_label).astype(int) #if label matches ther
            self._train_binary_class(X, binary_Y, class_label)

    def _train_all_pairs(self, X, Y):
        '''
        Trains the model using the All-Pairs (OvO) strategy.
        Each pair of classes is treated as a binary classification problem,
        and a separate weight vector is trained for each class pair.

        @params:
            X: A 2D Numpy array where each row is a feature vector of an exa
               padded with one column for the bias term.
            Y: A 1D Numpy array of class labels for each example in X.

            Only examples belonging to any two distinct classes are used for
        '''
        #The weights for all binary classifiers are stored in a dictionary
        #Keys: Tuples representing a pair of classes (e.g., (0, 1), (0, 2))
        #Values: Weight vectors for the corresponding classifier.
        self.weights = {}

        #a total of n(n-1)/2 classifiers are trained
        for i in range(self.n_classes):
            for j in range(i + 1, self.n_classes):
                #identifies the indices of examples where the label is eithe
                indices = np.where((Y == i) | (Y == j))[0]
                X_subset = X[indices]
                Y_subset = Y[indices]

                #labels converted into binary format
                binary_Y = (Y_subset == i).astype(int) #class i = 1, class j
                self.weights[(i, j)] = np.zeros(self.n_features + 1)
                self._train_binary_class(X_subset, binary_Y, (i, j))

    def _train_binary_class(self, X, Y, label):
        '''
        Trains a binary logistic regression model for a specific class or pa
        @params:
            X: A 2D Numpy array where each row contains a feature vector for
            Y: A 1D Numpy array with binary labels (0 or 1) corresponding to
            label: An integer (for OvA) or tuple (for OvO) representing the
        @return:
            Number of epochs taken to converge during the training process.
        '''
        num_examples = X.shape[0]
        epoch = 0
        converged = False
        last_loss = float('inf')
        while not converged:
            epoch += 1
            indices = np.arange(num_examples)
            np.random.shuffle(indices)
            X = X[indices]
            Y = Y[indices]

            for i in range(int(np.ceil(num_examples/self.batch_size))):
                batch_X = X[i * self.batch_size:(i + 1) * self.batch_size]
                batch_Y = Y[i * self.batch_size:(i + 1) * self.batch_size]

                grad_w = np.zeros_like(self.weights[label] if isinstance(lab
                for x, y in zip(batch_X, batch_Y):
                    raw = np.dot(self.weights[label], x)
                    prob = sigmoid(raw)  # Probability of positive class
                    grad_w += (prob - y) * x

                grad_w /= len(batch_X)
                self.weights[label] -= self.alpha * grad_w

            this_loss = self.loss(X, Y, label)
            if abs(this_loss - last_loss) < self.conv_threshold:
                converged = True

            last_loss = this_loss

        return epoch

    def predict(self, X):
        '''
        Predicts the class for each example in X.
        @params:
            X: 2D Numpy array of examples, padded with one column for bias.
        @return:
            1D Numpy array of predicted class labels.
        '''
        if self.strategy == 'one-vs-all':
            return self._predict_one_vs_all(X)
        elif self.strategy == 'all-pairs':
            return self._predict_all_pairs(X)
        else:
            raise ValueError(f"Invalid strategy: {self.strategy}. Use 'one-v

    def _predict_one_vs_all(self, X):
        '''
        Predicts the class labels for a given dataset using the One-vs-All (
        @params:
            X: A 2D Numpy array where each row is a feature vector of an exa
        @return:
            A 1D Numpy array containing the predicted class labels for each
            Each label corresponds to the class with the highest probability
        '''
        probabilities = np.dot(X, self.weights.T)
        return np.argmax(probabilities, axis=1)

    def _predict_all_pairs(self, X):
        '''
        Predicts the class labels for a given dataset using the All-Pairs (C
        @params:
            X: A 2D Numpy array where each row is an example, padded with or
        @return:
            A 1D Numpy array of predicted class labels for each example in X
        '''
        votes = np.zeros((X.shape[0], self.n_classes))
        for (i, j), weight in self.weights.items():
            #raw score for the (i,j) classifier
            raw = X @ weight
            #1 or class i if >= 0, 0 or class j if < 0, decision boundary
            predictions = (raw >= 0).astype(int)
            votes[:, i] += predictions
```

```
            votes[:, j] += (1 - predictions)
        #select class with the most votes
        return np.argmax(votes, axis=1)

    def loss(self, X, Y, label):
        '''
        Computes the log loss for the model.
        @params:
            X: 2D Numpy array of examples, padded with one column for bias.
            Y: 1D Numpy array of labels for each example.
            label: Binary classification label or class pair.
        @return:
            Average log loss.
        '''
        total_loss = 0
        num_examples = X.shape[0]

        if isinstance(label, tuple):
            # Binary classification loss (OvO for a specific class pair)
            for x, y in zip(X, Y):
                raw = np.dot(self.weights[label], x)  # Raw score for the Ov
                prob = sigmoid(raw)  # Sigmoid for binary probabilities
                if y == 1:  # Positive class in the pair
                    total_loss += -np.log(prob + 1e-6)
                else:  # Negative class in the pair
                    total_loss += -np.log(1 - prob + 1e-6)
        else:
            # Binary classification loss (OvA for a specific class)
            for x, y in zip(X, Y):
                raw = np.dot(self.weights[label], x)  # Raw score for the Ov
                probability = sigmoid(raw)  # Sigmoid for binary probabiliti
                if y == 1:  # Positive class
                    total_loss += -np.log(probability + 1e-6)
                else:  # Negative class (all other classes)
                    total_loss += -np.log(1 - probability + 1e-6)

        return total_loss / num_examples


    def accuracy(self, X, Y):
        '''
        Computes accuracy on a given dataset.
        @params:
            X: 2D Numpy array of examples, padded with one column for bias.
            Y: 1D Numpy array of true labels.
        @return:
            Float value representing accuracy.
        '''
        predictions = self.predict(X)
        return np.mean(predictions == Y)
```

Check Model

Binary classification check model:

In [5]:
```
import random
import pytest
from sklearn.multiclass import OneVsOneClassifier
from sklearn.multiclass import OneVsRestClassifier
from sklearn.metrics import log_loss
from sklearn.linear_model import SGDClassifier
# set random seed for testing purposes
random.seed(0)
np.random.seed(0)

# create test data
x_bias = np.array([[0,4,1], [0,3,1], [5,0,1], [4,1,1], [0,5,1]])
x = x_bias[:,:-1]
y = np.array([0,0,1,1,0])
x_bias_test = np.array([[0,0,1], [-5,3,1], [9,0,1], [1,0,1], [6,-7,1]])
x_test = x_bias_test[:,:-1]
y_test = np.array([0,0,1,0,1])

# create binary classification model
binary_test_model = MulticlassLogisticRegression(2, 2)
binary_test_model.weights = np.zeros((2, 3))

# test model loss
assert binary_test_model.loss(x_bias, y, 1) == pytest.approx(log_loss(y,sigm

binary_test_model._train_binary_class(x_bias, y, label=1)

# create one-vs-all binary classification model
one_vs_all_binary_test_model = MulticlassLogisticRegression(2, 2)
one_vs_all_binary_test_model.weights = np.zeros((2, 3))
one_vs_all_binary_test_model.train(x_bias, y)

# create all-pairs binary classification model
all_pairs_binary_test_model = MulticlassLogisticRegression(2, 2, strategy="a
all_pairs_binary_test_model.weights = np.zeros((2, 3))
all_pairs_binary_test_model.train(x_bias, y)

# Test Train Model and Checks Model Weights
assert np.allclose(binary_test_model.weights[1], one_vs_all_binary_test_mode
assert np.allclose(one_vs_all_binary_test_model.weights[0], all_pairs_binary

# Test Model Accuracy
assert binary_test_model.accuracy(x_bias_test, y_test) == one_vs_all_binary_
assert one_vs_all_binary_test_model.accuracy(x_bias_test, y_test) == all_pai

# Test model predict
assert (all_pairs_binary_test_model.predict(x_bias_test) == one_vs_all_binar

# create sklearn OneVsRestClassifier with SGDClassifier
sgd_logistic = SGDClassifier(
    loss='log_loss', penalty=None, alpha=0, max_iter=1000, tol=1e-4, shuffle
)
ova_model = OneVsRestClassifier(sgd_logistic)
ova_model.fit(x, y)
sklearn_ova_model_weight=[]
```

```
for i, estimator in enumerate(ova_model.estimators_):
    sklearn_ova_model_weight.append(np.hstack([estimator.coef_, estimator.ir

# create sklearn OneVsOneClassifier with SGDClassifier
sgd_logistic = SGDClassifier(loss='log_loss', penalty=None, alpha=0, max_ite
ovo_model = OneVsOneClassifier(sgd_logistic)
ovo_model.fit(x, y)
sklearn_ovo_model_weight = []
for i, estimator in enumerate(ovo_model.estimators_):
    sklearn_ovo_model_weight.append(np.hstack([estimator.coef_, estimator.ir
sklearn_ovo_model_weight *= -1

# test model accuracy equals to library function accuracy
assert ova_model.score(x_test,y_test) == one_vs_all_binary_test_model.accura
assert ovo_model.score(x_test,y_test) == all_pairs_binary_test_model.accurac

# test model predict equals to library function predict
assert (ova_model.predict(x_test) == one_vs_all_binary_test_model.predict(x_
assert (ovo_model.predict(x_test) == all_pairs_binary_test_model.predict(x_b

# test model weight equals to library function weight
assert np.allclose(one_vs_all_binary_test_model.weights[1], sklearn_ova_mode
for a, b in zip(all_pairs_binary_test_model.weights[(0,1)], sklearn_ovo_mode
    assert a == pytest.approx(b, rel=.01)
```

Multiclass classification check model:

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

# create test data
x_bias2 = np.array([[0,0,1], [0,3,1], [4,0,1], [6,1,1], [0,1,1], [0,4,1]])
y2 = np.array([0,1,2,2,0,1])
x_bias_test2 = np.array([[0,0,1], [-5,3,1], [9,0,1], [1,0,1]])
y_test2 = np.array([0,1,2,0])
x2 = x_bias2[:,:-1]
x2_test = x_bias_test2[:,:-1]

# create one-vs-all classification with 2 features and 3 classes
test_model_one_vs_all_1 = MulticlassLogisticRegression(2, 3)
test_model_one_vs_all_1.train(x_bias2, y2)

# create all-pairs classification with 2 features and 3 classes
test_model_all_pairs_1 = MulticlassLogisticRegression(2,3,strategy="all-pair
test_model_all_pairs_1.train(x_bias2, y2)

# create sklearn OneVsRestClassifier
sgd_logistic = SGDClassifier(loss='log_loss', penalty=None, alpha=0, max_ite
ova_model = OneVsRestClassifier(sgd_logistic)
ova_model.fit(x2, y2)
ova_model_weight = []
for i, estimator in enumerate(ova_model.estimators_):
    ova_model_weight.append(np.hstack([estimator.coef_, estimator.intercept_
# create sklearn OneVsOneClassifier
sgd_logistic = SGDClassifier(loss='log_loss', penalty=None, alpha=0, max_ite
ovo_model = OneVsOneClassifier(sgd_logistic)
ovo_model.fit(x2, y2)
ovo_model_weight = []
for i, estimator in enumerate(ovo_model.estimators_):
    ovo_model_weight.append(np.hstack([estimator.coef_, estimator.intercept_

# test model accuracy with multiple class
assert ova_model.score(x2_test,y_test2) == test_model_one_vs_all_1.accuracy(
assert ovo_model.score(x2_test,y_test2) == test_model_all_pairs_1.accuracy(x

# test model predict with multiple class
assert (ova_model.predict(x2_test) == test_model_one_vs_all_1.predict(x_bias
assert (ovo_model.predict(x2_test) == test_model_all_pairs_1.predict(x_bias_

# test model weight with multiple class
for a, b in zip(test_model_one_vs_all_1.weights, ova_model_weight):
    assert np.allclose(a, b, atol=0.5)
for a, b in zip(test_model_all_pairs_1.weights.values(), ovo_model_weight):
    a_ = -a
    assert (np.allclose(a, b, atol=0.1) | np.allclose(a_, b, atol=0.1))
```

In [7]:
```
# set random seed for testing purposes
random.seed(0)
np.random.seed(0)

# create test data
x_bias3 = np.array([[8, 1, 1],[1, 2, 1],[5, 5, 1],[1, 1, 1],[6, 5, 1],[9, 0,
y3 = np.array([2, 0, 1, 0, 1, 2])
x_bias_test3 = np.array([[1, 3, 1],[8, 2, 1],[5, 6, 1],[1, 1, 1]])
y_test3 = np.array([0, 2, 1, 0])
x3 = x_bias3[:, :-1]
x3_test = x_bias_test3[:, :-1]

# create one-vs-all classification with 2 features and 3 classes
test_model_one_vs_all_1 = MulticlassLogisticRegression(2, 3)
test_model_one_vs_all_1.train(x_bias3, y3)

# create all-pairs classification with 2 features and 3 classes
test_model_all_pairs_1 = MulticlassLogisticRegression(2,3,strategy="all-pair
test_model_all_pairs_1.train(x_bias3, y3)

# create sklearn OneVsRestClassifier
sgd_logistic = SGDClassifier(loss='log_loss', penalty=None, alpha=0, max_ite
ova_model = OneVsRestClassifier(sgd_logistic)
ova_model.fit(x3, y3)
ova_model_weight = []
for i, estimator in enumerate(ova_model.estimators_):
    ova_model_weight.append(np.hstack([estimator.coef_, estimator.intercept_

# create sklearn OneVsOneClassifier
sgd_logistic = SGDClassifier(loss='log_loss', penalty=None, alpha=0, max_ite
ovo_model = OneVsOneClassifier(sgd_logistic)
ovo_model.fit(x3, y3)
ovo_model_weight = []
```

```
for i, estimator in enumerate(ovo_model.estimators_):
    ovo_model_weight.append(np.hstack([estimator.coef_, estimator.intercept_

# test model accuracy with multiple class
assert ova_model.score(x3_test,y_test3) == test_model_one_vs_all_1.accuracy(
assert ovo_model.score(x3_test,y_test3) == test_model_all_pairs_1.accuracy(x

# test model predict with multiple class
assert (ova_model.predict(x3_test) == test_model_one_vs_all_1.predict(x_bias
assert (ovo_model.predict(x3_test) == test_model_all_pairs_1.predict(x_bias_

# test model weight with multiple class
for a, b in zip(test_model_one_vs_all_1.weights, ova_model_weight):
    assert np.allclose(a, b, atol=0.8)
for a, b in zip(test_model_all_pairs_1.weights.values(), ovo_model_weight):
    a_ = -a
    assert (np.allclose(a, b, atol=0.1) | np.allclose(a_, b, atol=0.1))
```

Edge case:

In [8]:
```
# test raise error with invalid input
classifier = MulticlassLogisticRegression(x_bias,y,strategy='one-vs-one')
with pytest.raises(ValueError, match="Invalid strategy"):
    classifier.predict(x_bias)
```

Previous work introduction

- The author demonstrates multinomial logistic regression using Scikit-learn on the Dry Bean Dataset, which is a publicly available dataset from Kaggle. The dataset comprises various features of different dry bean types, including Seker, Barbunya, Bombay, Cali, Horoz, Sira, and Dermason.
- Beginning by exploring the dataset, the author checks the absence of null values, which simplifies the analysis. He identifies an imbalance in the class distribution and addresses it through random under-sampling to achieve a more balanced dataset. After encoding the categorical 'Class' labels numerically, he examines correlations between features, deciding to remove 'ConvexArea' and 'EquivDiameter' due to their high correlation, to prevent potential overfitting.
- The data is then split into training and testing sets, with scaling applied to standardize the features. The author trains a multinomial logistic regression model using Scikit-learn's `LogisticRegression` class, specifying the 'multinomial' option for multi-class classification. He evaluates the model's performance using a confusion matrix and accuracy score, providing insights into its effectiveness in classifying the different types of dry beans.

In [9]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
from imblearn.under_sampling import RandomUnderSampler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, accuracy_score, ConfusionMatri
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import pandas as pd

DATA_FILE = '../data/Dry_Bean.csv'

def get_data(file_path):
    df = pd.read_csv(file_path)
    df['Class'].unique()
    if df.isnull().sum().sum() > 0:
        print("There are missing values in the dataset.")
    else:
        print("No missing values in the dataset.")

    undersample = RandomUnderSampler(random_state=42)

    X = df.drop('Class', axis=1)
    y = df.Class

    X_over, y_over = undersample.fit_resample(X, y)

    y_over.replace(list(np.unique(y_over)), [0, 1, 2, 3, 4, 5, 6], inplace=T
    df_dea = X_over
    df_dea['Class'] = y_over

    X_train, X_test, y_train, y_test = train_test_split(X_over, y_over, ranc

    # scale our data
    st_x = StandardScaler()
    X_train = st_x.fit_transform(X_train)
    X_test = st_x.transform(X_test)
    y_train = y_train.to_numpy()
    y_test = y_test.to_numpy()

    return X_train, y_train, X_test, y_test

def test_dry_bean_ovr():
    X_train, Y_train, X_test, Y_test = get_data(DATA_FILE)
    num_features = X_train.shape[1]
    NUM_CLASS = 7
    BATCH_SIZE = 100
    CONV_THRESHOLD = 1e-4

    X_train_b = np.hstack((X_train, np.ones((X_train.shape[0], 1))))
    X_test_b = np.hstack((X_test, np.ones((X_test.shape[0], 1))))

    model = MulticlassLogisticRegression(num_features, NUM_CLASS, BATCH_SIZE
    model.train(X_train_b, Y_train)
    acc = model.accuracy(X_test_b, Y_test)
    print("One-vs-all model accuracy: ",acc)

    sgd_logistic = SGDClassifier(
        loss='log_loss', penalty=None, alpha=0, max_iter=1000, tol=1e-4, shu
    )
    ova_model = OneVsRestClassifier(sgd_logistic)
    ova_model.fit(X_train, Y_train)
    print("Previous work accuracy: ",ova_model.score(X_test,Y_test))

def test_dry_bean_ovo():
    X_train, Y_train, X_test, Y_test = get_data(DATA_FILE)
    num_features = X_train.shape[1]
    NUM_CLASS = 7
    BATCH_SIZE = X_train.shape[0]+1
    CONV_THRESHOLD = 1e-4

    X_train_b = np.hstack((X_train, np.ones((X_train.shape[0], 1))))
    X_test_b = np.hstack((X_test, np.ones((X_test.shape[0], 1))))

    model = MulticlassLogisticRegression(num_features, NUM_CLASS, BATCH_SIZE
    model.train(X_train_b, Y_train)
    acc = model.accuracy(X_test_b, Y_test)
    print("All-pairs model accuracy: ",acc)

    sgd_logistic = SGDClassifier(
        loss='log_loss', penalty=None, alpha=0,
        max_iter=1000, tol=1e-4, shuffle=True,
        random_state=0, learning_rate='constant',
        eta0=0.03, early_stopping=False, epsilon=1e-6, average=X_train.shape
    )
    ovo_model = OneVsOneClassifier(sgd_logistic)
    ovo_model.fit(X_train, Y_train)
    print("Previous work accuracy: ",ovo_model.score(X_test,Y_test))

random.seed(0)
np.random.seed(0)
test_dry_bean_ovr()
test_dry_bean_ovo()
```

```
No missing values in the dataset.
/var/folders/8p/d41jll4x0c34grr0mk71fz580000gn/T/ipykernel_13066/1909965700.
py:29: FutureWarning: Downcasting behavior in `replace` is deprecated and wi
ll be removed in a future version. To retain the old behavior, explicitly ca
ll `result.infer_objects(copy=False)`. To opt-in to the future behavior, set
`pd.set_option('future.no_silent_downcasting', True)`
  y_over.replace(list(np.unique(y_over)), [0, 1, 2, 3, 4, 5, 6], inplace=Tru
e)
One-vs-all model accuracy:  0.9808481532147743
Previous work accuracy:  0.9863201094391245
No missing values in the dataset.
/var/folders/8p/d41jll4x0c34grr0mk71fz580000gn/T/ipykernel_13066/1909965700.
py:29: FutureWarning: Downcasting behavior in `replace` is deprecated and wi
ll be removed in a future version. To retain the old behavior, explicitly ca
ll `result.infer_objects(copy=False)`. To opt-in to the future behavior, set
`pd.set_option('future.no_silent_downcasting', True)`
  y_over.replace(list(np.unique(y_over)), [0, 1, 2, 3, 4, 5, 6], inplace=Tru
e)
```

```
All-pairs model accuracy:  0.9835841313269493
Previous work accuracy:  0.9931600547195623
```

We have successfully reproduce results from previous work.

References

- Hergir, D., n.d. Logistic Regression. [GitHub repository] Available at: https://github.com/danhergir/Logistic_regression [Accessed 15 December 2024].
- Hergir, D., 2022. Implementing Multi-Class Logistic Regression with Scikit-Learn. [Medium article] Available at: https://danhergir.medium.com/implementing-multi-class-logistic-regression-with-scikit-learn-53d919b72c13 [Accessed 15 December 2024].
- Dua, D. and Graff, C., 2019. UCI Machine Learning Repository: Dry Bean Dataset. [Dataset] Irvine, CA: University of California, School of Information and Computer Science. Available at: https://archive.ics.uci.edu/ml/datasets/Dry+Bean+Dataset [Accessed 15 December 2024].