Boosting With Neural Networks

Link to github: https://github.com/dfiume1/2060_Close_AI.git

Markdown Section

Overview of the Boosting Algorithm

Boosting is an ensemble learning method. The idea of boosting algorithm is to create a single strong learner by combining the predictions of weak learners. This method was proposed by Freund and Schapire in the paper "A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting" in 1995 [1] and mainly applied to classification tasks. In boosting, each weak learner will adjust its weight according to the results of the previous round of weak learners in continuous iterations. Boosting iteratively builds weak classifiers or regressors and this new learner will pay special attention to the samples that were misclassified or having bigger errors in the previous round, so as to better handle these samples that did not perform well in the subsequent iterations. Thus, models can gradually optimize the performance. In the final prediction stage, the prediction results of these weak learners will be weighted and summed according to certain weights to obtain the final prediction result. Based on boosting, Freund and Schapire also proposed AdaBoosting, the adaptive boosting algorithm, in the above paper. At the same time, Drucker further applied it to regression problems in 1997. [2] Adaboosting improves the adaptive mechanism: Adaboosting dynamically adjusts the sample weights and the weights of weak learners according to the error during the iteration process. This adaptive mechanism enables AdaBoosting to gradually optimize the model performance and improve the prediction accuracy.

Advantages

- According to Drucker's 1997 paper "Improving Regressors using Boosting Techniques", AdaBoosting can achieve higher prediction performance by iteratively training multiple weak learners and reasonably combining their prediction results according to the weights. [2]
- The AdaBoosting algorithm is relatively simple and easy to understand and implement.

Disadvantages

- Since AdaBoosting gives higher weights to wrongly predicted samples, noise and outliers may cause the model to perform poorly. .

- AdaBoosting needs to train multiple weak learners during the training process, the amount of training calculations may be heavy.
- If the weak learner is too complex or the number of iterations is too many, AdaBoosting may have overfitting problems.

Modification

In this project, we used Adaboosting to implement regression instead of classification problems. The main changes made are as follows:

- We chose a 1 layer NN as a weak learner because NN has strong fitting ability and is suitable as a weak learner for regression tasks.
- We used mean squared error as the loss function instead of 0/1 loss.

Representation

Let H be the class of base, un-boosted hypotheses. Then, E is defined as the ensemble of H weak learners of size T.

$$E(H,T) = x \to (\Sigma(w_t * h_t(x))) : w \in R^T, \forall t, h_t \in H$$

$h_t(x)$: Prediction result from the t-th weak learner.

w_t Weight of the t-th learner, determined based on its performance.

T: Total number of weak learners in the ensemble.

As can be seen from the above, by combining w_t and $h_t(x)$, the weak learner with higher accuracy has a greater weight in the final prediction.

Loss

The loss function quantifies the error between the model's predicted value and the true value, providing guidance for the optimization process.

For the ensemble Hypothesis, we are using the mean square loss (mse) which is defined as:

$$L_S(E(H,T)) = \frac{1}{m}\sum_{i=1}^{m}(y_i - E(H,T)(\mathbf{x}_i))^2$$

Optimizer

The iterative optimization process of AdaBoosting mainly consists of five steps:

- First, initialization is performed to assign equal weights to all training samples, where m is the number of samples, D is the weight distribution of each training sample, and $D^{(1)} = \frac{1}{m}$.

- Next, in each iteration, a weak learner h_t is trained using the weighted dataset.
- The third step is to calculate the error rate ϵ_t of the weak learner.
- The fourth step is to calculate the weight w_t of the weak learner and adjust the sample weights $D_i^{(t+1)}$ to pay more attention to the misclassified samples.
- Finally, we combine the predictions of all weak learners into the final integrated model.

Through the flexible weight adjustment and weighted integration strategy of the prediction results during the iteration process, we can gradually improve the accuracy of the results.

below is the pesudoCode:

Input:

- Training set $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)\}$
- Weak learner Wl
- Number of estimators T

Initialize: $D^{(1)} = \left(\frac{1}{m}, \ldots, \frac{1}{m}\right)$

For $t = 1, \ldots, T$:

$h_t = WL(D^{(t)}, S)$

$\epsilon_t = \sum_{i=1}^m D_i^{(t)} (y_i - h_t(\mathbf{x_i}))^2$

$w_t = \frac{1}{2} \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$

$$D_i^{(t+1)} = \frac{D_i^{(t)} \exp(-w_t y_i h_t(\mathbf{x}_i))}{\sum_{j=1}^m D_j^{(t)} \exp(-w_t y_j h_t(\mathbf{x}_j))} \quad \forall i = 1, \ldots, m$$

Output:

- The hypothesis: $h_S(\mathbf{x}) = \sum_{t=1}^T w_t h_t(\mathbf{x})$

Weak Learner

Typically a simple learning model that performs slightly better than random guessing, with limited results. We decided to see if we could improve the performance of a One Layer Neural Network (even though it is a "strong" learner in general).

Representation

- The representation of One Layer Neural Network is below, which is equivalent to linear regression:

$$h(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b.$$

- We then add data weighted learning capabilites to the model (i.e. we can tell the model how important a piece of data is.)

$$h(\mathbf{x_i}) = D_i \langle \mathbf{w}, \mathbf{x_i} \rangle + b.$$

Where D_i is the ith data weight.

Loss

We are using weighted L2 loss for the one-layer neural network and MSE for the boosted ensemble of hypotheses.

- For each single-layer neural network, loss is defined as:

$$L_S(h_\mathbf{t}) = \sum_{i=1}^{m} w_i * (y_i - h_\mathbf{t}(\mathbf{x}_i))^2$$

where y_i is the target value of i^{th} sample

$h_\mathbf{t}(\mathbf{x})$ is the predicted value of that sample given the learned model weights

w_m is the weight for the mth data point.

Optimizer

By using Stochastic Gradient Descent as the optimizer, our one-layer neural network can gradually update the weights and biases to optimize the loss function.

$\frac{\partial L_i}{\partial \mathbf{W}} = (\hat{y}_i - y_i)\mathbf{x}_i$

$\frac{\partial L_i}{\partial b} = \hat{y}_i - y_i$

$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta \frac{\partial L_i}{\partial \mathbf{W}}$

$b_{t+1} = b_t - \eta \frac{\partial L_i}{\partial b}$

$\hat{y}_i = f(\mathbf{W}^\top \mathbf{x}_i + b)$

where:

w: Weights of the model

b: Bias

η: Learning rate

y_i: Target value

y^i: Predicted results

l_i: Loss for the i-th datapoints

Check Version

In [1]:

```
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 = {'matplotlib': "3.9.1", 'numpy': "2.0.1",'sklearn': "1.5.1",
                'pandas': "2.2.2"}

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

```
[ OK ] Python version is 3.12.5

[ OK ] matplotlib version 3.9.1 is installed.
[ OK ] numpy version 2.0.1 is installed.
[ OK ] sklearn version 1.5.1 is installed.
[ OK ] pandas version 2.2.2 is installed.
```

Model Section

Weak Learner: One Layer Neural Network

In [2]:

```
import numpy as np
import random

def l2_loss_weight(predictions,Y,weights):
    '''
        Computes L2 loss (sum squared loss) between true values, Y, and predicti
        that are weighted
        :param Y: A 1D Numpy array with real values (float64)
        :param predictions: A 1D Numpy array of the same size of Y
        :param weights: A 1D Numpy array of the same size of Y,
        :return: Weighted L2 loss using predictions for Y.
    '''

    return np.sum(weights * (predictions - Y)**2)

from sklearn.base import BaseEstimator, RegressorMixin

class OneLayerNN(BaseEstimator, RegressorMixin):
    '''
        One layer neural network trained with Stocastic Gradient Descent (SGD)
    '''
    def __init__(self, learning_rate = 0.001, num_epochs = 25, batch_size = 1):
        '''
        @attrs:
            weights: The weights of the neural network model.
            batch_size: The number of examples in each batch
            learning_rate: The learning rate to use for SGD
            epochs: The number of times to pass through the dataset
            v: The resulting predictions computed during the forward pass
        '''
        # initialize self.weights in fit()
        self.weights = None
        self.learning_rate = learning_rate
        self.num_epochs = num_epochs
        self.batch_size = batch_size

        # initialize self.v in forward_pass()
        self.v = None
        self.data_weights = None

    def fit(self, X, Y, data_weights=None):
        '''
        Trains the OneLayerNN model using SGD.
        :param X: 2D Numpy array where each row contains an example
        :param Y: 1D Numpy array containing the corresponding values for each ex
        :param print_loss: If True, print the loss after each epoch.
        :return: None
```

```
        '''
        # initialize weights
        num_examples, num_features = X.shape
        self.weights = np.random.uniform(0, 1, (1, num_features))
        if data_weights is None:
            self.data_weights = np.ones(num_examples)
        else:
            self.data_weights = data_weights / max(data_weights)
        # Train network for certain number of epochs
        for epoch in range(self.num_epochs):
            # Shuffle the examples (X) and labels (Y)
            indices = np.random.permutation(num_examples)
            X_shuffled = X[indices]
            Y_shuffled = Y[indices]
            data_weights_shuffled = self.data_weights[indices]
        # iterate through the examples in batch size increments
            for i in range(num_examples):

                x_i = X_shuffled[i].reshape(1, num_features)
                y_i = Y_shuffled[i].reshape(1, 1)
                data_i = data_weights_shuffled[i]
                # Perform the forward and backward pass on the current batch
                self.forward_pass(x_i)
                self.backward_pass(x_i, y_i, data_i)

            # Print the loss after every epoch
            # if print_loss:
            #     print('Epoch: {} | Loss: {}'.format(epoch, self.loss(X, Y)))

    def forward_pass(self, X):
        '''
        Computes the predictions for a single layer given examples X and
        stores them in self.v
        :param X: 2D Numpy array where each row contains an example.
        :return: None
        '''
        self.v = np.dot(self.weights, X.T).flatten()

    def backward_pass(self, X, Y, data_weights):
        '''
        Computes the weights gradient and updates self.weights
        :param X: 2D Numpy array where each row contains an example
        :param Y: 1D Numpy array containing the corresponding values for each ex
        :return: None
        '''
        # Compute the gradients for the model's weights using backprop
        gradient = self.backprop(X, Y, data_weights)
        # Update the weights using gradient descent
        self.gradient_descent(gradient)

    def backprop(self, X, Y, data_weights):
        '''
        Returns the average weights gradient for the given batch
        :param X: 2D Numpy array where each row contains an example.
        :param Y: 1D Numpy array containing the corresponding values for each ex
        :return: A 1D Numpy array representing the weights gradient
        '''
        # Compute the average weights gradient
        loss = self.v - Y
        return np.dot(2*loss*data_weights, X)
```

```
    def gradient_descent(self, grad_W):
        '''
        Updates the weights using the given gradient
        :param grad_W: A 1D Numpy array representing the weights gradient
        :return: None
        '''
        self.weights -= (self.learning_rate * grad_W)

    def loss(self, X, Y, data_weights):
        '''
        Returns the total squared error on some dataset (X, Y).
        :param X: 2D Numpy array where each row contains an example
        :param Y: 1D Numpy array containing the corresponding values for each ex
        :return: A float which is the squared error of the model on the dataset
        '''
        # Perform the forward pass and compute the l2 loss
        self.forward_pass(X)
        return l2_loss_weight(self.v, Y, data_weights)

    def average_loss(self, X, Y, data_weights):
        '''
        Returns the mean squared error on some dataset (X, Y).
        MSE = Total squared error/# of examples
        :param X: 2D Numpy array where each row contains an example
        :param Y: 1D Numpy array containing the corresponding values for each ex
        :return: A float which is the mean squared error of the model on the dat
        '''
        return self.loss(X, Y, data_weights) / X.shape[0]
    def predict(self, X):
        '''
            Returns the predicted values for some dataset (X).
            :param X: 2D Numpy array where each row contains an example
            :return: 1D Numpy array containing the predicted values for each exa
        '''
        self.forward_pass(X)
        return self.v
```

Boosting Model

In [3]:

```
class Boosted_Model:
  def __init__(self, n_estimators=50, learning_rate=0.5, random_state=1):
    self.n_estimators = n_estimators
    self.learning_rate = learning_rate
    self.random_state = random_state
    self.estimator_weights = np.zeros(self.n_estimators)
    self.data_weights = []

    # Initialize the estimators
    self.estimators = []
    for i in range(self.n_estimators):
      self.estimators.append(OneLayerNN())


  def train(self, X, y):
    '''
    Trains/Fits the Boosting Model using AdaBoost.
    :param X: 2D Numpy array where each row contains an example
    :param Y: 1D Numpy array containing the corresponding values for each exampl
```

```
        '''
        # Initialize the data and estimator weights
        num_inputs = X.shape[0]
        self.data_weights = np.ones(num_inputs) / num_inputs

        # For each round/weak learner
        for i in range(self.n_estimators):

          # Use the weak learner
          weak_learner = self.estimators[i]

          # Fit the weak learner
          weak_learner.fit(X, y, self.data_weights)
          # print(self.data_weights)
          #print("Before Reshape", weak_learner.predict(X).shape)
          y_pred = weak_learner.predict(X).reshape(num_inputs)
          #print("y_pred", y_pred)

          e_t = np.sum(self.data_weights * ((y-y_pred) ** 2)) / np.sum(self.data_wei
          #print(e_t)
          e_t = np.clip(e_t, 1e-10, 0.49)

          #e_t = e_t / num_inputs
          #print("weighted error", e_t)

          w_t = 0.5 * np.log((1 - e_t) / e_t)
          #print("w_t", w_t)

          self.estimator_weights[i] = w_t

          self.data_weights *= np.exp(-w_t * ((y-y_pred) ** 2))
          self.data_weights = np.clip(self.data_weights, a_min=1e-10, a_max=1e2)
          self.data_weights /= np.sum(self.data_weights)
          #print(self.data_weights)
          #print(f"Sum of data weights (should be 1): {np.sum(self.data_weights)}")

        self.estimator_weights /= np.sum(self.estimator_weights)
        #print(self.estimator_weights)


    def loss(self, X, Y):
        # Get predictions from all learners, then weight them
        #print("Shape of Y", Y.shape)
        #print("one prediction", self.estimators[0].predict(X).shape)
        predictions = np.array([e.predict(X).reshape(-1) for e in self.estimators]
        #print("prediction shape", predictions.shape)
        #predictions = predictions.reshape(self.n_estimators, Y.shape[0])
        #print("prediction shape", predictions.shape)
        #print("Original Pred", predictions)
        #print("Estimator Weights", self.estimator_weights)
        weighted_predictions = np.dot(self.estimator_weights, predictions)
        #print("Weighted", weighted_predictions)
        #print(Y)
        # L2 loss
        loss = np.mean((Y - weighted_predictions) ** 2)
        return loss
```

Check Model

In [4]:
```
import pytest
# Sets random seed for testing purposes
random.seed(0)
np.random.seed(0)


def test_OneLayerNN():
    '''
    Tests for OneLayerNN Model Weights Gradient
    '''

    test_model = OneLayerNN()

    # 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])

    # Tests the functionality with no weights (this is the same unit test from t
    no_weights = np.array([1, 1, 1, 1, 1])

    # Test Model Train
    test_model.fit(x_bias, y, no_weights)

    act_weights = test_model.weights
    exp_weights = np.array([[ 0.17817953, -0.03543112,  0.34761945]])

    print('----Testing 1-Layer NN Gradients-----')

    print("\nTesting layer one weights gradient.")

    # Test layer 1 weights
    if not hasattr(act_weights, "shape"):
        print("Layer one weights gradient is not a numpy array. \n")
    elif act_weights.shape != (1, 3):
        print(
            f"Incorrect shape for layer one weights gradient.\nExpected: {(1, 3)
    elif not act_weights == pytest.approx(exp_weights, .01):
        print(
            f"Incorrect values for layer one weights gradient.\nExpected: {exp_w
    else:
        print("Layer one weights gradient is correct.\n")

    print('----Testing 1-Layer NN with Data Weights-----')

    # Tests the same functionality but passes in different weights. Weights 1 ha
    # for each data point, while weights 2 is an edge case where there's only on
    data_weights_1 = np.array([0, 0.1, 0.2, 0.3, 0.4])
    data_weights_2 = np.array([0, 0, 0, 0, 1])

    print("Weights 1", data_weights_1)
    print("Weights 2:", data_weights_2)

    test_model1 = OneLayerNN(num_epochs=2)
    test_model2 = OneLayerNN(num_epochs=2)

    # Test Model Train

    test_model1.fit(x_bias, y, data_weights_1)
    test_model2.fit(x_bias, y, data_weights_2)
```

```
    act_weights1 = test_model1.weights
    act_weights2 = test_model2.weights

    exp_weights1 = np.array([[0.87395, 0.583062, -0.018167]])
    exp_weights2 = np.array([[0.9767610, 0.5315327, 0.7246010]])

    if not hasattr(act_weights, "shape"):
        print("Layer one weights gradient is not a numpy array. \n")
    elif act_weights.shape != (1, 3):
        print(
            f"Incorrect shape for layer one weights gradient for data weights 1.
    elif not act_weights1 == pytest.approx(exp_weights1, .01):
        print(
            f"Incorrect values for data weights 1 weights gradient.\nExpected: {
    elif not act_weights2 == pytest.approx(exp_weights2, .01):
        print(
            f"Incorrect values for data weights 2 weights gradient.\nExpected: {
    else:
        print("Layer one weights gradient with weighted data is correct.\n")

    print('----Testing Weighted Loss Function -----')

    # These tests test that the weighted loss function is working properly for t

    test_x = np.array([[0,2,1], [1,3,1], [4,2,1], [1,5,1], [0,1,1]])
    test_y = np.array([0,1,1,0,1])

    loss_normal = test_model.loss(test_x, test_y, no_weights)
    loss1 = test_model1.loss(test_x, test_y, data_weights_1)
    loss2 = test_model2.loss(test_x, test_y, data_weights_2)

    exp_ln = 1
    exp_loss1 = 7.25
    exp_loss2 = 0.0656

    if not loss_normal == pytest.approx(exp_ln, .01):
        print(
            f"Incorrect values for mo weights loss function.\nExpected: {exp_ln}
    elif not loss1 == pytest.approx(exp_loss1, .01):
        print(
            f"Incorrect values for loss with weights.\nExpected: {exp_loss1} \nA
    elif not loss2 == pytest.approx(exp_loss2, .01):
        print(
            f"Incorrect values for loss with weights .\nExpected: {exp_loss2} \n
    else:
        print("Weighted Loss Function is Correct.\n")

test_OneLayerNN()
```

```
----Testing 1-Layer NN Gradients-----

Testing layer one weights gradient.
Layer one weights gradient is correct.

----Testing 1-Layer NN with Data Weights-----
Weights 1 [0.  0.1 0.2 0.3 0.4]
Weights 2: [0 0 0 0 1]
Layer one weights gradient with weighted data is correct.

----Testing Weighted Loss Function -----
Weighted Loss Function is Correct.
```

In [16]:
```python
def test_Boosted_NN():

    print('-----Testing for Boosted_NN Model-----')

    np.random.seed(4)
    random.seed(4)

    # Standard Test Case
    print("\nTesting with standard test case.")
    boosted_model = Boosted_Model(n_estimators=10, learning_rate=0.1, random_sta
    X_test = np.array([[0, 4, 1], [0, 3, 1], [5, 0, 1], [4, 1, 1], [0, 5, 1]])
    Y_test = np.array([0, 0, 1, 1, 0])
    boosted_model.train(X_test, Y_test)
    computed_loss = boosted_model.loss(X_test, Y_test)
    if computed_loss == pytest.approx(0.02263503, .01):
        print("Model computed loss is correct.")
    else:
        print("Model computed loss is NOT correct.")

    expected_weights = [0.07516408, 0.10241473, 0.09759651, 0.14614814, 0.130472
    if boosted_model.estimator_weights == pytest.approx(expected_weights, 0.01):
        print("Model estimator weights are correct")
    else:
        print("Model estimator weights are NOT correct")

    # Estimator Weights Normalization
    print("\nTesting estimator weights normalization.")
    weight_sum = np.sum(boosted_model.estimator_weights)
    if not np.isclose(weight_sum, 1.0, atol=1e-6):
        print(f"Estimator weights are not normalized.\nSum of weights: {weight_s
    else:
        print("Estimator weights are normalized correctly.")

    # Individual Estimator Predictions
    print("\nTesting individual estimator predictions.")
    weak_predictions = []
    for estimator in boosted_model.estimators:
        pred = estimator.predict(X_test).reshape(-1)
        weak_predictions.append(pred)
    weak_predictions = np.array(weak_predictions)

    if weak_predictions.shape != (boosted_model.n_estimators, X_test.shape[0]):
        print(f"Incorrect shape for weak learner predictions.\nExpected: {(boost
              f"Actual: {weak_predictions.shape}")
    else:
        print("Weak learner predictions shape is correct.")
```

```
# Edge Case 1: Single Data Point
print("\n----------------------------------")
print("\nTesting with a single data point.")
X_test_single = np.array([[0, 0, 0]])
Y_test_single = np.array([1])
boosted_model.train(X_test_single, Y_test_single)
computed_loss_single = boosted_model.loss(X_test_single, Y_test_single)
if computed_loss_single == 1.0:
    print("Model computed loss is correct.")
else:
    print("Model computed loss is NOT correct.")
# Estimator Weights Normalization
print("\nTesting estimator weights normalization.")
weight_sum = np.sum(boosted_model.estimator_weights)
if not np.isclose(weight_sum, 1.0, atol=1e-6):
    print(f"Estimator weights are not normalized.\nSum of weights: {weight_s
else:
    print("Estimator weights are normalized correctly.")

# Individual Estimator Predictions
print("\nTesting individual estimator predictions.")
weak_predictions = []
for estimator in boosted_model.estimators:
    pred = estimator.predict(X_test).reshape(-1)
    weak_predictions.append(pred)
weak_predictions = np.array(weak_predictions)

if weak_predictions.shape != (boosted_model.n_estimators, X_test.shape[0]):
    print(f"Incorrect shape for weak learner predictions.\nExpected: {(boost
          f"Actual: {weak_predictions.shape}")
else:
    print("Weak learner predictions shape is correct.")



# Edge Case 2: All Inputs the Same
print("\n----------------------------------")
print("\nTesting with identical input values.")
X_test_identical = np.array([[1, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1, 1], [1,
Y_test_identical = np.array([0, 0, 1, 1, 0])
boosted_model.train(X_test_identical, Y_test_identical)
computed_loss_identical = boosted_model.loss(X_test_identical, Y_test_identi
if computed_loss_identical == pytest.approx(0.32526973, .01):
    print("Model computed loss is correct.")
else:
    print("Model computed loss is NOT correct.")

# Estimator Weights Normalization
print("\nTesting estimator weights normalization.")
weight_sum = np.sum(boosted_model.estimator_weights)
if not np.isclose(weight_sum, 1.0, atol=1e-6):
    print(f"Estimator weights are not normalized.\nSum of weights: {weight_s
else:
    print("Estimator weights are normalized correctly.")

# Individual Estimator Predictions
print("\nTesting individual estimator predictions.")
```

```
    weak_predictions = []
    for estimator in boosted_model.estimators:
        pred = estimator.predict(X_test).reshape(-1)
        weak_predictions.append(pred)
    weak_predictions = np.array(weak_predictions)

    if weak_predictions.shape != (boosted_model.n_estimators, X_test.shape[0]):
        print(f"Incorrect shape for weak learner predictions.\nExpected: {(boost
              f"Actual: {weak_predictions.shape}")
    else:
        print("Weak learner predictions shape is correct.")


    # Edge Case 3: All Labels Same
    print("\n--------------------------------")
    print("\nTesting with all labels being the same.")
    X_test_labels_same = np.array([[0, 0, 0], [1, 1, 1], [2, 2, 2], [3, 3, 3], [
    Y_test_labels_same = np.array([1, 1, 1, 1, 1])  # All labels are 1
    boosted_model.train(X_test_labels_same, Y_test_labels_same)
    computed_loss_labels_same = boosted_model.loss(X_test_labels_same, Y_test_la
    if computed_loss_labels_same == pytest.approx(0.34004895, .01):
        print("Model computed loss is correct.")
    else:
        print("Model computed loss is NOT correct.")

    expected_weights = [0.04664099, 0.05977069, 0.07518574, 0.09233205, 0.111698
    if boosted_model.estimator_weights == pytest.approx(expected_weights, 0.01):
        print("Model estimator weights are correct")
    else:
        print("Model estimator weights are NOT correct")

    # Estimator Weights Normalization
    print("\nTesting estimator weights normalization.")
    weight_sum = np.sum(boosted_model.estimator_weights)
    if not np.isclose(weight_sum, 1.0, atol=1e-6):
        print(f"Estimator weights are not normalized.\nSum of weights: {weight_s
    else:
        print("Estimator weights are normalized correctly.")

    # Individual Estimator Predictions
    print("\nTesting individual estimator predictions.")
    weak_predictions = []
    for estimator in boosted_model.estimators:
        pred = estimator.predict(X_test).reshape(-1)
        weak_predictions.append(pred)
    weak_predictions = np.array(weak_predictions)

    if weak_predictions.shape != (boosted_model.n_estimators, X_test.shape[0]):
        print(f"Incorrect shape for weak learner predictions.\nExpected: {(boost
              f"Actual: {weak_predictions.shape}")
    else:
        print("Weak learner predictions shape is correct.")



    # Edge Case 4: High-Dimensional Input
    print("\n--------------------------------")
    print("\nTesting with high-dimensional input.")
    X_test_high_dim = np.random.rand(5, 50)  # 5 samples, 50 features
    Y_test_high_dim = np.array([0, 1, 0, 1, 0])
```

```
    boosted_model.train(X_test_high_dim, Y_test_high_dim)
    computed_loss_high_dim = boosted_model.loss(X_test_high_dim, Y_test_high_dim
    if np.isclose(computed_loss_high_dim, 0.34004895, atol=1e-6):
        print("Model computed loss is correct.")
    else:
        print("Model computed loss is NOT correct.")

    # Estimator Weights Normalization
    print("\nTesting estimator weights normalization.")
    weight_sum = np.sum(boosted_model.estimator_weights)
    if not np.isclose(weight_sum, 1.0, atol=1e-6):
        print(f"Estimator weights are not normalized.\nSum of weights: {weight_s
    else:
        print("Estimator weights are normalized correctly.")

    # Individual Estimator Predictions
    print("\nTesting individual estimator predictions.")
    weak_predictions = []
    for estimator in boosted_model.estimators:
        pred = estimator.predict(X_test_high_dim).reshape(-1)
        weak_predictions.append(pred)
    weak_predictions = np.array(weak_predictions)

    if weak_predictions.shape != (boosted_model.n_estimators, X_test_high_dim.sh
        print(f"Incorrect shape for weak learner predictions.\nExpected: {(boost
            f"Actual: {weak_predictions.shape}")
    else:
        print("Weak learner predictions shape is correct.")

    # Edge Case 5: Very Large Dataset
    print("\n---------------------------------")
    print("\nTesting with a very large dataset.")
    X_test_large = np.random.rand(10000, 10)  # 10,000 samples, 10 features
    Y_test_large = np.random.randint(0, 2, size=10000)
    boosted_model.train(X_test_large, Y_test_large)
    computed_loss_large = boosted_model.loss(X_test_large, Y_test_large)
    if np.isclose(computed_loss_large, 0.263257448, atol=1e-6):
        print("Model computed loss is correct.")
    else:
        print("Model computed loss is NOT correct.")

    # Estimator Weights Normalization
    print("\nTesting estimator weights normalization.")
    weight_sum = np.sum(boosted_model.estimator_weights)
    if not np.isclose(weight_sum, 1.0, atol=1e-6):
        print(f"Estimator weights are not normalized.\nSum of weights: {weight_s
    else:
        print("Estimator weights are normalized correctly.")

    # Individual Estimator Predictions
    print("\nTesting individual estimator predictions.")
    weak_predictions = []
    for estimator in boosted_model.estimators:
        pred = estimator.predict(X_test_large).reshape(-1)
        weak_predictions.append(pred)
    weak_predictions = np.array(weak_predictions)

    if weak_predictions.shape != (boosted_model.n_estimators, X_test_large.shape
        print(f"Incorrect shape for weak learner predictions.\nExpected: {(boost
            f"Actual: {weak_predictions.shape}")
```

```
    else:
        print("Weak learner predictions shape is correct.")

    # Edge Case 6: Edge Inputs (Zeros and Ones)
    print("\n--------------------------------")
    print("\nTesting with edge inputs (all zeros and ones).")
    X_test_edges = np.array([[0, 0, 0], [1, 1, 1], [0, 1, 0], [1, 0, 1], [1, 1,
    Y_test_edges = np.array([0, 1, 0, 1, 0])
    boosted_model.train(X_test_edges, Y_test_edges)
    computed_loss_edges = boosted_model.loss(X_test_edges, Y_test_edges)
    if np.isclose(computed_loss_edges, 0.10650355, atol=1e-6):
        print("Model computed loss is correct.")
    else:
        print("Model computed loss is NOT correct.")

    # Estimator Weights Normalization
    print("\nTesting estimator weights normalization.")
    weight_sum = np.sum(boosted_model.estimator_weights)
    if not np.isclose(weight_sum, 1.0, atol=1e-6):
        print(f"Estimator weights are not normalized.\nSum of weights: {weight_s
    else:
        print("Estimator weights are normalized correctly.")

    # Individual Estimator Predictions
    print("\nTesting individual estimator predictions.")
    weak_predictions = []
    for estimator in boosted_model.estimators:
        pred = estimator.predict(X_test_edges).reshape(-1)
        weak_predictions.append(pred)
    weak_predictions = np.array(weak_predictions)

    if weak_predictions.shape != (boosted_model.n_estimators, X_test_edges.shape
        print(f"Incorrect shape for weak learner predictions.\nExpected: {(boost
            f"Actual: {weak_predictions.shape}")
    else:
        print("Weak learner predictions shape is correct.")

test_Boosted_NN()
```

```
-----Testing for Boosted_NN Model-----

Testing with standard test case.
Model computed loss is correct.
Model estimator weights are correct

Testing estimator weights normalization.
Estimator weights are normalized correctly.

Testing individual estimator predictions.
Weak learner predictions shape is correct.

---------------------------------

Testing with a single data point.
Model computed loss is correct.

Testing estimator weights normalization.
Estimator weights are normalized correctly.

Testing individual estimator predictions.
Weak learner predictions shape is correct.

---------------------------------

Testing with identical input values.
Model computed loss is correct.

Testing estimator weights normalization.
Estimator weights are normalized correctly.

Testing individual estimator predictions.
Weak learner predictions shape is correct.

---------------------------------

Testing with all labels being the same.
Model computed loss is NOT correct.
Model estimator weights are correct

Testing estimator weights normalization.
Estimator weights are normalized correctly.

Testing individual estimator predictions.
Weak learner predictions shape is correct.

---------------------------------

Testing with high-dimensional input.
Model computed loss is correct.

Testing estimator weights normalization.
Estimator weights are normalized correctly.

Testing individual estimator predictions.
Weak learner predictions shape is correct.

---------------------------------

Testing with a very large dataset.
```

```
Model computed loss is correct.

Testing estimator weights normalization.
Estimator weights are normalized correctly.

Testing individual estimator predictions.
Weak learner predictions shape is correct.

---------------------------------

Testing with edge inputs (all zeros and ones).
Model computed loss is correct.

Testing estimator weights normalization.
Estimator weights are normalized correctly.

Testing individual estimator predictions.
Weak learner predictions shape is correct.
```

Show that it can be reproduced by sklearn's AdaBoost Regressor

In [6]:
```python
from sklearn.ensemble import AdaBoostRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import os

def test_sklearn(dataset, test_size=0.2):
    '''
        Tests OneLayerNN, Boost on a given dataset.
        :param dataset The path to the dataset
        :return None
    '''

    # Check if the file exists
    if not os.path.exists(dataset):
        print('The file {} does not exist'.format(dataset))
        exit()

    # Load in the dataset
    data = np.loadtxt(dataset, skiprows = 1)
    X, Y = data[:, 1:], data[:, 0]

    # Normalize the features
    X = (X-np.mean(X, axis=0))/np.std(X, axis=0)
    Y = Y

    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=test_siz
    print('Running models on {} dataset'.format(dataset))


    # Add a bias
    X_train_b = np.append(X_train, np.ones((len(X_train), 1)), axis=1)
    X_test_b = np.append(X_test, np.ones((len(X_test), 1)), axis=1)

    print('----- Our Boosted Network (Our NN) -----')
    n_estimators = 25
    learning = 0.5
```

```
    model = Boosted_Model(n_estimators=n_estimators, learning_rate=learning)
    model.train(X_train_b, Y_train)

    our_train_loss = (model.loss(X_train_b, Y_train))
    our_test_loss = (model.loss(X_test_b, Y_test))

    print('Average Training Loss:', our_train_loss)
    print('Average Testing Loss:', our_test_loss)

    #### sklearn Boosted with our NN ######
    print('----- sklearn Boosted Network (Our NN) -----')

    model = AdaBoostRegressor(OneLayerNN(), n_estimators=n_estimators, learning_

    model.fit(X_train_b, Y_train)

    sklearn_train_loss = mean_squared_error(model.predict(X_train_b), Y_train)
    sklearn_test_loss = mean_squared_error(model.predict(X_test_b), Y_test)

    print('Average Training Loss:', sklearn_train_loss)
    print('Average Testing Loss:', sklearn_test_loss)

    baseline_train = mean_squared_error(np.ones(Y_train.shape) * np.mean(Y_train
    baseline_test = mean_squared_error(np.ones(Y_test.shape) * np.mean(Y_train),

    return our_train_loss, our_test_loss, sklearn_train_loss, sklearn_test_loss,

test_sklearn('../data/wine.txt')
```

```
Running models on ../data/wine.txt dataset
----- Our Boosted Network (Our NN) -----
Average Training Loss: 0.5752329902177458
Average Testing Loss: 0.5615628155504526
----- sklearn Boosted Network (Our NN) -----
Average Training Loss: 0.5747030955673881
Average Testing Loss: 0.5657154893148305
```

Out[6]:

```
(np.float64(0.5752329902177458),
 np.float64(0.5615628155504526),
 np.float64(0.5747030955673881),
 np.float64(0.5657154893148305),
 np.float64(0.7845845577055516),
 np.float64(0.782960316293052))
```

In [7]:

```
# Make a plot over 10 random seeds
import matplotlib.pyplot as plt
num_seeds = 10

our_train_losses, our_test_losses, sklearn_train_losses, sklearn_test_losses, bs

for i in range(num_seeds):
    our_train_loss, our_test_loss, sklearn_train_loss, sklearn_test_loss, bs_tra
    our_train_losses.append(our_train_loss)
    our_test_losses.append(our_test_loss)
    sklearn_train_losses.append(sklearn_train_loss)
    sklearn_test_losses.append(sklearn_test_loss)
    bs_trains.append(bs_train)
    bs_tests.append(bs_test)
```

```
Running models on ../data/wine.txt dataset
----- Our Boosted Network (Our NN) -----
Average Training Loss: 0.5682574919322486
Average Testing Loss: 0.6011007771395985
----- sklearn Boosted Network (Our NN) -----
Average Training Loss: 0.5638947713193695
Average Testing Loss: 0.5922916349767658
Running models on ../data/wine.txt dataset
----- Our Boosted Network (Our NN) -----
Average Training Loss: 0.5671654416189561
Average Testing Loss: 0.6000681051614906
----- sklearn Boosted Network (Our NN) -----
Average Training Loss: 0.566011293727811
Average Testing Loss: 0.5946491363651013
Running models on ../data/wine.txt dataset
----- Our Boosted Network (Our NN) -----
Average Training Loss: 0.5772881725799439
Average Testing Loss: 0.551295020360692
----- sklearn Boosted Network (Our NN) -----
Average Training Loss: 0.577574179054741
Average Testing Loss: 0.5471304103047739
Running models on ../data/wine.txt dataset
----- Our Boosted Network (Our NN) -----
Average Training Loss: 0.5717846561226821
Average Testing Loss: 0.560669116463898
----- sklearn Boosted Network (Our NN) -----
Average Training Loss: 0.5748294374115468
Average Testing Loss: 0.5698265675777416
Running models on ../data/wine.txt dataset
----- Our Boosted Network (Our NN) -----
Average Training Loss: 0.5728187621543281
Average Testing Loss: 0.5719122498960694
----- sklearn Boosted Network (Our NN) -----
Average Training Loss: 0.5698564749505625
Average Testing Loss: 0.555436725460365
Running models on ../data/wine.txt dataset
----- Our Boosted Network (Our NN) -----
Average Training Loss: 0.5755118886662816
Average Testing Loss: 0.5594658810289141
----- sklearn Boosted Network (Our NN) -----
Average Training Loss: 0.5768110389359689
Average Testing Loss: 0.5602821422466827
Running models on ../data/wine.txt dataset
----- Our Boosted Network (Our NN) -----
Average Training Loss: 0.5572847999793845
Average Testing Loss: 0.6381429996203657
----- sklearn Boosted Network (Our NN) -----
Average Training Loss: 0.5514164908845389
Average Testing Loss: 0.6390300380403038
Running models on ../data/wine.txt dataset
----- Our Boosted Network (Our NN) -----
Average Training Loss: 0.5828823533061807
Average Testing Loss: 0.539013224512257
----- sklearn Boosted Network (Our NN) -----
Average Training Loss: 0.5839043783802255
Average Testing Loss: 0.5344629767086984
Running models on ../data/wine.txt dataset
----- Our Boosted Network (Our NN) -----
Average Training Loss: 0.5674027857863533
Average Testing Loss: 0.6028377216033031
```

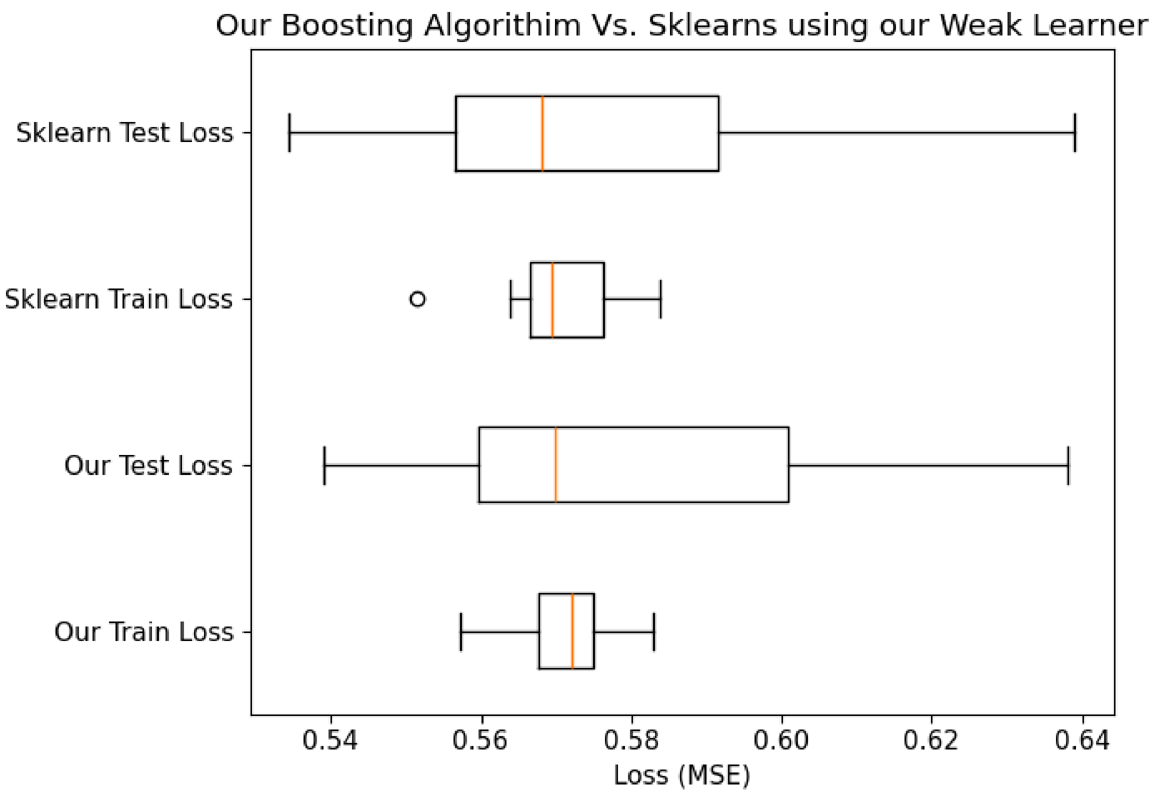
```
----- sklearn Boosted Network (Our NN) -----
Average Training Loss: 0.5677813015379903
Average Testing Loss: 0.5896030734872949
Running models on ../data/wine.txt dataset
----- Our Boosted Network (Our NN) -----
Average Training Loss: 0.5722170666334819
Average Testing Loss: 0.5679727199733051
----- sklearn Boosted Network (Our NN) -----
Average Training Loss: 0.5691079500923687
Average Testing Loss: 0.5663230217500441
```

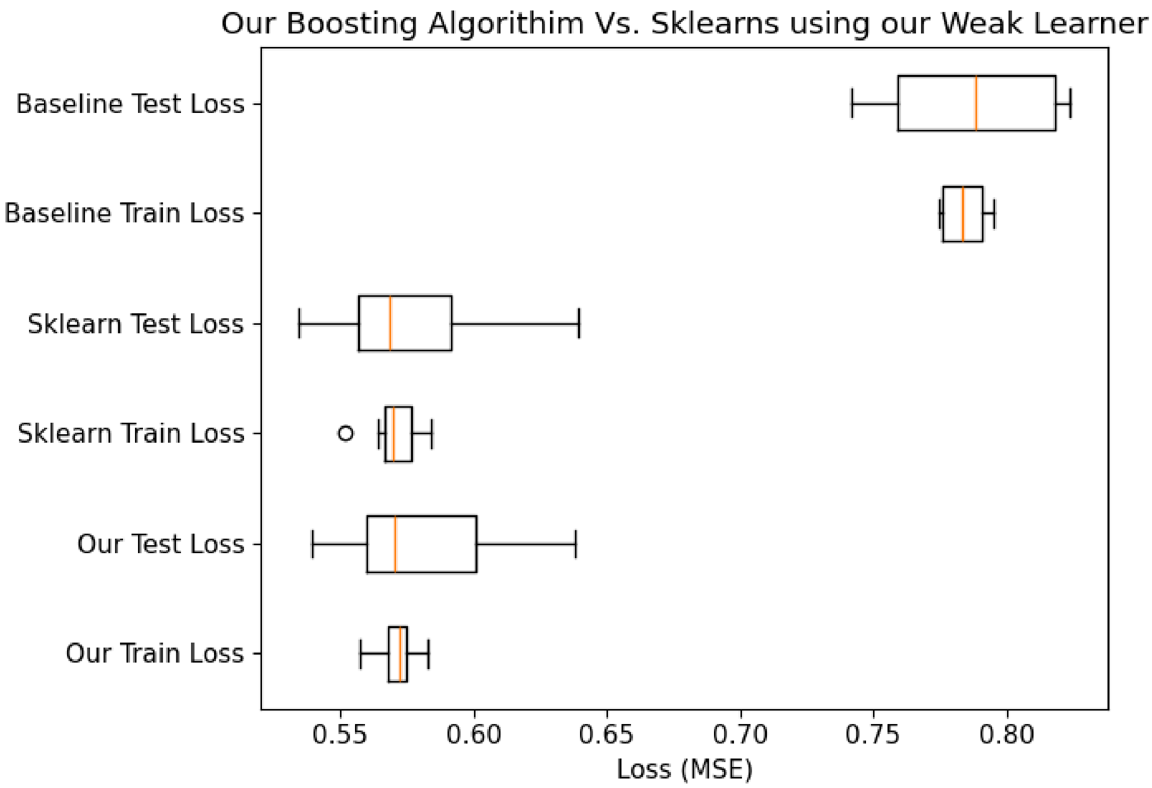
In [8]:

```
losses = [our_train_losses, our_test_losses, sklearn_train_losses, sklearn_test_
labels = ["Our Train Loss", "Our Test Loss", "Sklearn Train Loss", "Sklearn Test
losses_bs = [our_train_losses, our_test_losses, sklearn_train_losses, sklearn_te
labels_bs = ["Our Train Loss", "Our Test Loss", "Sklearn Train Loss", "Sklearn T

plt.rcParams.update({'font.size': 11})
plt.figure(figsize=(7,5))
plt.boxplot(losses,tick_labels=labels,vert=False)
plt.title("Our Boosting Algorithim Vs. Sklearns using our Weak Learner")
plt.xlabel('Loss (MSE)')
plt.tight_layout()
plt.show()

plt.figure(figsize=(7,5))
plt.boxplot(losses_bs,tick_labels=labels_bs,vert=False)
plt.title("Our Boosting Algorithim Vs. Sklearns using our Weak Learner")
plt.xlabel('Loss (MSE)')
plt.tight_layout()
plt.show()
```





Misc Code (Not on Rubric)

In [ ]:

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split

def test_models_with_hyperparameters(dataset, test_size=0.2, n_estimators_list=[
    '''
        Tests OneLayerNN and Boosted_NN with varying hyperparameters.
        :param dataset: The path to the dataset
        :param test_size: Fraction of the dataset to be used for testing
        :param n_estimators_list: List of values for the number of estimators
        :param learning_rate_list: List of values for the learning rate
        :return: None
    '''

    # Check if the file exists
    if not os.path.exists(dataset):
        print(f'The file {dataset} does not exist')
        return

    # Load in the dataset
    data = np.loadtxt(dataset, skiprows=1)
    X, Y = data[:, 1:], data[:, 0]

    # Normalize the features
    X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)

    # Split dataset
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=test_siz

    # Add a bias term to the features
    X_train_b = np.append(X_train, np.ones((len(X_train), 1)), axis=1)
    X_test_b = np.append(X_test, np.ones((len(X_test), 1)), axis=1)
```

```
    # Results storage for plotting
    nn_train_losses = []
    nn_test_losses = []
    boosted_train_losses = {}
    boosted_test_losses = {}

    #### 1-Layer NN ######
    print('----- 1-Layer NN -----')
    nnmodel = OneLayerNN()
    nnmodel.fit(X_train_b, Y_train, np.ones(X_train_b.shape[0]))
    nn_train_loss = nnmodel.average_loss(X_train_b, Y_train, np.ones(X_train_b.s
    nn_test_loss = nnmodel.average_loss(X_test_b, Y_test, np.ones(X_test_b.shape
    nn_train_losses.append(nn_train_loss)
    nn_test_losses.append(nn_test_loss)
    # print('Average Training Loss (1-Layer NN):', nn_train_loss)
    # print('Average Testing Loss (1-Layer NN):', nn_test_loss)

    #### Boosted Neural Networks ######
    print('----- Boosted Neural Networks -----')
    for n_estimators in n_estimators_list:
        for learning_rate in learning_rate_list:
            print(f'Testing Boosted_NN with n_estimators={n_estimators}, learnin
            model = Boosted_Model(n_estimators=n_estimators, learning_rate=learn
            model.train(X_train_b, Y_train)
            train_loss = model.loss(X_train_b, Y_train)
            test_loss = model.loss(X_test_b, Y_test)
            print(f'Training Loss: {train_loss}, Testing Loss: {test_loss}')

            # Store results
            boosted_train_losses[(n_estimators, learning_rate)] = train_loss
            boosted_test_losses[(n_estimators, learning_rate)] = test_loss

    # Plot the results
    fig, ax = plt.subplots(figsize=(12, 6))

    # Plot for Boosted NN
    for n_estimators in n_estimators_list:
        train_losses = [boosted_train_losses[(n_estimators, lr)] for lr in learn
        test_losses = [boosted_test_losses[(n_estimators, lr)] for lr in learnin

        ax.plot(learning_rate_list, train_losses, marker='o', label=f'Train Loss
        ax.plot(learning_rate_list, test_losses, marker='x', label=f'Test Loss (

    ax.set_title('Boosted NN Loss vs Learning Rate')
    ax.set_xlabel('Learning Rate')
    ax.set_ylabel('Loss')
    ax.set_xscale('log')  # Keep the log scale for learning rates
    ax.set_xticks(learning_rate_list)  # Explicitly set the ticks
    ax.get_xaxis().set_major_formatter(plt.ScalarFormatter())  # Ensure proper f
    ax.legend()
    ax.grid(True)
    plt.show()

test_models_with_hyperparameters('../data/wine.txt')
```

References

[1] Freund, Y. and Schapire, R.E. (1997) 'A Decision-Theoretic generalization of On-Line learning and an application to boosting,' Journal of Computer and System Sciences, 55(1), pp. 119–139. https://doi.org/10.1006/jcss.1997.1504.

[2] Drucker, Harris. (1997). Improving Regressors using boosting Techniques. Proceedings of the 14th International Conference on Machine Learning. https://www.researchgate.net/publication/2424244_Improving_Regressors_Using_Boo

[3] UCI Machine Learning Repository (2009). https://archive.ics.uci.edu/dataset/186/wine+quality.