DATA2060 Final Project

Jessica Wan, Michael Lu, Angela Zhu, Qiming Fang

October 2024

GitHub link: `https://github.com/mdlu02/DATA2060_Final/tree/main`

1 Overview of Multi-class Classification

Multi-class classification involves the use of a binary classification algorithm to determine the most probable decision from multiple class possibilities. Two main techniques utilizing logistical regression for multi-class classification are the one-vs-all and all-pairs algorithms.

1.1 Representation

Both one-vs-all and all-pairs multi-class classification algorithms are built using an ensemble of simple binary classification models. Any type of binary classification model can be used for the basic models. For our project we will be using the logistic regression binary classification model we completed in homework 3. The hypothesis of binary logistic regression of class j (or the probability of datapoint x being in class j for j either = 0 or 1) is given below, which is a variation of the one from our class slides.

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

The weights matrix has size 2xd where d is the number of features, and the first row of the weights matrix = - the second row of the weights matrix (so $h_{w_0}(x) = 1 - h_{w_1}(x)$). w_j represents the jth row of our weights matrix. The sigmoid function here is equivalent to the softmax function for two classes.

1.1.1 One-vs-all

This algorithm trains n binary classification models where n is the number of classes. Each of these model predicts whether a sample belongs to the n^{th} class. When predicting, the model selects the class with the highest predicted probability. For example, if we have 3 classes A, B, C, we would train 3 binary classification models as follows:

- Model 1: A vs. not A (B or C).

- Model 2: B vs. not B (A or C).
- Model 3: C vs. not C (A or B).

For a given sample, if the binary classification predictions are 0.76, 0.43, and 0.5 for classifiers 1, 2, and 3 respectively, the model would predict class A since classifier 1 had the highest predicted probability of the sample belonging to its predicted class.

Here the pseudo code for the algorithm (Shalev-Shwartz and Ben-David, 2014, pg. 191):

```
input:
    training set $S = (x_1, y_1), ..., (x_m, y_m)$
    algorithm for binary classification model
foreach $i \in \mathcal{Y}$
    let $S_i = \big((x_1, (1)^{\mathbb{1}[y_1 \neq i]}), ..., (x_m, (1)^{\mathbb{1}[y_m \neq i]})\big)$
    let $h_i = A(S_i)$
output:
    the multi-class hypothesis defined by $h(x) \in \text{argmax}_{i \in \mathcal{Y}} h_i(x)$
```

One-vs-all is a very simple algorithm to implement and can provide explainable results compared to some more complex models. However, for tasks with large numbers of classes, this approach can be computationally expensive as each class would require its own binary classification model. Additionally, large numbers of classes will lead to increasingly imbalanced data for each binary classification model. If we have five classes with a perfectly balanced distribution, each binary classification model will have to train on a dataset with a 20/80 imbalance.

1.1.2 All-pairs

This algorithm uses binary classification on all pairs of possible classes, hence if we have n classes, then the algorithm would train $\binom{n}{2}$ binary classification models. For each pair of classes, call this (i, j) with i and j as different classes/labels, we would take all samples with label either i or j, then train our binary classification model on this subset of data to predict whether or not a sample belongs to class i or class j. In our binary classification model we can label class i as 1 and class j as -1. When predicting, for each class i we will essentially see how often each binary classification for i against every other class j will predict i, and the class i with the highest winning rate will be our final prediction.

Here is the pseudocode for the algorithm (Shalev-Shwartz and Ben-David, 2014, pg. 192):

```
input:
    training set $S = (x_1, y_1), ..., (x_m, y_m)$
    algorithm for binary classification A
```

```
foreach $i, j \in \mathcal{Y}, i < j$:
    $S_{i,j}$ initialize to be empty
    for t=1,...,m:
        if $y_t = i$, add $(x_t, 1)$ to $S_{i,j}$
        if $y_t = j$, add $(x_t, -1)$ to $S_{i,j}$        let $h_{i,j} = A(S_{i,j})$
output:
    the multi-class hypothesis defined by $h(x) \in argmax_{i\in\mathcal{Y}}(\sum_{j\in\mathcal{Y}} sign(j - i)h_{i,j}(x))$
```

Compared to the one-vs-all approach, the all-pairs algorithm is faster to train on each binary classification model as it trains on a smaller portion of data than one-vs-all, however all-pairs takes longer to compute our final prediction. With increasing number of classes, this method can also be computationally expensive, as we have to train $\binom{n}{2}$ models, where n is our number of classes.

1.2 Loss

We use log loss, also known as logistic loss, as the loss function for binary classification. Log loss is given by the equation below (Zsom, 2024).

$$L(h_w) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\hat{y}^{(i)} + (1 - y^{(i)})\log(1 - \hat{y}^{(i)})\right]$$

where m is the number of observations, $y^{(i)}$ is the true label for the ith sample. $\hat{y}^{(i)}$ is the predicted probability that output is 1 for the ith sample.

1.3 Optimizer

In order to optimize the weights to minimize loss with logistic regression, we must perform stochastic gradient descent (SGD) for each batch of training data used to learn our model. Stochastic gradient descent follows the form (Zsom, 2024):

$$w = w - \alpha\nabla L_s(h_w)$$

where w represents the weighting coefficient vector for each feature, α determines the step size of our descent, and $\nabla L_s(h_w)$ is the gradient of the loss function associated with the probability that a data point x belongs to a class s. The batch size, which ranges from (1, size of training set), can be determined according to the size of the data set and what is optimal for the model's time-space expectations. Smaller batch sizes result in noisy estimates, but faster convergence. Larger batch sizes are slower, but more accurate.

We continue the descent until reaching a convergence threshold ϵ, at which point we can assume we have reached the local minimum of the loss function.

The algorithm for SGD to find the minimum loss until convergence is described below:

input:
  training examples X,Y of size S
  step size α
  batch size $b < S$

```
converge = False
while not converge:
    shuffle X,Y
    prev_epoch_loss = loss(X,Y)
    for each batch:
        Lw = zeros((num_classes, num_features + 1))
        for x,y in batch:
            probabilities = softmax(x)
            for class j in classes:
                h_wx = probabilities[j]
                if y == j: Lw[j] += (h_wx - 1)*x
                else: Lw[j] += (h_wx)*x
        w -= alpha * Lw / num_examples_in_batch
    curr_epoch_loss = loss(X,Y)
    converge = diff(curr_epoch_loss, prev_epoch_loss) < threshold
```

multiclass_regression_final-v2

December 15, 2024

1 DATA 2060 Final Notebook

Link to Github Repo: https://github.com/mdlu02/DATA2060_Final/tree/main

2 Overview of Multiclass Classification (One vs All and All Pairs) Algorithms

[22]:
```python
from IPython.display import IFrame

IFrame("../data2060_final_proj.pdf", width=1000, height=1000)
```

[22]:
```
<IPython.lib.display.IFrame at 0x105d67260>
```

3 Model

Includes the underlying binary logistic regression class, along with the OneVsAll and AllPairs classes.

[25]:
```python
import pytest
import random
import numpy as np
import pandas as pd
import itertools as it
from sklearn.datasets import load_iris
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import normalize
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import SGDClassifier
from sklearn.multiclass import OneVsOneClassifier, OneVsRestClassifier

SEED = 42
random.seed(SEED)

def softmax(x):
    '''
    Apply softmax to an array
    @params:
        x: the original array
    @return:
        an array with softmax applied elementwise.
    '''
    inner = np.array(x - np.max(x))
    e = np.exp(inner.astype(np.float64))
    return (e + 1e-6) / (np.sum(e) + 1e-6)

class LogisticRegression:
    '''
    Multiclass Logistic Regression that learns weights using
    stochastic gradient descent. In our case number of classes will always be 2.
    '''
    def __init__(self, n_features, n_classes, batch_size, conv_threshold):
        '''
        Initializes a LogisticRegression classifer.
        @attrs:
            n_features: the number of features in the classification problem + 1␣
 ↪for bias
            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.n_classes = n_classes
        self.n_features = n_features
        self.weights = np.zeros((n_classes, n_features))
        self.alpha = 0.03  # DO NOT TUNE THIS PARAMETER
        self.batch_size = batch_size
        self.conv_threshold = conv_threshold

    def fit(self, X, Y):
        '''
        Trains the model using 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:
            num_epochs: integer representing the number of epochs taken to reach␣
 ↪convergence
        '''

        w = self.weights
        alpha = self.alpha
        b = self.batch_size
        num_epoch = 0
        converge = False
        X_and_Y = np.hstack([X,np.zeros([1,len(Y)]).T]) # so when we shuffle the␣
↪labels and examples still match up
        X_and_Y[:,-1] = Y
        while converge == False:
            num_epoch += 1
            np.random.shuffle(X_and_Y) #shuffle training data, so X_and_Y is now␣
↪a shuffled matrix
            shuffled_X = X_and_Y[:,:-1]
            shuffled_Y = X_and_Y[:,-1]
            last_epoch_loss = self.loss(X,Y) #calculate loss with our current␣
↪weights
            for i in range(0,int(np.ceil(len(Y)/b))):
                X_batch = shuffled_X[i*b:(i+1)*b,:]
                Y_batch = shuffled_Y[i*b:(i+1)*b] #1xn array
                deriv_L_w = np.zeros(np.shape(w)) #our derivative L_w matrix
                for data_point_row in range(0,len(Y_batch)):
                    for j in range(0,self.n_classes):
                        x = X_batch[data_point_row,:]
                        if Y_batch[data_point_row] == j:
                            deriv_L_w[j,:] = deriv_L_w[j,:]+(softmax(w@x.
↪T)[j]-1)*x
                        else:
                            deriv_L_w[j,:] = deriv_L_w[j,:]+softmax(w@x.T)[j]*x

                w = w - (alpha*deriv_L_w)/len(X_batch)

            self.weights = w
            this_epoch_loss = self.loss(X,Y) #calculate loss with our new weights
            if np.abs(this_epoch_loss-last_epoch_loss) < self.conv_threshold:
                converge = True
        #print('number of epochs: ' + str(num_epoch))
        return num_epoch

    def loss(self, X, Y):
        '''
        Returns the total log loss on some dataset (X, Y), divided by the number␣
↪of examples.
        @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
        '''
        # calculate loss for each example (x,y), then average them
        total_loss = 0
        n = len(Y)
        # predictions = self.predict(X)

        for i in range(n):
            loss = 0
            probabilities = self.getSoftmaxProbability(X[i])

            for j in range(self.n_classes):
                if Y[i] == j:
                    # calculate our probability that x in j based on weights:
                    h_w_x = probabilities[j]
                    if h_w_x > 0: loss += np.log(h_w_x)
            total_loss += loss

        # return avg loss
        avg_loss = -total_loss/n
        return avg_loss


    def getSoftmaxProbability(self, x):
        '''
        Compute softmax regular probability vector of an example belonging to␣
↪each possible class.
        @params:
            x: a 1D Numpy array that is one example from the training set,␣
↪padded by 1 column for the bias
        @return:
            A 1D Numpy array with one element for each possible class j,
            where the element at j represents the probability of x belonging to␣
↪j.
        '''
        return softmax(np.dot(self.weights, x))

    def predict(self, X):
        '''
        Compute predictions based on the learned weigths 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.
        '''
        # use one vs all algorithm for returning class with highest probablity␣
↪of that x belonging to it
```

```
        # let j be each possible class

        predictions = []
        for x in X:
            normalized_prob = self.getSoftmaxProbability(x)
            # print("norm prob:", normalized_prob.shape)
            # the index of highest probablity is used as the predicted class
            x_prediction = np.argmax(normalized_prob)
            predictions.append(x_prediction)

        return np.array(predictions)

    def accuracy(self, X, Y):
        '''
        Outputs 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)
        '''

        # count how many 0s (accurately predicted examples) from the difference
        predict = self.predict(X)
        diff = predict - Y
        num_correct = np.sum(diff == 0)

        accuracy = num_correct/len(Y)

        return accuracy



### One vs. All Logistic Regression multi-class regression classifier ###

class OneVsAll:
    def __init__(
        self,
        n_features: int,
        n_classes: int,
        batch_size: int,
        conv_threshold: float,
        classifier = 'own',
        random_seed = SEED
    ) -> None:
        '''
        Initializes a OneVsAll multiclass classifer.
        @attrs:
            n_classes: the number of classes in the classification problem
            classifier: either 'own' or 'sk' to use either our own underlying␣
↪LogisticRegression or sklearn's SGDClassifier
            models: an array of either LogisticRegression or SGDClassifier␣
↪objects
        @inputs:
            n_features: the number of features in the classification problem + 1␣
↪for bias
            batch_size: size of every batch used in stochastic gradient descent␣
↪in binary LogisticRegression
            conv_threshold: the convergence threshold in stochastic gradient␣
↪descent in binary LogisticRegression
            SEED: an integer to set our random seed
        '''
        self.n_classes = n_classes
        self.classifier = classifier
        if classifier == 'own':
            self.models = [
                LogisticRegression(n_features, 2, batch_size, conv_threshold)
                for _ in range(n_classes)
            ]
        else:
            self.models = [
                SGDClassifier(
                    loss="log_loss",
                    fit_intercept = False,
                    penalty = None,
                    random_state=random_seed,
                    tol=1e-4,
                    eta0 = 0.03,
                    learning_rate = 'constant'
                )
                for _ in range(n_classes)
            ]

    def train(self, X: np.ndarray, Y: np.ndarray) -> None:
        # Train a binary classifier for each class against all others
        for class_label in range(self.n_classes):
            binary_labels = (Y == class_label).astype(int)
            self.models[class_label].fit(X, binary_labels)

    def predict(self, X: np.ndarray) -> np.ndarray:
        # Get the probabilities of each class for each data point
        # Returns a numpy array with a prediction for each datapoint
        predictions = np.zeros(len(X))
        if self.classifier == 'own':
            for j in range(0,len(X)): #for each datapoint
                probabilities = np.zeros(len(self.models))
                for i in range(0,len(self.models)):
                    #for each model i, it trains on if we relabel current class␣
↪i as '1' and all other classes as '0'
                    #so, the value from getSoftmaxProbability at index 1 would␣
↪be probability of being in class 1 for that model, or our current class i
                    prob_x_class_i = self.models[i].getSoftmaxProbability(X[j,:
↪])[1]
                    probabilities[i] = prob_x_class_i
                predictions[j] = np.argmax(probabilities) #for all classes get␣
↪the class with greatest probability
        else:
            for j in range(0,len(X)): #for each datapoint
                probabilities = np.zeros(len(self.models))
                for i in range(0,len(self.models)):
                    #same as our own function except we construct our weights␣
↪for both classes instead of just one class like from sklearn's coef_
                    weights = np.zeros([2,len(self.models[i].coef_[0])])
                    weights[0,:] = -self.models[i].coef_[0]
                    weights[1,:] = self.models[i].coef_[0]
                    prob_x_class_i = softmax(np.dot(weights,(X[j,:])))[1]
                    probabilities[i] = prob_x_class_i
                predictions[j] = np.argmax(probabilities) #for all classes get␣
↪the class with greatest probability

        return predictions

    def accuracy(self, X: np.ndarray, Y: np.ndarray) -> float:
        #Returns percentage of datapoints where prediction matches actual label
        predictions = self.predict(X)
        return np.mean(predictions == Y)



### All Pairs Logistic Regression multi-class regression classifier ###

class AllPairs:
    def __init__(
        self,
        n_features: int,
        n_classes: int,
        batch_size: int,
        conv_threshold: float,
        classifier: str = "own",
        random_seed: int = SEED
    ):
        '''
        Initializes a OneVsAll multiclass classifer.
        @attrs:
            n_classes: the number of classes in the classification problem
            pairs: all different possible pairs of classes
            models: an array of either LogisticRegression or SGDClassifier␣
↪objects
        @inputs:
            n_features: the number of features in the classification problem + 1␣
↪for bias
            batch_size: size of every batch used in stochastic gradient descent␣
↪in binary LogisticRegression
            conv_threshold: the convergence threshold in stochastic gradient␣
↪descent in binary LogisticRegression
            classifier: either 'own' or 'sk' to use either our own underlying␣
↪LogisticRegression or sklearn's SGDClassifier
            SEED: an integer to set our random seed
        '''
        self.n_classes = n_classes
        self.pairs = list(it.combinations(range(n_classes), 2))
        print(self.pairs)
        if classifier == "own":
            self.models = {
                (i, j): LogisticRegression(
                    n_features,
                    2,
                    batch_size,
                    conv_threshold
                )
                for i, j in self.pairs
            }
        else:
            self.models = {
                (i, j): SGDClassifier(
                    loss="log_loss",
                    fit_intercept = False,
                    penalty = None,
                    random_state=random_seed,
                    tol=1e-4,
                    eta0 = 0.03,
                    learning_rate = 'constant'
                )
                for i, j in self.pairs
            }

    def train(self, X: np.ndarray, Y: np.ndarray) -> None:
        # Iterate over all pair combinations and train the underlying binary␣
↪classification model
        print("self pairs:", self.pairs)
        for i, j in self.pairs:
            # print("ij", i,j)
            indices = (Y == i) | (Y == j)

            # Get appropriate data
            X_pair, Y_pair = X[indices], Y[indices]
            Y_pair = (Y_pair == j).astype(int)

            # Train on the pair
            self.models[(i, j)].fit(X_pair, Y_pair)

    def predict(self, X: np.ndarray) -> np.ndarray:
        # Count predictions for each class
        votes = np.zeros((len(X), self.n_classes))
        for (i, j), model in self.models.items():
            predictions = np.array(model.predict(X))
            # print("predictions shape:", predictions.shape, predictions)
            votes[:, j] += predictions
            votes[:, i] += (1 - predictions)

        # Return the class with the most predictions
        return np.argmax(votes, axis=1)

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

4 Check Model

Methods used to check model are from sklearn (references [1],[2],[3],[4] from overview).

4.1 LogisticRegression Unit Tests

[26]:
```
### Unit Tests LogisticRegression (for 2 classes) ###

np.random.seed(SEED)

#softmax function, calculated results by hand
softmax_test_1 = np.array([1,2,3,4,5]) #generic example
assert softmax(softmax_test_1) == pytest.approx([0.01166,0.03168,0.08612,0.
↪23412,0.6364086], .001)
```

```
softmax_test_2 = np.array([-2,0,3,-2,3]) #tests if can handle multiple of the␣
↪same values/max value occurs multiple times
assert softmax(softmax_test_2) == pytest.approx([0.003266,0.024131,0.484669,0.
↪003266,0.484669], .001)


#weights during fit
#training points can be split by a halfspace,
X_weights_1 = np.
↪array([[-2,0,1],[-1,0,1],[-1,1,1],[0,2,1],[0,1,1],[0,-1,1],[0,-2,1],[1,0,1],[1,-1,1],[2,0,1]]
↪#includes bias term
y_weights_1 = np.array([0,0,0,0,0,1,1,1,1,1])

#the training points cannot be split by a halfspace, at least 2 training points␣
↪will have to be mislabeled
X_weights_2 = np.
↪array([[-2,0,1],[-1,0,1],[-1,1,1],[0,2,1],[0,1,1],[0,-1,1],[0,-2,1],[1,0,1],[1,-1,1],[2,0,1]]
↪#includes bias term
y_weights_2 = np.array([0,0,1,0,0,1,1,1,0,1])

own_log_reg = LogisticRegression(3,2,1,1e-4)
#penalty = None means no regularization, learning rate = eta0 is our alpha in␣
↪LogisticRegression, tolerance = tol set to our convergence threshold,␣
↪n_iter_no_change = 1 means if we hit convergence once to stop
sk_sgd_log_reg = SGDClassifier(loss = 'log_loss', fit_intercept = False, penalty␣
↪= None,learning_rate = 'constant',eta0=0.03,tol=1e-4,n_iter_no_change=1)

#first check shape of weights
assert (own_log_reg.weights.shape == np.array([2,3])).all()

#check for each of these datasets, if the normalized weights for each model are␣
↪within 0.01 of each other for each feature
own_log_reg.fit(X_weights_1,y_weights_1)
sk_sgd_log_reg.fit(X_weights_1,y_weights_1)
assert (abs(normalize(own_log_reg.weights,axis=1)[1] - normalize(sk_sgd_log_reg.
↪coef_)[0]) <= 0.01).all()

own_log_reg.fit(X_weights_2,y_weights_2)
sk_sgd_log_reg.fit(X_weights_2,y_weights_2)
assert (abs(normalize(own_log_reg.weights,axis=1)[1] - normalize(sk_sgd_log_reg.
↪coef_)[0]) <= 0.01).all()



#### loss function ###
log_reg = LogisticRegression(3,2,1,1e-4)
```

```
log_reg.weights = np.array([[-0.5,2,0.1],[0.5,-2,-0.1]]) #set weights to some␣
 ↪random 2 class, 3 features (including bias) array
#test on dataset with one datapoint
#case with vector of all 0's, should be same loss for both classes
X_loss_1 = np.array([[0,0,0]])
assert log_reg.loss(X_loss_1,np.array([0])) == pytest.approx(np.array([0.
 ↪6931472]),0.001) #-ln(0.5)
assert log_reg.loss(X_loss_1,np.array([1])) == pytest.approx(np.array([0.
 ↪6931472]),0.001)

#with multiple datapoints, random test
X_loss_2 = np.array([[-1,2,0.3],[1.5,-0.1,2]])
y_loss_2 = np.array([0,1])
assert log_reg.loss(X_loss_2,y_loss_2) == pytest.approx(0.10076465,0.001)



### getSoftmaxProbability function ###
X_gsp_1 = np.array([[-0.5,5,1],[1,1,1],[10,2,1],[-10,-2,1],[2,1,1]]) #random␣
 ↪test data with bias included
y_gsp_1 = np.array([0,1,1,0,1])

log_reg = LogisticRegression(3,2,1,1e-4)
log_reg.fit(X_gsp_1,y_gsp_1)
test_gsp_1 = np.array([0,0,0])

#no matter our weights, if datapoint is vector of 0's, getSoftmaxProbability␣
 ↪should give us around an equal probability for each class
assert log_reg.getSoftmaxProbability(test_gsp_1) == pytest.approx(np.array([0.
 ↪5,0.5]),0.001)
#set weights to some random 2 class, 3 features (including bias) array, test on␣
 ↪random datapoint
log_reg.weights = np.array([[-0.5,2,0.1],[0.5,-2,-0.1]])
test_gsp_2 = np.array([3,-0.9,1.3])
assert log_reg.getSoftmaxProbability(test_gsp_2) == pytest.approx(np.array([0.
 ↪0017612,0.9982388]),0.001) #calculated by hand



### predict function ###
#the training points can be split by a halfspace
X_1 = np.
 ↪array([[-2,0,1],[-1,0,1],[-1,1,1],[0,2,1],[0,1,1],[0,-1,1],[0,-2,1],[1,0,1],[1,-1,1],[2,0,1]]
 ↪#includes bias term
y_1 = np.array([0,0,0,0,0,1,1,1,1,1])
```

```
X_test_1 = np.array([[0,5,1],[-10,1,1],[-2,2,1],[2,-2,1],[1,-10,1],[5,0,1]])␣
 ↪#includes terms that would be right on the edge of a correct halfspace

#the training points cannot be split by a halfspace, at least 2 training points␣
 ↪will have to be mislabeled
X_2 = np.
 ↪array([[-2,0,1],[-1,0,1],[-1,1,1],[0,2,1],[0,1,1],[0,-1,1],[0,-2,1],[1,0,1],[1,-1,1],[2,0,1]]
 ↪#includes bias term
y_2 = np.array([0,0,1,0,0,1,1,1,0,1])
X_test_2 = np.array([[0,5,1],[-10,1,1],[-2,2,1],[2,-2,1],[1,-10,1],[5,0,1]])␣
 ↪#includes terms that would be right on the edge of a correct halfspace

#uneven number of training points for each label
X_3 = np.
 ↪array([[-2,0,1],[-1,1,1],[0,2,1],[0,-1,1],[0,-2,1],[1,0,1],[1,-1,1],[2,0,1]])␣
 ↪#includes bias term
y_3 = np.array([0,0,0,1,1,1,1,1])
X_test_3 = np.array([[0,5,1],[-10,1,1],[-2,2,1],[2,-2,1],[1,-10,1],[5,0,1]])␣
 ↪#includes terms that would be right on the edge of a correct halfspace

#initialize LogisticRegression model with n_classes = 2, batch_size = 1,␣
 ↪conv_threshold = 1e-4
log_reg = LogisticRegression(3,2,1,1e-4)

#train our LogisticRegression model on each training dataset, then test␣
 ↪predictions
log_reg.fit(X_1,y_1)
assert (log_reg.predict(X_test_1) == np.array([0,0,0,1,1,1])).all()

log_reg.fit(X_2,y_2)
assert (log_reg.predict(X_test_2) == np.array([0,0,0,1,1,1])).all()

log_reg.fit(X_3,y_3)
assert (log_reg.predict(X_test_3) == np.array([0,0,0,1,1,1])).all()



### accuracy function ###
#test accuracy based on previous prediction tests, we know prediction should be␣
 ↪[0,0,0,1,1,1]
accuracy_test_1 = np.array([1,1,1,0,0,0]) #make sure labeling is correct (flip␣
 ↪labels, get 0 accuracy)
accuracy_test_2 = np.array([1,0,0,1,0,1]) #make sure percentage for accuracy is␣
 ↪correct to the nearest 0.001 for long decimals
log_reg.fit(X_1,y_1)
assert log_reg.accuracy(X_test_1, accuracy_test_1) == 0
assert log_reg.accuracy(X_test_1, accuracy_test_2) == pytest.approx(0.667,0.001)

print('Passed all tests!')
```

```
Passed all tests!
```

4.2 Logistic Regression Unit Tests from HW3 (just to double check)

[27]:
```
# Creates Test Model with 2 predictors, 2 classes, a Batch Size of 5 and a␣
↪Threshold of 1e-2 (used 3 for predictors to include bias)
test_model1 = LogisticRegression(3, 2, 5, 1e-2)

# Creates Test Data
x_bias = np.array([[0,4,1], [0,3,1], [5,0,1], [4,1,1], [0,5,1]])
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]])
y_test = np.array([0,0,1,0,1])

# Creates Test Model with 2 predictors, 1 classes, a Batch Size of 1 and a␣
↪Threshold of 1e-2
test_model2 = LogisticRegression(3, 3, 1, 1e-2)

# Creates 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])


# Test Model Loss
assert test_model1.loss(x_bias, y) == pytest.approx(0.693, .001) # Checks if␣
↪answer is within .001
assert test_model2.loss(x_bias2, y2) == pytest.approx(1.099, .001) # Checks if␣
↪answer is within .001

# Test Train Model and Checks Model Weights
assert test_model1.fit(x_bias, y) == 14
assert test_model1.weights == pytest.approx(np.array([[-0.218, 0.231, 0.0174], [␣
↪0.218, -0.231, -0.0174]]), 0.01) # Answer within .01

assert test_model2.fit(x_bias, y) == 9
assert test_model2.weights == pytest.approx(np.array([[-0.300,  0.560,  0.093],␣
↪[ 0.523, -0.257,  0.032], [-0.226, -0.304, -0.123]]), .05)

# Test Model Predict
assert (test_model1.predict(x_bias_test) == np.array([0., 0., 1., 1., 1.])).all()
assert (test_model2.predict(x_bias_test2) == np.array([0, 0, 1, 1])).all()
```

```
# Test Model Accuracy
assert test_model1.accuracy(x_bias_test, y_test) == .8
assert test_model2.accuracy(x_bias_test2, y_test2) == .25

print("Passed all tests!")
```

```
Passed all tests!
```

4.3 All Pairs Unit Tests

[28]:
```
# Run SKLearn's one verse all logistic regression using the example X_1 and X_2␣
 ↪datasets.
# Get the weight matrix after training for each of the sub binary classifiers to
# compare against the weights of our own implementation of one verse all
# logistic regression. Also compare predictions and accuracy.

# Set the random seed
np.random.seed(42)

# Smaller test data
X_1 = np.random.rand(6, 3)
y_1 = np.array([0, 1, 2])
X_test_1 = np.random.rand(3, 3)

# Larger test data
X_2 = np.random.rand(8, 4)
y_2 = np.array([0, 1, 2, 0, 1, 2, 0, 2])
X_test_2 = np.random.rand(4, 4)

# Train sklearn all pairs logistic regression
sk_model = OneVsOneClassifier(SGDClassifier(
    loss="log_loss",
    fit_intercept = False,
    penalty = None,
    random_state=SEED,
    tol=1e-4,
    eta0 = 0.03,
    learning_rate = 'constant'
))
sk_model.fit(X_2, y_2)

# Print weights of each classes binary classification model
print("SKLearn multiclass + SKLearn SGDClassifier weights:")
print(sk_model.estimators_)
for i, model in enumerate(sk_model.estimators_):
    print(f"Class {i} weights {model.coef_}")
    print()
```

```python
# Train our own all pairs logistic regression
all_pairs1 = AllPairs(4, 3, 1, 1e-4, "sk")
all_pairs1.train(X_2, y_2)

# Print weights of each classes binary classification model
print("Local multiclass + SKLearn SGDClassifier weights:")
for idx, (i, model) in enumerate(all_pairs1.models.items()):
    print(f"Class {i} weights {model.coef_}")
    assert np.all(sk_model.estimators_[idx].coef_ == model.coef_)
    print()

print("Passed all tests!")

all_pairs1 = AllPairs(4, 3, 100000000, 1e-4, "own")
all_pairs1.train(X_2, y_2)

# Print weights of each classes binary classification model
print("Local multiclass + SKLearn SGDClassifier weights:")
for idx, (i, model) in enumerate(all_pairs1.models.items()):
    print(f"Class {i} weights {model.weights}")
    print()

print("Passed all tests!")
```

```
/opt/anaconda3/envs/data2060/lib/python3.12/site-
packages/sklearn/linear_model/_stochastic_gradient.py:744: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(
/opt/anaconda3/envs/data2060/lib/python3.12/site-
packages/sklearn/linear_model/_stochastic_gradient.py:744: ConvergenceWarning:
Maximum number of iteration reached before convergence. Consider increasing
max_iter to improve the fit.
  warnings.warn(

SKLearn multiclass + SKLearn SGDClassifier weights:
(SGDClassifier(eta0=0.03, fit_intercept=False, learning_rate='constant',
              loss='log_loss', penalty=None, random_state=42, tol=0.0001),
SGDClassifier(eta0=0.03, fit_intercept=False, learning_rate='constant',
              loss='log_loss', penalty=None, random_state=42, tol=0.0001),
SGDClassifier(eta0=0.03, fit_intercept=False, learning_rate='constant',
              loss='log_loss', penalty=None, random_state=42, tol=0.0001))
Class 0 weights [[ 0.08194396 -5.02884238  1.47399865  1.61517363]]

Class 1 weights [[ 2.86128199 -5.16187421 -3.94720969  2.8980914 ]]
Class 2 weights [[ 2.64198104  1.53082514 -6.63277401  1.54183575]]

[(0, 1), (0, 2), (1, 2)]
self pairs: [(0, 1), (0, 2), (1, 2)]
Local multiclass + SKLearn SGDClassifier weights:
Class (0, 1) weights [[ 0.08194396 -5.02884238  1.47399865  1.61517363]]

Class (0, 2) weights [[ 2.86128199 -5.16187421 -3.94720969  2.8980914 ]]

Class (1, 2) weights [[ 2.64198104  1.53082514 -6.63277401  1.54183575]]

Passed all tests!
[(0, 1), (0, 2), (1, 2)]
self pairs: [(0, 1), (0, 2), (1, 2)]
Local multiclass + SKLearn SGDClassifier weights:
Class (0, 1) weights [[ 0.16124371  1.64817051 -0.63521881 -0.47042871]
 [-0.16125316 -1.64817779  0.63520822  0.47041932]]

Class (0, 2) weights [[-0.90404055  1.27223664  1.31040133 -0.77521409]
 [ 0.90402606 -1.27224618 -1.31040868  0.77520357]]

Class (1, 2) weights [[-0.99750845 -0.64934053  2.37249553 -0.5370203 ]
 [ 0.99749159  0.64933387 -2.37250648  0.53700372]]

Passed all tests!
```

4.3.1 One Vs All Unit Tests

[29]:
```
# Run SKLearn's one verse all logistic regression using the example X_1 and X_2␣
 ↪datasets.
# Get the weight matrix after training for each of the sub binary classifiers to
# compare against the weights of our own implementation of one verse all
# logistic regression. Also compare predictions and accuracy.

# Smaller test data
X_1 = np.random.rand(6, 3)
y_1 = np.array([0, 1, 2])
X_test_1 = np.random.rand(3, 3)

# Larger test data (4 features, 3 classes)
X_2 = np.random.rand(8, 4)
y_2 = np.array([0, 1, 2, 0, 1, 2, 0, 2])
X_test_2 = np.random.rand(4, 4)

# Largest test data (6 features, 6 classes)
X_3 = np.random.rand(20, 6)
y_3 = np.random.randint(0, 6, size=20)
```

```python
X_test_3 = np.random.rand(17, 6)

# Train sklearn one v all
sk_model = OneVsRestClassifier(SGDClassifier(
    loss="log_loss",
    fit_intercept = False,
    penalty = None,
    random_state=SEED,
    tol=1e-4,
    eta0 = 0.03,
    learning_rate = 'constant'
))
sk_model.fit(X_2, y_2)
sk_score = sk_model.score(X_2, y_2)
sk_predicts = sk_model.predict(X_test_2)

sk_weights = []
for i, model in enumerate(sk_model.estimators_):
    sk_weights.append(model.coef_)

# Train our own one v all
one_all = OneVsAll(4, 3, 1, 1e-4, "sk")
one_all.train(X_2, y_2)
our_score = one_all.accuracy(X_2, y_2)
our_predicts = one_all.predict(X_test_2)

# Print weights of each classes binary classification model
our_weights = []
for model in one_all.models:
    our_weights.append(model.coef_)


sk_weights = np.array(sk_weights)
our_weights = np.array(our_weights)

## Assertions
assert our_weights.shape == (3,1,4), "Check Model Weights Shape"
assert our_weights.shape == sk_weights.shape, "Compare Model Weights Shape"
assert (sk_weights == our_weights).all(), "Compare Model Weights"
assert our_score == pytest.approx(sk_score, .001), "Compare Accuracy"
assert (sk_predicts == our_predicts).all(), "Compare Predictions"

# NEXT SET
# Train sklearn one v all
sk_model_3 = OneVsRestClassifier(SGDClassifier(
    loss="log_loss",
    fit_intercept = False,
    penalty = None,
    random_state=SEED,
    tol=1e-4,
    eta0 = 0.03,
    learning_rate = 'constant'
))
sk_model_3.fit(X_3, y_3)
sk_score = sk_model_3.score(X_3, y_3)
sk_predicts = sk_model_3.predict(X_test_3)

sk_weights_3 = []
for i, model in enumerate(sk_model_3.estimators_):
    sk_weights_3.append(model.coef_)

# Train our own one v all
one_all = OneVsAll(6, 6, 3, 1e-4, "sk")
one_all.train(X_3, y_3)
our_score = one_all.accuracy(X_3, y_3)
our_predicts = one_all.predict(X_test_3)

# Print weights of each classes binary classification model
our_weights_3 = []
for model in one_all.models:
    our_weights_3.append(model.coef_)


sk_weights_3 = np.array(sk_weights_3)
our_weights_3 = np.array(our_weights_3)

## Assertions
assert our_weights_3.shape == (6,1,6), "Check Model 3 Weights Shape"
assert our_weights_3.shape == sk_weights_3.shape, "Compare Model 3 Weights Shape"
assert (sk_weights_3 == our_weights_3).all(), "Compare Model 3 Weights"
assert our_score == pytest.approx(sk_score, .001), "Compare Accuracy 3"
assert (sk_predicts == our_predicts).all(), "Compare Predictions 3"

print("Passed all tests!")
```

```
Passed all tests!
```

4.4 Testing to match SKLearn

We tried extensive tuning of SKLearn's SGDCLassifier including setting the a constant learning rate, fixing the random state, removing regularization, and fixing the convergence tolerance to match our binary logistic regression from HW3. However, we were never able to learn the same weights.

4.5 German Numerical Credit Data

Simple binary classification test.

```
[30]: def get_credit():
          """
          Gets and preprocesses German Credit data
          """
          #data = pd.read_csv('./data/german_numerical-binsensitive.csv') # Reads file␣
       ↪- may change
          data = pd.read_csv('../data/german_numerical-binsensitive.csv')
          # MONTH categorizing
          data['month'] = pd.cut(data['month'],3, labels=['month_1', 'month_2',␣
       ↪'month_3'], retbins=True)[0]
          # month bins: [ 3.932      , 26.66666667, 49.33333333, 72.        ]
          a = pd.get_dummies(data['month'])
          data = pd.concat([data, a], axis = 1)
          data = data.drop(['month'], axis=1)

          # CREDIT categorizing
          data['credit_amount'] = pd.cut(data['credit_amount'], 3,␣
       ↪labels=['cred_amt_1', 'cred_amt_2', 'cred_amt_3'], retbins=True)[0]
          # credit bins: [  231.826,  6308.   , 12366.   , 18424.   ]
          a = pd.get_dummies(data['credit_amount'])
          data = pd.concat([data, a], axis = 1)
          data = data.drop(['credit_amount'], axis=1)

          for header in ['investment_as_income_percentage', 'residence_since',␣
       ↪'number_of_credits']:
              a = pd.get_dummies(data[header], prefix=header)
              data = pd.concat([data, a], axis = 1)
              data = data.drop([header], axis=1)

          # change from 1-2 classes to 0-1 classes
          data['people_liable_for'] = data['people_liable_for'] -1
          data['credit'] = -1*(data['credit']) + 2 # original encoding 1: good, 2: bad.
       ↪ we switch to 1: good, 0: bad

          # balance dataset
          data = data.reindex(np.random.permutation(data.index)) # shuffle
          pos = data.loc[data['credit'] == 1]
          neg = data.loc[data['credit'] == 0][:350]
          combined = pd.concat([pos, neg])

          y = data.iloc[:, data.columns == 'credit'].to_numpy()
          x = data.drop(['credit', 'sex', 'age', 'sex-age'], axis=1).to_numpy()
```

```
    # split into train and validation
    X_train, X_val, y_train, y_val = x[:350, :], x[351:526, :], y[:350, :].
↪reshape([350,]), y[351:526, :].reshape([175,])

    # keep info about sex and age of validation rows for fairness portion
    x_sex = data.iloc[:, data.columns == 'sex'].to_numpy()[351:526].
↪reshape([175,])
    x_age = data.iloc[:, data.columns == 'age'].to_numpy()[351:526].
↪reshape([175,])
    x_sex_age = data.iloc[:, data.columns == 'sex-age'].to_numpy()[351:526].
↪reshape([175,])

    return X_train, X_val, y_train, y_val, x_sex, x_age, x_sex_age

np.random.seed(42)
X_train, X_test, y_train, y_test, x_sex, x_age, x_sex_age = get_credit()
n_samples = len(X_train)
n_features = len(X_train[0])+1
n_classes = 2

# process data
X_train[X_train == False] = 0.0
X_train[X_train == True] = 1.0
X_test[X_test == False] = 0.0
X_test[X_test == True] = 1.0

train_bias = np.ones((n_samples, 1))
X_train = np.hstack((X_train, train_bias))

test_bias = np.ones((len(X_test), 1))
X_test = np.hstack((X_test, test_bias))

print("x shape:", X_train.shape, X_test.shape)
print("num features:", n_features)

batch_size = 1
conv_threshold = 1e-4 #default tolerance for SKLogisticRegression from sklearn,␣
↪they use different kind of SGD,

# One-vs-All
ova = OneVsAll(n_features, n_classes, batch_size=batch_size,␣
↪conv_threshold=conv_threshold)
ova.train(X_train, y_train)
print(f"One-vs-All Accuracy (own ova own classifier): {ova.accuracy(X_test,␣
↪y_test)}")
```

```
ova = OneVsAll(n_features, n_classes, batch_size=batch_size,␣
 ↪conv_threshold=conv_threshold, classifier='sk')
ova.train(X_train, y_train)
print(f"One-vs-All Accuracy (own ova sk classifier): {ova.accuracy(X_test,␣
 ↪y_test)}")

# All-Pairs
all_pairs = AllPairs(n_features, n_classes, batch_size=batch_size,␣
 ↪conv_threshold=conv_threshold)
all_pairs.train(X_train, y_train)
print(f"All-Pairs Accuracy (own all pairs own classifier): {all_pairs.
 ↪accuracy(X_test, y_test)}")

all_pairs = AllPairs(n_features, n_classes, batch_size=batch_size,␣
 ↪conv_threshold=conv_threshold, classifier='sk')
all_pairs.train(X_train, y_train)
print(f"All-Pairs Accuracy (own all pairs sk classifier): {all_pairs.
 ↪accuracy(X_test, y_test)}")
```

```
x shape: (350, 70) (175, 70)
num features: 70
One-vs-All Accuracy (own ova own classifier): 0.7257142857142858
One-vs-All Accuracy (own ova sk classifier): 0.7542857142857143
[(0, 1)]
self pairs: [(0, 1)]
All-Pairs Accuracy (own all pairs own classifier): 0.7257142857142858
[(0, 1)]
self pairs: [(0, 1)]
All-Pairs Accuracy (own all pairs sk classifier): 0.7542857142857143
```

4.6 SkLearn Iris Data

Multiclass classification test.

```
[31]: data = load_iris()
X = data.data
Y = data.target

n_samples = len(X)
n_features = len(X[0])+1
n_classes = 3
batch_size = 1
conv_threshold = 1e-4 #this is default tolerance 'tol' parameter for␣
 ↪SKLogisticRegression

print("Number samples: ", n_samples, "Number Feats: ", n_features)
```

```
# shuffle data
p = np.random.permutation(n_samples)
X_ = X[p]
Y_ = Y[p]

# add bias to shuffled data
bias = np.ones((n_samples,1))
X_ = np.hstack((X_, bias))

# split data 80/20
train_size = int(.8*n_samples)
X_train = X_[:train_size,:]
y_train = Y_[:train_size]

X_test = X_[train_size:, :]
y_test = Y_[train_size:]

np.random.seed(SEED)
# # One-vs-All
ova = OneVsAll(n_features, n_classes, batch_size=batch_size,␣
 ↪conv_threshold=conv_threshold)
ova.train(X_train, y_train)
print(f"One-vs-All Accuracy (own ova own classifier): {ova.accuracy(X_test,␣
 ↪y_test)}")

ova = OneVsAll(n_features, n_classes, batch_size=batch_size,␣
 ↪conv_threshold=conv_threshold, classifier='sk')
ova.train(X_train, y_train)
print(f"One-vs-All Accuracy (own ova sk classifier): {ova.accuracy(X_test,␣
 ↪y_test)}")

ova = OneVsRestClassifier(SGDClassifier(
    loss="log_loss",
    fit_intercept = False,
    penalty = None,
    random_state=SEED,
    tol=1e-4,
    eta0 = 0.03,
    learning_rate = 'constant'
))
ova.fit(X_train, y_train)
print(f"One-vs-All Accuracy (sk ova and sk classifier): {ova.score(X_test,␣
 ↪y_test)}")

# All-Pairs
all_pairs = AllPairs(n_features, n_classes, batch_size=batch_size,␣
 ↪conv_threshold=conv_threshold)
```

```
all_pairs.train(X_train, y_train)
print(f"All-Pairs Accuracy (own all pairs own classifier): {all_pairs.
↪accuracy(X_test, y_test)}")

all_pairs = AllPairs(n_features, n_classes, batch_size=batch_size,␣
↪conv_threshold=conv_threshold, classifier='sk')
all_pairs.train(X_train, y_train)
print(f"All-Pairs Accuracy (own all pairs sk classifier): {all_pairs.
↪accuracy(X_test, y_test)}")

all_pairs = OneVsOneClassifier(SGDClassifier(
    loss="log_loss",
    fit_intercept = False,
    penalty = None,
    random_state=SEED,
    tol=1e-4,
    alpha = 0.03
))
all_pairs.fit(X_train, y_train)
print(f"All-Pairs Accuracy (sk all pairs and sk classifier): {all_pairs.
↪score(X_test, y_test)}")
```

```
Number samples:  150 Number Feats:  5
One-vs-All Accuracy (own ova own classifier): 0.9
One-vs-All Accuracy (own ova sk classifier): 0.9333333333333333
One-vs-All Accuracy (sk ova and sk classifier): 0.9333333333333333
[(0, 1), (0, 2), (1, 2)]
self pairs: [(0, 1), (0, 2), (1, 2)]
All-Pairs Accuracy (own all pairs own classifier): 0.9333333333333333
[(0, 1), (0, 2), (1, 2)]
self pairs: [(0, 1), (0, 2), (1, 2)]
All-Pairs Accuracy (own all pairs sk classifier): 0.9333333333333333
All-Pairs Accuracy (sk all pairs and sk classifier): 0.9333333333333333
```

4.7 Penguins Dataset

Multiclass classification test

[32]:
```
penguin = pd.read_csv('../data/penguins.csv')

# Handling missing values
imputer = SimpleImputer(strategy='most_frequent')
penguin.iloc[:, :] = imputer.fit_transform(penguin)

# Convert sex to 0(male) and 1(female)
penguin['sex'] = penguin['sex'].map({'MALE': 0, 'FEMALE': 1})

# Convert categorical to numbers
```

```
encoder = LabelEncoder()
penguin['island'] = encoder.fit_transform(penguin['island'])
penguin['species'] = encoder.fit_transform(penguin['species'])

# Separate X and Y
X = penguin.drop('species', axis=1).to_numpy()
Y = penguin['species'].to_numpy()

# Set hyperparmaeters
n_samples = len(X)
n_features = X.shape[1] + 1
n_classes = 3
batch_size = 1
conv_threshold = 1e-4 #this is default tolerance 'tol' parameter for␣
 ↪SKLogisticRegression

print("Number samples: ", n_samples, "Number Features: ", n_features)

# shuffle data
p = np.random.permutation(n_samples)
X_shuf = X[p]
Y_shuf = Y[p]

# add bias to shuffled data
bias = np.ones((n_samples,1))
X_shuf = np.hstack((X_shuf, bias))

# split data 80/20
train_size = int(.8*n_samples)
X_train = X_shuf[:train_size,:]
y_train = Y_shuf[:train_size]

X_test = X_shuf[train_size:, :]
y_test = Y_shuf[train_size:]
```

```
Number samples:  344 Number Features:  7
```

[33]:
```
np.random.seed(SEED)
ova = OneVsAll(n_features, n_classes, batch_size=batch_size,␣
 ↪conv_threshold=conv_threshold)
ova.train(X_train, y_train)
print(f"One-vs-All Accuracy (own ova own logreg): {ova.accuracy(X_test,␣
 ↪y_test)}")

ova = OneVsAll(n_features, n_classes, batch_size=batch_size,␣
 ↪conv_threshold=conv_threshold, classifier='sk')
ova.train(X_train, y_train)
```

```
print(f"One-vs-All Accuracy (own ova sk logreg): {ova.accuracy(X_test, y_test)}")

ova = OneVsRestClassifier(SGDClassifier(
    loss="log_loss",
    fit_intercept = False,
    penalty = None,
    random_state=SEED,
    tol=1e-4,
    eta0 = 0.03,
    learning_rate = 'constant'
))
ova.fit(X_train, y_train)
print(f"One-vs-All Accuracy (sk ova and sk classifier): {ova.score(X_test,␣
 ↪y_test)}")

# All-Pairs
all_pairs = AllPairs(n_features, n_classes, batch_size=batch_size,␣
 ↪conv_threshold=conv_threshold)
all_pairs.train(X_train, y_train)
print(f"All-Pairs Accuracy (own all pairs own classifier): {all_pairs.
 ↪accuracy(X_test, y_test)}")

all_pairs = AllPairs(n_features, n_classes, batch_size=batch_size,␣
 ↪conv_threshold=conv_threshold, classifier='sk')
all_pairs.train(X_train, y_train)
print(f"All-Pairs Accuracy (own all pairs sk classifier): {all_pairs.
 ↪accuracy(X_test, y_test)}")

all_pairs = OneVsOneClassifier(SGDClassifier(
    loss="log_loss",
    fit_intercept = False,
    penalty = None,
    random_state=SEED,
    tol=1e-4,
    eta0 = 0.03,
    learning_rate = 'constant'
))
all_pairs.fit(X_train, y_train)
print(f"All-Pairs Accuracy (sk all pairs and sk classifier): {all_pairs.
 ↪score(X_test, y_test)}")
```

```
One-vs-All Accuracy (own ova own logreg): 0.36231884057971014
One-vs-All Accuracy (own ova sk logreg): 0.36231884057971014
One-vs-All Accuracy (sk ova and sk classifier): 0.36231884057971014
[(0, 1), (0, 2), (1, 2)]
self pairs: [(0, 1), (0, 2), (1, 2)]
All-Pairs Accuracy (own all pairs own classifier): 0.36231884057971014
```

```
[(0, 1), (0, 2), (1, 2)]
self pairs: [(0, 1), (0, 2), (1, 2)]
All-Pairs Accuracy (own all pairs sk classifier): 0.4927536231884058
All-Pairs Accuracy (sk all pairs and sk classifier): 0.14492753623188406
```

```
[ ]:
```

References

[1] scikit-learn (n.d.) *The Iris Dataset* [Online]. Available at: `https://scikit-learn.org/1.5/auto_examples/datasets/plot_iris_dataset.html` (Accessed: 22 November 2024).

[2] scikit-learn (n.d.) *LogisticRegression* [Online]. Available at: `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html` (Accessed: 5 December 2024).

[3] scikit-learn (n.d.) *Multiclass and multioutput algorithms* [Online]. Available at: `https://scikit-learn.org/1.5/modules/multiclass.html#multiclass-classification` (Accessed: 5 December 2024).

[4] scikit-learn (n.d.) *SGDClassifier* [Online]. Available at: `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html` (Accessed: 13 December 2024).

[5] Shalev-Shwartz, S. and Ben-David, S. (2014) *Understanding machine learning: From theory to algorithms.* Cambridge: Cambridge University Press.

[6] Zsom, A. (2024) *DATA2060-slides* [Online]. Available at: `https://drive.google.com/drive/u/2/folders/1U-B6Z2rv78uoQMvduCTQxQPokohFA_2e` (Accessed: 27 October 2024)