TubesC_13518017

April 2, 2021

```
In [1]: from pandas import *

In [11]: def confusion_matrix(y_true, y_pred):
             possible_val = []
             for val in y_true:
                 if val not in possible_val:
                     possible_val.append(val)

             for val in y_pred:
               if val not in possible_val:
                 possible_val.append(val)

             dic = {}
             possible_val.sort()
             for i in range(len(possible_val)):
                 dic[possible_val[i]] = i

             mat = [[0 for j in range(len(possible_val))]
                    for i in range(len(possible_val))]

             for i in range(len(y_true)):
                 mat[dic[y_true[i]]][dic[y_pred[i]]] += 1

             return mat

In [13]: def prettify(data, index, columns):
             ret = DataFrame(data)
             ret.index = index
             ret.columns = columns
             return ret

In [3]: class Metrics:
            def __init__(self, y_true, y_pred):
                # self.y_true = y_true
                # self.y_pred = y_pred
                self.confusion_matrix = confusion_matrix(y_true, y_pred)
                self.labels = []
                for val in y_true:
```

```
            if val not in self.labels:
                self.labels.append(val)

        for val in y_pred:
            if val not in self.labels:
                self.labels.append(val)

        self.labels.sort()

    def get_label_tp_tn_fp_fn(self, label):
        tp = self.confusion_matrix[label][label]
        tn, fp, fn = 0, 0, 0
        n = len(self.confusion_matrix)
        for i in range(n):
            for j in range(n):
                val = self.confusion_matrix[i][j]
                if (i != label and j != label):
                    tn += val
                if (j == label and i != label):
                    fp += val
                if (i == label and j != label):
                    fn += val
        return tp, tn, fp, fn

    def overall_accuracy(self):
        tp = 0
        tot = 0
        n = len(self.confusion_matrix)
        for i in range(n):
            tp += self.confusion_matrix[i][i]
            for j in range(n):
                tot += self.confusion_matrix[i][j]
        return tp / tot

    # label yg dimaksud itu indeks labelnya
    # label = ["a", "b", "c"]
    # kalo mau accuracy "a" manggil self.accuracy(0)
    def accuracy(self, label):
        tp, tn, fp, fn = self.get_label_tp_tn_fp_fn(label)
        if(tp + tn == 0):
          return 0
        return (tp + tn) / (tp + tn + fn + fp)

    def all_accuracy(self):
        n = len(self.confusion_matrix)
        tot = 0
        for i in range(n):
            tot += self.accuracy(i)
```

```python
        if(tot == 0):
            return 0
        return tot / n

    def precision(self, label):
        tp, tn, fp, fn = self.get_label_tp_tn_fp_fn(label)
        if(tp == 0):
          return 0
        return tp / (tp + fp)

    def all_precision(self):
        n = len(self.confusion_matrix)
        tot = 0
        for i in range(n):
            tot += self.precision(i)
        if(tot == 0):
            return 0
        return tot / n

    def recall(self, label):
        tp, tn, fp, fn = self.get_label_tp_tn_fp_fn(label)
        if(tp == 0):
          return 0
        return tp / (tp + fn)

    def all_recall(self):
        n = len(self.confusion_matrix)
        tot = 0
        for i in range(n):
            tot += self.recall(i)
        if(tot == 0):
            return 0
        return tot / n

    def f1_score(self, label):
        tp, tn, fp, fn = self.get_label_tp_tn_fp_fn(label)
        precision = self.precision(label)
        recall = self.recall(label)
        if(precision * recall == 0):
            return 0
        return 2 * precision * recall / (precision + recall)

    def all_f1_score(self):
        n = len(self.confusion_matrix)
        tot = 0
        for i in range(n):
            tot += self.f1_score(i)
```

```
            if(tot == 0):
                return 0
            return tot / n

        def report(self, digits = 3):
            # accuracy, precision, recall, f1
            n = len(self.confusion_matrix)
            ret = [[0 for j in range(4)] for i in range(n)]
            for i in range(n):
                ret[i][0] = round(self.accuracy(i), digits)
                ret[i][1] = round(self.precision(i), digits)
                ret[i][2] = round(self.recall(i), digits)
                ret[i][3] = round(self.f1_score(i), digits)

            print(prettify(ret, self.labels, ["accuracy", "precision", "recall", "f1"]))
            print(f'overall accuracy: {self.all_accuracy():.3f}')
            print(f'overall precision: {self.all_precision():.3f}')
            print(f'overall recall: {self.all_recall():.3f}')
            print(f'overall f1_score: {self.all_f1_score():.3f}')
```

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

In [5]:
```
def sigmoid(x):
    # applying the sigmoid function
    return 1 / (1 + np.exp(-x))


def sigmoid_derivative(x):
    # computing derivative to the Sigmoid function
    return x * (1 - x)


def relu(x):
    # compute relu
    return np.maximum(0, x)


def relu_derivative(x):
    return np.where(x < 0, 0, 1)


def linear(x):
    return x


def linear_derivative(x):
    return np.full(x.shape, 1)
```

```python
def softmax(x):
    ret = np.zeros(x.shape)
    for i in range(x.shape[1]):
        ex = np.exp(x[:, i])
        x[:, i] = ex / np.sum(ex)
    return ret


def softmax_derivative(x):
    ret = np.zeros(x.shape);
    for i in range(x.shape[1]):
        j = np.sum(softmax_derivative_util(x[:, i]), axis=1)
        ret[:, i] = j
    return ret


def softmax_derivative_util(x):
    rx = x.reshape(-1, 1)
    return np.diagflat(rx) - np.dot(rx, rx.T)


def sum_of_squared_error(t, o):
    sub = t - o
    return 0.5 * np.sum(sub**2)


def cross_entropy(t, o):
    ret = 0
    for i in range(o.shape[1]):
        j = np.argmax(o[:, i])
        ct = t[j, i]
        # ct = clip_scalar(t[j, i])
        ret += -np.log2(ct)
    return ret


def cross_entropy_derivative(t, o):
    ret = o
    for i in range(o.shape[1]):
        j = np.argmax(o[:, i])
        ret[j, i] = -(1-ret[j, i])
    return ret


clip_upper_threshold = 5
clip_lower_threshold = 0.5
```

```
def clip(x):
    ret = x
    norm = np.sum(x * x)
    if norm > clip_upper_threshold ** 2:
        ret = ret * (clip_upper_threshold / np.sqrt(norm))
#     if norm < clip_lower_threshold ** 2:
#         ret = ret * (clip_lower_threshold / np.sqrt(norm))
    return ret


activation_functions = {
    # activation_functions
    "relu": relu,
    "sigmoid": sigmoid,
    "softmax": softmax,
    "linear": linear,
}

activation_functions_derivative = {
    # activation_functions_derivative
    "relu": relu_derivative,
    "sigmoid": sigmoid_derivative,
    "softmax": softmax_derivative,
    "linear": linear_derivative
}

error_functions = {
    # error_functions
    "relu": sum_of_squared_error,
    "sigmoid": sum_of_squared_error,
    "softmax": cross_entropy,
    "linear": sum_of_squared_error,
}
```

In [6]:
```
class Layer:
    def __init__(self, activation, input, output):
        self.activation_name = activation
        self.activation = activation_functions[activation]
        self.cost_function = error_functions[activation]
        self.activation_derivative = activation_functions_derivative[activation]
        self.W = np.random.randn(output, input)
        self.b = np.random.randn(output, 1)

        self.reset_delta(output)
        self.reset_delta_weight()
        self.reset_delta_bias()
        self.output = np.zeros(output)
        self.net = np.zeros(output)
```

```
    def set_delta(self, delta):
        self.delta = delta

    def set_weight(self, weight):
        self.W = weight

    def set_bias(self, bias):
        self.b = bias

    def add_delta_weight(self, delta):
        self.delta_weight += delta

    def add_delta_bias(self, delta_bias):
        self.delta_bias += delta_bias

    def reset_delta(self, output):
        self.delta = np.zeros(output)

    def reset_delta_weight(self):
        self.delta_weight = np.zeros(self.W.shape)

    def reset_delta_bias(self):
        self.delta_bias = np.zeros(self.b.shape)

    def set_output(self, output):
        self.output = output

    def set_net(self, net):
        self.net = net
```

In [7]:
```
class NeuralNetwork:
    def __init__(self, learning_rate=0.05, max_iter=500, error_threshold=0.01, batch_s
        # seeding for random number generation
        np.random.seed(1)
        self.layers = []
        self.depth = 0
        self.learning_rate = learning_rate
        self.max_iter = max_iter
        self.batch_size = batch_size
        self.verbose = verbose
        self.error_threshold = error_threshold

    def add(self, layer):
        self.layers.append(layer)
        self.depth += 1

    def set_params(self, **kwargs):
```

```python
        self.__dict__.update(kwargs)

    def init_layers(self, layer_description):
        for a in layer_description:
            layer = self.Layer(a.activation_type,
                               a.previous_neuron, a.current_neuron)
            self.add(layer)

    def load_model(self, filename):
        f = open(filename, "r")

        self.depth = int(f.readline())

        for i in range(self.depth):

            n_neuron = int(f.readline())
            activation_type = f.readline()[:-1]
            weight = []

            n_neuron_prev = -1
            for j in range(n_neuron):
                temp = list(map(float, f.readline().split()))
                weight.append(temp)
                if (n_neuron_prev == -1):
                    n_neuron_prev = len(temp)

            layer = Layer(activation_type, n_neuron_prev, n_neuron)
            layer.set_weight(np.array(weight))
            bias = np.array(
                list(map(lambda x: [float(x)],
                         f.readline().split())))
            layer.set_bias(bias)

            self.layers.append(layer)

    def save_model(self, filename):
        f = open(filename, "w")

        f.write(str(self.depth) + "\n")
        for layer in self.layers:
            n_neuron = len(layer.b)
            f.write(str(n_neuron) + "\n")
            f.write(layer.activation_name + "\n")
            for i in range(n_neuron):
                f.write(" ".join(list(map(str, layer.W[i]))) + "\n")
            f.write(" ".join(list(map(lambda x: str(x[0]), layer.b))) + "\n")

    def forward_propagate(self, x_inputs):
```

```
        a = np.array(x_inputs).T
        for layer in self.layers:
            z = np.dot(layer.W, a) + layer.b
            layer.set_net(z)
            a = layer.activation(z)
            layer.set_output(a)
        return a

    def backward_propagate(self, X_train, y_train, prediction):
        grad = {}

        num_layers = len(self.layers)

        for i in reversed(range(num_layers)):
            layer = self.layers[i]

            # if output layer
            if i == num_layers - 1:
                # use squared error derivative if not softmax
                if self.layers[-1].activation != 'softmax':
                    layer.delta = clip((prediction - y_train) * layer.activation_deriva
                else:
                    layer.delta = cross_entropy_derivative(prediction, y_train) * layer
            else:
                next_layer = self.layers[i + 1]
                error = np.dot(next_layer.W.T, next_layer.delta)
                layer.delta = clip(error * layer.activation_derivative(layer.net))

        for i in range(num_layers):
            layer = self.layers[i]
            input_activation = np.atleast_2d(X_train if i == 0 else self.layers[i - 1]
            grad["dW" + str(i)] = clip(np.dot(layer.delta, input_activation.T) * self.l
            grad["db" + str(i)] = clip(layer.delta * self.learning_rate)

        return grad

    def shuffle(self, x_train, y_train):
        sz = len(y_train)
        ids = [i for i in range(sz)]
        np.random.shuffle(ids)
        ret_x, ret_y = [], []
        for i in ids:
            ret_x.append(list(x_train[i]))
            ret_y.append(list(y_train[i]))
        return (ret_x), (ret_y)

    def split_batch(self, x_train, y_train):
        batches_x = []
```

```
        batches_y = []

        length = len(x_train)

        for i in range((length // self.batch_size)):
            x_batch = x_train[i * self.batch_size: (i + 1) * self.batch_size]
            y_batch = y_train[i * self.batch_size: (i + 1) * self.batch_size]
            batches_x.append(np.array(x_batch))
            batches_y.append(np.array(y_batch))
        if length % self.batch_size != 0:
            i = length // self.batch_size
            x_batch = x_train[i * self.batch_size:]
            y_batch = y_train[i * self.batch_size:]
            batches_x.append(np.array(x_batch))
            batches_y.append(np.array(y_batch))

        return (batches_x), (batches_y)

    def fit(self, x_train, y_train):
        for iteration in range(self.max_iter):

            x_train, y_train = self.shuffle(x_train, y_train)

            batches_x, batches_y = self.split_batch(x_train, y_train)

            cost_function = 0
            for i in range(len(batches_x)):
                x_input = batches_x[i]
                y_output = batches_y[i]

                prediction = self.forward_propagate(x_input)
                cost_function += self.layers[-1].cost_function(
                    prediction, y_output.T)
                gradients = self.backward_propagate(
                    x_input.T, y_output.T, prediction)

                # update delta phase
                for j, layer in enumerate(self.layers):
                    layer.add_delta_weight(gradients["dW" + str(j)])
                    grad_bias = gradients["db" + str(j)]
                    layer.add_delta_bias(np.sum(grad_bias, axis=1).reshape(len(grad_bia

                # update weights phase
                for j, layer in enumerate(self.layers):
                    layer.W += layer.delta_weight  # gradients["dW" + str(j)]
                    layer.b += layer.delta_bias  # gradients["db" + str(j)]

                for j, layer in enumerate(self.layers):
```

```
                    layer.reset_delta_weight()
                    layer.reset_delta_bias()

            cost_function /= len(x_train)
            if iteration % 50 == 0:
                print(f"Iteration {iteration}: ", cost_function)

            if cost_function < self.error_threshold:
                break

    def predict(self, x_test):
        prediction = self.forward_propagate(x_test)
        return prediction

    def __str__(self):
        index = 1
        res = ""
        for layer in self.layers:
            res += "{}-th layer\n".format(index)
            res += f"Activation: {layer.activation_name}\n"
            res += "Weight matrix:\n"
            res += layer.W.__str__() + "\n"
            res += "Bias\n"
            res += layer.b.__str__() + "\n\n"
            index += 1
        return res
```

In [8]:

```
from sklearn.datasets import load_iris
from sklearn.preprocessing import OneHotEncoder

enc = OneHotEncoder(handle_unknown='ignore')

data = load_iris()
X = data.data
y = data.target
y = y.reshape(-1,1)
enc.fit(y)
y = enc.transform(y).toarray()
```

In [9]:

```
#train-test-split 90%-10%
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=

# print(X_train.shape)
# print(X_test.shape)
# print(y_train.shape)
# print(y_test.shape)
```

```python
    # create model
    model = NeuralNetwork(learning_rate=0.001, max_iter=2000, verbose=False)
    model.add(Layer("relu", 4, 10))
    model.add(Layer("relu", 10, 10))
    model.add(Layer("linear", 10, 5))
    model.add(Layer("sigmoid", 5, 3))
    model.fit(X_train, y_train)

    #bikin confusion matrix dan metric dari training ini:
    prediction = model.predict(X_test)
    label_pred = []
    for i in range(prediction.shape[1]):
        label_pred.append(np.argmax(prediction[:, i]))
    y_test_label = []
    for i in range(y_test.shape[0]):
        y_test_label.append(np.argmax(y_test[i, :]))
    metrics = Metrics(y_test_label, label_pred)
    print("ACCURACY SLURRR: ", metrics.all_accuracy())
    print("PRECISION SLURRR: ", metrics.all_precision())
    print("RECALL SLURRR: ", metrics.all_recall())
    print("F1 SLURRR: ", metrics.all_f1_score())
    print("CONFUSION MATRIXX: ")
    print(confusion_matrix(y_test_label, label_pred))

    print("--------------------------------------")
```

```
Iteration 0:  0.3183902492549609
Iteration 50:  0.04764504032348603
Iteration 100:  0.03805905068935214
Iteration 150:  0.020535108455993445
Iteration 200:  0.03784032822775462
Iteration 250:  0.024026457059622326
Iteration 300:  0.026548040779272957
Iteration 350:  0.02422937261430657
Iteration 400:  0.02995375376641365
Iteration 450:  0.03782240507182985
Iteration 500:  0.015999987520767332
Iteration 550:  0.02800142893615686
Iteration 600:  0.025151904103365683
ACCURACY SLURRR:  1.0
PRECISION SLURRR:  1.0
RECALL SLURRR:  1.0
F1 SLURRR:  1.0
CONFUSION MATRIXX:
[[6, 0, 0], [0, 5, 0], [0, 0, 4]]
--------------------------------------
```

```
In [14]: #10-fold cross validation
         from sklearn.model_selection import KFold

         k_fold = KFold(n_splits=10)
         print(k_fold)
         scores = []
         for train_index, test_index in k_fold.split(X):
             print("TRAIN DATA INDEX")
             print(train_index)
             print("TEST DATA INDEX")
             print(test_index)
             X_train, X_test = X[train_index], X[test_index]
             y_train, y_test = y[train_index], y[test_index]
             model_tmp = NeuralNetwork(learning_rate=0.001, max_iter=1000, verbose=False)
             model_tmp.add(Layer("relu", 4, 10))
             model_tmp.add(Layer("relu", 10, 10))
             model_tmp.add(Layer("linear", 10, 5))
             model_tmp.add(Layer("sigmoid", 5, 3))
             model_tmp.fit(X_train, y_train)
             prediction = model_tmp.predict(X_test)
             label_pred = []
             for i in range(prediction.shape[1]):
                 label_pred.append(np.argmax(prediction[:, i]))
             y_test_label = []
             for i in range(y_test.shape[0]):
                 y_test_label.append(np.argmax(y_test[i, :]))
             metrics = Metrics(y_test_label, label_pred)
             metrics.report()
```

```
KFold(n_splits=10, random_state=None, shuffle=False)
TRAIN DATA INDEX
[ 15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32
  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86
  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104
 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122
 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140
 141 142 143 144 145 146 147 148 149]
TEST DATA INDEX
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
Iteration 0:  0.39937328880962825
Iteration 50:  0.04978689816065027
Iteration 100:  0.046302060608163095
Iteration 150:  0.037639947227490186
Iteration 200:  0.03416496216213545
Iteration 250:  0.04312085493846274
Iteration 300:  0.03329399228431469
```

```
Iteration 350:  0.03093125061546607
Iteration 400:  0.014304452140782633
Iteration 450:  0.029751889241869937
Iteration 500:  0.021201205326134735
Iteration 550:  0.039217012901519105
Iteration 600:  0.025280085328910466
Iteration 650:  0.0335931204038605
Iteration 700:  0.024661866496472253
Iteration 750:  0.04610135821713308
Iteration 800:  0.026617263125413067
   accuracy  precision  recall   f1
0       1.0        1.0     1.0  1.0
overall accuracy: 1.000
overall precision: 1.000
overall recall: 1.000
overall f1_score: 1.000
TRAIN DATA INDEX
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  30  31  32
  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86
  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104
 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122
 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140
 141 142 143 144 145 146 147 148 149]
TEST DATA INDEX
[15 16 17 18 19 20 21 22 23 24 25 26 27 28 29]
Iteration 0:  0.3906122588872123
Iteration 50:  0.040330498972528175
Iteration 100:  0.044265798311197424
Iteration 150:  0.04761492490747132
Iteration 200:  0.04008368427565766
Iteration 250:  0.042881030116223316
Iteration 300:  0.01802330962363282
Iteration 350:  0.035884799111425024
Iteration 400:  0.013495687794104781
Iteration 450:  0.030725500672805488
Iteration 500:  0.024275915683943446
Iteration 550:  0.03772128805073547
Iteration 600:  0.02961105748346587
Iteration 650:  0.03065620876581967
Iteration 700:  0.03490447961261477
Iteration 750:  0.03740345364270223
Iteration 800:  0.030301807033795536
   accuracy  precision  recall   f1
0       1.0        1.0     1.0  1.0
overall accuracy: 1.000
overall precision: 1.000
```

```
overall recall: 1.000
overall f1_score: 1.000
TRAIN DATA INDEX
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  45  46  47  48  49  50
  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86
  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104
 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122
 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140
 141 142 143 144 145 146 147 148 149]
TEST DATA INDEX
[30 31 32 33 34 35 36 37 38 39 40 41 42 43 44]
Iteration 0:  0.4171566821647122
Iteration 50:  0.05121232113920098
Iteration 100:  0.04832819610607747
Iteration 150:  0.03932093651271849
Iteration 200:  0.037864136865288625
Iteration 250:  0.04205261687157766
Iteration 300:  0.0260725463678152
Iteration 350:  0.031901396754422794
Iteration 400:  0.014496844039040066
Iteration 450:  0.03231508969632846
Iteration 500:  0.01978443166479547
Iteration 550:  0.0336957246221723
Iteration 600:  0.026642502208786164
Iteration 650:  0.03401433752221281
Iteration 700:  0.04730733574523879
Iteration 750:  0.038297356028911926
Iteration 800:  0.03167615270351398
   accuracy  precision  recall   f1
0       1.0        1.0     1.0  1.0
overall accuracy: 1.000
overall precision: 1.000
overall recall: 1.000
overall f1_score: 1.000
TRAIN DATA INDEX
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  60  61  62  63  64  65  66  67  68
  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86
  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104
 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122
 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140
 141 142 143 144 145 146 147 148 149]
TEST DATA INDEX
[45 46 47 48 49 50 51 52 53 54 55 56 57 58 59]
Iteration 0:  0.3448949921806437
```

```
Iteration 50:  0.048575447258842726
Iteration 100:  0.04561748098143514
Iteration 150:  0.03880669683318288
Iteration 200:  0.04189337252801134
Iteration 250:  0.04251926218956849
Iteration 300:  0.026009144990873336
Iteration 350:  0.036587074617222076
Iteration 400:  0.014079026714813469
Iteration 450:  0.033456479329546956
Iteration 500:  0.02307831874421066
Iteration 550:  0.038271425585731884
Iteration 600:  0.02878933652485788
Iteration 650:  0.03147011551925716
Iteration 700:  0.024868763593859873
Iteration 750:  0.043971287664432875
Iteration 800:  0.03013056461758563
   accuracy  precision  recall   f1
0       1.0        1.0     1.0  1.0
1       1.0        1.0     1.0  1.0
overall accuracy: 1.000
overall precision: 1.000
overall recall: 1.000
overall f1_score: 1.000
TRAIN DATA INDEX
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  75  76  77  78  79  80  81  82  83  84  85  86
  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104
 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122
 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140
 141 142 143 144 145 146 147 148 149]
TEST DATA INDEX
[60 61 62 63 64 65 66 67 68 69 70 71 72 73 74]
Iteration 0:  0.3303470341475604
Iteration 50:  0.04490973512656807
   accuracy  precision  recall     f1
1     0.933        1.0   0.933  0.966
2     0.933        0.0   0.000  0.000
overall accuracy: 0.933
overall precision: 0.500
overall recall: 0.467
overall f1_score: 0.483
TRAIN DATA INDEX
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
```

```
  72  73  74  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104
 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122
 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140
 141 142 143 144 145 146 147 148 149]
TEST DATA INDEX
[75 76 77 78 79 80 81 82 83 84 85 86 87 88 89]
Iteration 0:  0.34615210216737174
Iteration 50:  0.03862184379370044
   accuracy  precision  recall     f1
1       0.8        1.0     0.8  0.889
2       0.8        0.0     0.0  0.000
overall accuracy: 0.800
overall precision: 0.500
overall recall: 0.400
overall f1_score: 0.444
TRAIN DATA INDEX
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122
 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140
 141 142 143 144 145 146 147 148 149]
TEST DATA INDEX
[ 90  91  92  93  94  95  96  97  98  99 100 101 102 103 104]
Iteration 0:  0.34249223341965745
Iteration 50:  0.04407883744396616
Iteration 100:  0.035251149574306265
Iteration 150:  0.03252114344034898
Iteration 200:  0.03289340710227642
Iteration 250:  0.03656951104281978
Iteration 300:  0.026352208609194655
Iteration 350:  0.033658939017660504
Iteration 400:  0.03238711174262154
Iteration 450:  0.019992134207046826
Iteration 500:  0.025011538210929787
Iteration 550:  0.02344106821867751
Iteration 600:  0.02679920139042679
Iteration 650:  0.027429468616654483
Iteration 700:  0.014740286334608037
Iteration 750:  0.025197407007096224
Iteration 800:  0.037622560438355906
Iteration 850:  0.021052006821737534
Iteration 900:  0.025914507895461514
Iteration 950:  0.03271751004548546
   accuracy  precision  recall   f1
1       1.0        1.0     1.0  1.0
```

```
2       1.0        1.0     1.0  1.0
overall accuracy: 1.000
overall precision: 1.000
overall recall: 1.000
overall f1_score: 1.000
TRAIN DATA INDEX
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 120 121 122
 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140
 141 142 143 144 145 146 147 148 149]
TEST DATA INDEX
[105 106 107 108 109 110 111 112 113 114 115 116 117 118 119]
Iteration 0:  0.34273233128616604
Iteration 50:  0.03210394959736271
Iteration 100:  0.03617168908211247
Iteration 150:  0.025587564788053168
Iteration 200:  0.032702259961308947
Iteration 250:  0.03612408645466872
Iteration 300:  0.020956489714018534
Iteration 350:  0.02789720217492909
Iteration 400:  0.03262911422713517
Iteration 450:  0.022201179857161333
Iteration 500:  0.030214560535601773
Iteration 550:  0.023574650407387824
Iteration 600:  0.03319796112084177
Iteration 650:  0.025197663929691045
Iteration 700:  0.021627060702647763
Iteration 750:  0.02607886894287598
Iteration 800:  0.023519971831936205
Iteration 850:  0.020267720438139687
Iteration 900:  0.02264894685418159
Iteration 950:  0.02064012475387959
   accuracy  precision  recall   f1
2       1.0        1.0     1.0  1.0
overall accuracy: 1.000
overall precision: 1.000
overall recall: 1.000
overall f1_score: 1.000
TRAIN DATA INDEX
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
```

```
  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
 108 109 110 111 112 113 114 115 116 117 118 119 135 136 137 138 139 140
 141 142 143 144 145 146 147 148 149]
TEST DATA INDEX
[120 121 122 123 124 125 126 127 128 129 130 131 132 133 134]
Iteration 0:  0.3430396268693301
Iteration 50:  0.020770697177730887
Iteration 100:  0.01662455867042806
Iteration 150:  0.009862355883482698
   accuracy  precision  recall     f1
1     0.467        0.0   0.000  0.000
2     0.467        1.0   0.467  0.636
overall accuracy: 0.467
overall precision: 0.500
overall recall: 0.233
overall f1_score: 0.318
TRAIN DATA INDEX
[  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125
 126 127 128 129 130 131 132 133 134]
TEST DATA INDEX
[135 136 137 138 139 140 141 142 143 144 145 146 147 148 149]
Iteration 0:  0.3381399813716052
Iteration 50:  0.030400819974985293
Iteration 100:  0.02191116753617455
Iteration 150:  0.03078450054761991
Iteration 200:  0.025219806448061047
Iteration 250:  0.03177964842412813
Iteration 300:  0.021684666575930916
Iteration 350:  0.01600712722919134
Iteration 400:  0.022201802141446788
Iteration 450:  0.021056863404572662
Iteration 500:  0.01625162528651977
Iteration 550:  0.0245440473018561
Iteration 600:  0.015486039849931318
Iteration 650:  0.023483799797568046
Iteration 700:  0.022403023733162112
Iteration 750:  0.022751337962156736
Iteration 800:  0.01703452544700577
Iteration 850:  0.02686978855382639
Iteration 900:  0.023782512512942378
Iteration 950:  0.027000018532561898
   accuracy  precision  recall   f1
```

```
2       1.0        1.0     1.0  1.0
overall accuracy: 1.000
overall precision: 1.000
overall recall: 1.000
overall f1_score: 1.000
```

In [15]:
```python
# Simpan model
model_filename = "model.txt"
model.save_model(model_filename)

# Load model yang baru disimpan
loaded_model = NeuralNetwork(learning_rate=0.001, max_iter=2000, verbose=False)
loaded_model.load_model(model_filename)

# Bikin instance data baru, predict pake model yg di-load
instances = [
  [6.9, 3.2, 4.7, 1.4], # versicolor (1)
  [5.0, 3.5, 1.4, 0.2], # setosa (0)
  [6.3, 3.3, 6.0, 2.4], # virginica (2)
]
result = loaded_model.forward_propagate(instances)
print(list(map(np.argmax, result)))
```

```
[1, 0, 2]
```

In [16]:
```python
#Analisis dari 2 hal ini:
# 2. Lakukan pengujian dengan membandingkan confusion matrix dan perhitungan kinerja
# dari sklearn.
# 3. Lakukan pembelajaran FFNN untuk dataset iris dengan skema split train 90% dan
# test 10%, dan menampilkan kinerja serta confusion matrixnya.
# berdasarkan hasil yang sudah kami jalankan untuk skema split train 90% dan
# test 10%, model yang didapatkan sudah cukup akurat dan ini dapat dilihat
# dari hasil accuracy, precision, recall, dan F1nya. begitu juga confusion
# matrixnya yang tidak ada persebaran selain di cell yang tepat prediksi
# dan aslinya.
```