In [162]:

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

In [163]:

```
# Load data
iris = load_iris()
datasets = train_test_split(iris.data, iris.target,test_size=0.2)
train_data, test_data, train_labels, test_labels = datasets

# Normalize
scaler = StandardScaler()
scaler.fit(train_data)

train_data = scaler.transform(train_data)
test_data = scaler.transform(test_data)

clf = MLPClassifier(solver='sgd', hidden_layer_sizes=(3, 2, 5), max_iter=100, batch_size=2)
clf.fit(train_data, train_labels)
clf.score(train_data, train_labels)

# Bobot layer
i = 0
for w in clf.coefs_:
    print(f"Layer{i}")
    print(w)
    i+=1

print("Jumlah iterasi")
print(clf.n_iter_)
```

```
Layer0
[[ 0.56412144 -0.50199081 -0.05985622]
 [ 0.5888403  -0.3877157   0.36730191]
 [ 0.49647127  1.12292054  0.85033458]
 [ 0.30465167  0.09938027  0.46861153]]
Layer1
[[-0.66813397 -0.8335493 ]
 [-0.73122328 -0.3439389 ]
 [-0.82494887 -0.15708275]]
Layer2
[[-0.51071857  0.79342412 -0.17041735  1.38541966 -0.50092169]
 [ 0.63231207 -0.47530754  0.58464833 -0.8890474   0.55270545]]
Layer3
[[-7.81591590e-01  4.39672763e-01 -4.03526617e-01]
 [ 3.87265158e-01 -3.67183768e-01 -7.11383716e-01]
 [-8.33120691e-04  7.74381057e-01 -2.08942362e-01]
 [ 9.76188684e-01 -4.90602202e-01 -5.95119233e-01]
 [-3.94498660e-01 -1.61040155e-01  7.15696440e-01]]
Jumlah iterasi
12

C:\Users\ACER\AppData\Local\Programs\Python\Python310\lib\site-packages\skle
arn\neural_network\_multilayer_perceptron.py:699: UserWarning: Training inte
rrupted by user.
  warnings.warn("Training interrupted by user.")
```

In [ ]:

```
import numpy as np
from random import seed
from random import random, uniform
from sklearn import datasets
from sklearn.datasets import load_iris
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler
import graphviz
import copy
from random import randint
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score

class Data:
    def __init__(self):
        self.data = []
        self.target = []
        self.target_names = []

def linear(x):
    return x


def sigmoid(x):
    return 1/(1+np.exp(-x))


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


def softmax(x):
    net_h = np.array(x)
    numerator = np.exp(net_h)
    denominator = np.sum(np.exp(x))
    softmax_output = numerator / denominator
    return softmax_output

def linear_derivative(x):
    return 1

def sigmoid_derivative(x):
    s = 1 / (1 + np.exp(-x))
    return x * (1 - x)

# todo
def relu_derivative(x):
    return [0 if (el < 0) else 1 for el in x]

# todo
def softmax_derivative(x):
    return [-(1-el) for el in x]

def split_dataset_90_10(dataset):
    data_length = len(dataset.target)
```

```
    n_train = round(data_length * 9/10)
    n_test = data_length - n_train

    train = Data()
    test = Data()
    train.target_names = dataset.target_names
    test.target_names = dataset.target_names

    test_idx = []
    while len(test_idx) < n_test:
        idx = randint(0, 149)
        try:
            test_idx.index(idx)
        except:
            test_idx.append(idx)
            test.data.append(dataset.data[idx])
            test.target.append(dataset.target[idx])

    for i in range(data_length):
        try:
            test_idx.index(i)
        except:
            train.data.append(dataset.data[i])
            train.target.append(dataset.target[i])
    return train, test

def confusionMatrix(y_test, y_pred):
    x = len(set(y_test))
    confusion_matrix = [[0 for i in range(x)] for j in range(x)]
    for i in range(len(y_test)):
        confusion_matrix[y_test[i]][y_pred[i]] += 1
    return np.array(confusion_matrix)

def accuracy(confusion_matrix):
    np.seterr(invalid='ignore')
    return np.nan_to_num(np.sum(np.diag(confusion_matrix)) / np.sum(confusion_matrix))

def precision(confusion_matrix):
    np.seterr(invalid='ignore')
    return np.nan_to_num(np.diag(confusion_matrix) / np.sum(confusion_matrix, axis=0))

def recall(confusion_matrix):
    np.seterr(invalid='ignore')
    return np.nan_to_num(np.diag(confusion_matrix) / np.sum(confusion_matrix, axis=1))

def f1(confusion_matrix):
    np.seterr(invalid='ignore')
    return np.nan_to_num(2 * precision(confusion_matrix) * recall(confusion_matrix) / (prec

def summary(confusion_matrix):
    print("Confusion Matrix:")
    print(confusion_matrix)
    print("Accuracy:", accuracy(confusion_matrix))
    print("Precision:", precision(confusion_matrix))
    print("Recall:", recall(confusion_matrix))
    print("F1:", f1(confusion_matrix))

class Layer:
    def __init__(self, n_input, n_nodes):
        self.weights = []
        self.n_input = n_input
```

```
        self.n_nodes = n_nodes
        self.activations = ""
        self.input = []
        self.output = []
        self.error = []
        self.updated_weights = []

    def update_weights_back_propagation(self):
        self.weights = self.updated_weights.copy()
        self.updated_weights = []

class NeuralNetwork:
    def __init__(self, n_layers, n_neuron=[], activation=[],
                 learning_rate=0.1, err_threshold=0.01,
                 max_iter=100, batch_size=2, dataset=load_iris(),
                 n_input=4, n_output=3):
        # Load iris dataset
        self.dataset = dataset  # dataset
        self.input = dataset.data  # input
        self.target = dataset.target  # target
        self.target_names = dataset.target_names  # target class name
        self.n_attr = n_input  # n input attribute

        # Neural network
        self.n_layers = n_layers  # how many hidden layers
        self.n_neuron = n_neuron  # how many neuron for each hidden layer
        self.activation = activation  # activation for each layer

        self.learning_rate = learning_rate
        self.err_threshold = err_threshold
        self.max_iter = max_iter
        self.batch_size = batch_size
        self.layers = []
        self.bias = 1
        self.output = []  # final output from forward propagate
        self.mse = 999

        # Back prop
        self.error_hidden_value = 0
        self.updated_weights = []
        self.error = 999  # current error?
        self.weights = []  # last updated weight
        self.predict = []

        for i in range(n_layers):
            # n input = n neuron in layer
            if i == 0:
                layer = Layer(self.n_attr+1, n_neuron[i])
                layer.weights = [
                    [uniform(-0.5, 0.5) for i in range(n_neuron[i])] for j in range(self.n_
            else:
                layer = Layer(n_neuron[i-1]+1, n_neuron[i])
                layer.weights = [
                    [uniform(-0.5, 0.5) for i in range(n_neuron[i])] for j in range(n_neuro
            # initalize weight
            layer.activations = activation[i]

            self.layers.append(layer)

        # add last layer, last hidden to output
        layer = Layer(n_neuron[-1] + 1, n_output)
```

```
        layer.weights = [[uniform(-0.5, 0.5) for i in range(n_output)]
                         for j in range(n_neuron[-1] + 1)]
        layer.activations = "softmax"
        self.layers.append(layer)

        # with open(file_name, "r") as f:
        #     line = f.readline().split()
        #     self.n_layers = len(line) - 1 # how many hidden layers + ouput

        #     for i in range(self.n_layers + 1):
        #         if i == 0:
        #             self.layers.append(Layer(self.n_attr, int(line[i])))
        #         else:
        #             self.layers.append(Layer(int(line[i-1]), int(line[i])))

        #     for i in range(self.n_layers + 1):
        #         f.readline()
        #         for j in range(self.layers[i].n_input + 2):
        #             weight = []
        #             line = f.readline().strip(" \n").split(" ")
        #             for k in range(len(line)):
        #                 if (j == 0):
        #                     self.layers[i].activations = str(line[k])
        #                 else:
        #                     weight.append(float(line[k]))
        #             if j!=0:
        #                 self.layers[i].weights.append(weight)

    def load_model(self, file_name):
        with open(file_name, "r") as f:
            line = f.readline().split()
            self.n_layers = len(line) - 1 # how many hidden layers + ouput

            for i in range(self.n_layers + 1):
                if i == 0:
                    self.layers.append(Layer(self.n_attr, int(line[i])))
                else:
                    self.layers.append(Layer(int(line[i-1]), int(line[i])))

            for i in range(self.n_layers + 1):
                f.readline()
                self.layers[i].weights = []
                for j in range(self.layers[i].n_input + 1):
                    weight = []
                    line = f.readline().strip(" \n").split(" ")

                    if len(line[0]) != 0:
                        for k in range(len(line)):
                            if (j == 0):
                                self.layers[i].activations = str(line[k])
                            else:
                                weight.append(float(line[k]))
                        if j!=0:
                            self.layers[i].weights.append(weight)

    def save_model(self, filename):
        new_file = []
        with open(filename) as file:
            lines = [line.rstrip().split() for line in file]

            n_neuron = []
```

```
            for i in range(self.n_layers + 1):
                n_neuron.append(self.layers[i].n_nodes)

            new_file.append(n_neuron)
            new_file.append('')

            for i in range(self.n_layers + 1):
                new_file.append([self.layers[i].activations])
                for new_weight in self.layers[i].weights:
                    new_file.append(new_weight)
                if i < self.n_layers:
                    new_file.append('')

            for line in range(len(new_file)):
                str_line = ''
                for i in range(len(new_file[line])):
                    str_line += str(new_file[line][i])
                    if i < len(new_file[line]) - 1:
                        str_line += ' '
                new_file[line] = str_line

        new_filename = filename.split(".")[0] + "_updated_weights"
        with open(new_filename + '.txt', 'w') as f:
            for line in range(len(new_file)):
                f.write(new_file[line])
                if line < len(new_file) - 1:
                    f.write('\n')

    def forward_propagation(self, type):
        for i in range(self.n_layers + 1):
            self.layers[i].input = []
        # The first input layer
        if type == "train":
            self.layers[0].input = self.input
        elif type == "predict":
            self.layers[0].input = self.predict

        # All hidden layers
        for i in range(self.n_layers + 1):
            # add bias
            bias_input = self.bias
            if (i == 0):  # if first hidden layer, convert to array from ndarray and then a
                temp_input = []
                for j in range(len(self.layers[i].input)):
                    input_row = []
                    for k in range(len(self.layers[i].input[j])):
                        input_row.append(self.layers[i].input[j][k])
                    input_row.append(bias_input)
                    temp_input.append(input_row)
                self.layers[i].input = temp_input
            else:  # if not first layer the immediately add the bias in the last index
                for j in range(len(self.layers[i].input)):
                    self.layers[i].input[j].append(bias_input)

            # calculate sigma
            self.layers[i].output = np.dot(
                self.layers[i].input, self.layers[i].weights)

            # activation function
            for j in range(len(self.layers[i].output)):
                input_next_layer = []  # temporary list to store the next layer's input
```

```
                for k in range(len(self.layers[i].output[j])):
                    x = self.layers[i].output[j][k]
                    result = 0
                    if i != self.n_layers:
                        if (self.layers[i].activations.lower() == "linear"):
                            result = format(linear(x))
                        elif (self.layers[i].activations.lower() == "sigmoid"):
                            result = format(sigmoid(x))
                        elif (self.layers[i].activations.lower() == "relu"):
                            result = format(relu(x))
                        # elif (self.layers[i].activations.lower() == "softmax"):
                        #     result = format(softmax(x))
                        else:  # if activation is not linier, relu, sigmoid, or softmax
                            print(
                                f"{self.layers[i].activations}: Invalid activation method!"
                            return
                    else:
                        result = format(softmax(x))

                    # append output, actually layers[i].output == layers[i+1].input
                    self.layers[i].output[j][k] = result
                    # append input for next layer in temporary list (input_next_layer)
                    input_next_layer.append(float(result))

                if (i < self.n_layers):  # if there is still the next layer
                    # append input for next layer in layers[i+1].input
                    self.layers[i+1].input.append(input_next_layer)

        # output in the last layer
        self.output = self.layers[-1].output.copy()

    # todo
    def error_output(self):
        # get output layer
        mse_sum = 0
        total = 0
        output_layer = self.layers[self.n_layers]
        activation_rule = output_layer.activations.lower()
        output_layer.error = output_layer.output.copy()
        for i in range(len(self.input)):
            expected_target = []
            if (self.target[i] == 0):
                expected_target = [1, 0, 0]
            if (self.target[i] == 1):
                expected_target = [0, 1, 0]
            if (self.target[i] == 2):
                expected_target = [0, 0, 1]
            for j in range(3):
                output_layer.error[i][j] = expected_target[j] - output_layer.error[i][j]
                mse_sum += pow(expected_target[j] - output_layer.error[i][j], 2)
                total += 1
        self.mse = (1/total) * mse_sum
        self.error = np.mean(output_layer.error)

        # calculate error per output node
        for i in range(len(output_layer.error)):
            if (activation_rule == "sigmoid"):
                output_layer.error[i] *= sigmoid_derivative(
                    output_layer.output[i])
            elif (activation_rule == "relu"):
                output_layer.error[i] *= relu_derivative(
```

```
                    output_layer.output[i])
            elif (activation_rule == "linear"):
                output_layer.error[i] *= linear_derivative(
                    output_layer.output[i])
            elif (activation_rule == "softmax"):
                output_layer.error[i] *= softmax_derivative(
                    output_layer.output[i])

        return

    # todo
    def error_hidden(self):
        #menghitung delta net j = delta k * weight j k
        delta_net_j = 0

        for i in range(self.n_layers-1, -1, -1):
            delta_net_j_array = []
            # print( self.layers[i].error, "\n", self.layers[i+1].weights[i])
            for j in range(len(self.layers[i+1].error)):
                delta_net_j_data = []
                for k in range(len(self.layers[i+1].weights)-1):
                    # print(self.layers[i+1].error)
                    delta_net_j = np.dot(self.layers[i+1].error[j], np.transpose(self.layer
                    delta_net_j_data.append(delta_net_j)


                delta_net_j_array.append(delta_net_j_data)

            self.layers[i].error = self.layers[i].output.copy()

            #menghitung error di hidden layer
            for j in range(len(self.layers[i].error)):
                if (self.layers[i].activations == "sigmoid"):
                    self.layers[i].error[j] = np.dot(sigmoid_derivative(
                        self.layers[i].output[j]),  delta_net_j_array[j])
                elif (self.layers[i].activations == "relu"):
                    self.layers[i].error[j] = np.dot(relu_derivative(
                        self.layers[i].output[j]), delta_net_j_array[j])
                elif (self.layers[i].activations == "linear"):
                    self.layers[i].error[j] = np.dot(linear_derivative(
                        self.layers[i].output[j]), delta_net_j_array[j])

        return

    # todo
    def update_weights(self, row_input, old_weight, error_term):
        new_weight_list = []

        # new weight for hidden layers
        for i in range(len(row_input)):
            new_weight = old_weight[i] - self.learning_rate * error_term * row_input[i]
            new_weight_list.append(new_weight)

        return new_weight_list

    # todo
    def back_propagation(self):
        # output layer
        self.error_output()
        output_layer = self.layers[self.n_layers]
        self.error = np.mean(output_layer.error)
```

```
        updated_weights_temp = []

        for i in range(len(output_layer.output)):
            updated_weights_temp.append(self.update_weights(
                output_layer.input[i], output_layer.weights, output_layer.error[i]))
        output_layer.updated_weights = output_layer.weights.copy()
        output_layer.updated_weights = np.mean(
            updated_weights_temp, axis=0)

        self.error_hidden()
        for i in range(self.n_layers-1, -1 ,-1):
            self.error_hidden_value = np.mean(self.layers[i].error)
            updated_weights_temp_hidden = []
            for j in range(len(self.layers[i].input)):
                updated_weights_temp_hidden.append(self.update_weights(self.layers[i].input
            self.layers[i].updated_weights = self.layers[i].weights.copy()
            self.layers[i].updated_weights = np.mean(updated_weights_temp_hidden, axis=0)

        # update all weights
        for layer in self.layers:
            layer.update_weights_back_propagation()
        return

    # todo
    def train(self):
        it = 0
        input_temp = copy.copy(self.input)

        while ((it < self.max_iter) and (self.err_threshold < self.mse)):
            output = []
            error = []
            for i in range(int(len(self.input)/self.batch_size)):
                idx = i * self.batch_size
                self.input = input_temp[idx:idx+self.batch_size]
                self.forward_propagation(type="train")
                self.back_propagation()
                output.append(self.layers[self.n_layers].output)
                error.append(self.layers[self.n_layers].error)
            it += 1
            self.error = self.error

        return

    def set_predict(self, input):
        self.predict = input

    def prediction(self, test_target):
        self.forward_propagation(type="predict")
        self.convert_output_to_class(test_target)

        return self.output

    def prediction2(self):
        self.forward_propagation(type="predict")
        self.convert_output_to_class_2()
        return self.output

    def check_sanity(self):
        for layer in self.layers:
            print(layer.weights)
```

```
    def convert_output_to_class(self, test_target):
        self.output_predict = test_target.copy()
        for i in range(len(self.output)):
            if (self.output[i][0] > self.output[i][1]):
                self.output_predict[i] = 0 if (self.output[i][0] > self.output[i][2]) else
            elif (self.output[i][0] < self.output[i][1]):
                self.output_predict[i] = 1 if (self.output[i][1] > self.output[i][2]) else

    def convert_output_to_class_2(self):
        self.output_predict = self.target.copy()
        for i in range(len(self.output)):
            if (self.output[i][0] > self.output[i][1]):
                self.output_predict[i] = 0 if (self.output[i][0] > self.output[i][2]) else
            elif (self.output[i][0] < self.output[i][1]):
                self.output_predict[i] = 1 if (self.output[i][1] > self.output[i][2]) else

    def cross_validate(self):
        # shuffle dataset
        label = np.array(self.dataset.data)
        target = np.array(self.dataset.target)

        indices = np.arange(label.shape[0])
        np.random.shuffle(indices)

        label = label[indices]
        target = target[indices]

        # split into 10
        n = len(self.dataset.target)
        j = int(np.ceil(n / 10))

        total_mse = 0
        acc_score = 0
        prec_score = 0
        f1_score = 0
        rec_score = 0

        for it in range(10):
            data_train_label = copy.copy(label)
            data_train_target = copy.copy(target)

            data_train_label = np.concatenate((data_train_label[0:it*j], data_train_label[i
            data_train_target = np.concatenate((data_train_target[0:it*j], data_train_targe

            self.input = data_train_label
            self.target = data_train_target

            data_test_label = label[it*j:it*j+j]
            data_test_target = target[it*j:it*j+j]
            self.predict = data_test_label

            # train and predict
            self.train()

            # calculate error
            pred = self.prediction2()
            self.convert_output_to_class_2()
            expec = []

            # transform to [x,x,x]
            for i in range(len(self.output)):
```

```
                expected_target = []
                # print(data_test_label[i])
                if (data_test_target[i] == 0):
                    expected_target = [1, 0, 0]
                if (data_test_target[i] == 1):
                    expected_target = [0, 1, 0]
                if (data_test_target[i] == 2):
                    expected_target = [0, 0, 1]
                expec.append(expected_target)

            # calculate the MSE
            pred = np.concatenate(pred).ravel()
            expec = np.concatenate(expec).ravel()

            # calculate confusion matrix
            confusion_matrix = confusionMatrix(data_test_target, self.output_predict)
            acc_score += accuracy(confusion_matrix)
            f1_score += f1(confusion_matrix)
            rec_score += recall(confusion_matrix)
            prec_score += precision(confusion_matrix)

            sum_mse_cv = 0

            for i in range(len(pred)):
                sum_mse_cv += pow(pred[i] - expec[i], 2)

            sum_mse_cv = float(sum_mse_cv/len(pred))
            total_mse += sum_mse_cv

        mse_cv = float(total_mse / 10)
        print(f"MSE Score: {1 - mse_cv}")
        print(f"Average accurarcy: {acc_score/10}")
        print(f"Average precision: {prec_score/10}")
        print(f"Average F1: {f1_score/10}")
        print(f"Average recall: {rec_score/10}")



    def draw_model(self):
        f = graphviz.Digraph('Feed Forward Neural Network', filename="model")
        f.attr('node', shape='circle', width='1.0')
        f.edge_attr.update(arrowhead='vee', arrowsize='2')

        for i in range(self.n_layers):
            if i == 0:
                for j in range(len(self.layers[i].weights)): #count weights
                    for k in range(len(self.layers[i].weights[j])): #output node
                        if j==0:
                            f.edge(f"bx{j}", f"h{i+1}_{k}",
                                f"{self.layers[i].weights[j][k]:.2f}")
                        else:
                            f.edge(f"x{j}", f"h{i+1}_{k}",
                                f"{self.layers[i].weights[j][k]:.2f}")
            else:
                for j in range(len(self.layers[i].weights)): #count weights
                    for k in range(len(self.layers[i].weights[j])): #output node
                        if j==0:
                            f.edge(f"bh{i}", f"h{i+1}_{k}",
                                f"{self.layers[i].weights[j][k]:.2f}")
                        else:
                            f.edge(f"h{i}_{j-1}", f"h{i+1}_{k}",
```

```
                                f"{self.layers[i].weights[j][k]:.2f}")

        print(f.source)
        f.render(directory='model').replace('\\', '/')

    def testForward(self):
        self.prediction_forward()
    def set_input(self, inp):
        self.input = inp
    def set_target(self, target):
        self.target = target

    def predict_new_data(self, data):
        self.forward_propagation()
```

In [ ]:

```
# Train model
dataset = load_iris()
```

In [ ]:

```
# UJI CONFUSION MATRIX MODEL IMPLEMENTASI
print("============ UJI IMPLEMENTASI MATRIX CONFUSSION =================")
nn = NeuralNetwork(n_layers=3, dataset=dataset, batch_size=50, n_neuron=[3, 2, 5], activati
nn.train()
nn.set_predict(dataset.data)
nn.prediction2()
summary(confusionMatrix(dataset.target, nn.output_predict))
```

```
============ UJI IMPLEMENTASI MATRIX CONFUSSION =================
Confusion Matrix:
[[50  0  0]
 [ 0 50  0]
 [ 0  0 50]]
Accuracy: 1.0
Precision: [1. 1. 1.]
Recall: [1. 1. 1.]
F1: [1. 1. 1.]
```

In [ ]:

```
# UJI CONFUSION MATRIX MODEL SKLEARN
print("============ UJI SKLEARN MATRIX CONFUSION =================")
# Normalize
scaler = StandardScaler()
scaler.fit(dataset.data)

train_data = scaler.transform(dataset.data)

clf = MLPClassifier(solver='adam', hidden_layer_sizes=(3, 2, 5), max_iter=100, batch_size=5
clf.fit(train_data, dataset.target)

print(confusion_matrix(dataset.target, clf.predict(train_data)))
print(f"Accuracy: {accuracy_score(dataset.target, clf.predict(train_data), normalize=False)
print(f"Precision: {precision_score(dataset.target, clf.predict(train_data), average=None)}
print(f"Recall: {recall_score(dataset.target, clf.predict(train_data), average=None)}")
print(f"F1: {f1_score(dataset.target, clf.predict(train_data), average=None)}")
```

```
============ UJI SKLEARN MATRIX CONFUSION =================
[[50  0  0]
 [50  0  0]
 [50  0  0]]
Accuracy: 0.3333333333333333
Precision: [0.33333333 0.         0.        ]
Recall: [1. 0. 0.]
F1: [0.5 0.  0. ]

C:\Users\ACER\AppData\Local\Programs\Python\Python310\lib\site-packages\skle
arn\neural_network\_multilayer_perceptron.py:692: ConvergenceWarning: Stocha
stic Optimizer: Maximum iterations (100) reached and the optimization hasn't
converged yet.
  warnings.warn(
C:\Users\ACER\AppData\Local\Programs\Python\Python310\lib\site-packages\skle
arn\metrics\_classification.py:1318: UndefinedMetricWarning: Precision is il
l-defined and being set to 0.0 in labels with no predicted samples. Use `zer
o_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

In [ ]:

```
# UJI SKEMA SPLIT TRAIN
print("============ UJI IMPLEMENTASI SPLIT TEST =================")
train, test = split_dataset_90_10(dataset)
nn = NeuralNetwork(n_layers=3, dataset=train, batch_size=50, n_neuron=[3, 2, 5], activation
nn.train()
nn.set_predict(test.data)
nn.prediction(test.target)
summary(confusionMatrix(test.target, nn.output_predict))

# sklearn train result
print("============ UJI SKLEARN SPLIT TEST =================")
scaler = StandardScaler()
scaler.fit(train.data)
train_data = scaler.transform(train.data)
scaler.fit(test.data)
test_data = scaler.transform(test.data)

clf = MLPClassifier(solver='adam', hidden_layer_sizes=(3, 2, 5), max_iter=1000, batch_size=
clf.fit(train_data, train.target)
target_pred = clf.predict(test_data)
print(confusion_matrix(test.target, target_pred))
print(accuracy_score(test.target, target_pred, normalize=False)/float(150))
print(precision_score(test.target, target_pred, average=None))
print(recall_score(test.target, target_pred, average=None))
print(f1_score(test.target, target_pred, average=None))
```

```
============ UJI IMPLEMENTASI SPLIT TEST =================
Confusion Matrix:
[[6 0 0]
 [0 5 0]
 [0 0 4]]
Accuracy: 1.0
Precision: [1. 1. 1.]
Recall: [1. 1. 1.]
F1: [1. 1. 1.]
============ UJI SKLEARN SPLIT TEST =================
[[6 0 0]
 [0 4 1]
 [0 0 4]]
0.09333333333333334
[1.  1.  0.8]
[1.  0.8 1. ]
[1.         0.88888889 0.88888889]
```

In [ ]:

```
# UJI SKEMA CROSS VALIDATION
print("============ UJI IMPLEMENTASI CROSS VALIDATION ================")
from sklearn.model_selection import KFold
nn = NeuralNetwork(n_layers=3, dataset=dataset, batch_size=50, n_neuron=[3, 2, 5], activati
nn.train()
# 10 fold implementation
nn.cross_validate()

# 10 Fold sklearn
print("============ UJI IMPLEMENTASI CROSS VALIDATION (SKLEARN) ================")

X = dataset.data
y = dataset.target

kf = KFold(n_splits=10)
for train_index, test_index in kf.split(X):
# ...     print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    nn.set_input(X_train)
    nn.set_target(y_train)
    nn.train()

    nn.set_predict(X_test)
    nn.prediction(y_test)

    summary(confusionMatrix(test.target, nn.output_predict))
```

```
============ UJI IMPLEMENTASI CROSS VALIDATION ================
MSE Score: 0.33333333333333326
Average accurarcy: 0.37333333333333335
Average precision: [0.325      0.36       0.41666667]
Average F1: [0.29795094 0.35833056 0.42374847]
Average recall: [0.2852381 0.375     0.4797619]
============ UJI IMPLEMENTASI CROSS VALIDATION (SKLEARN) ================
Confusion Matrix:
[[6 0 0]
 [5 0 0]
 [4 0 0]]
Accuracy: 0.4
Precision: [0.4 0.  0. ]
Recall: [1. 0. 0.]
F1: [0.57142857 0.         0.        ]
Confusion Matrix:
[[6 0 0]
 [5 0 0]
 [4 0 0]]
```

In [ ]:

```
print("============ UJI SKLEARN CROSS VALIDATION =================")
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from numpy import mean
from numpy import std

# Load data
iris = load_iris()
X = iris.data
y = iris.target

# Normalize
scaler = StandardScaler()
scaler.fit(train_data)


train_data = scaler.transform(X)

clf = MLPClassifier(solver='sgd', hidden_layer_sizes=(3, 2, 5), max_iter=1000, batch_size=2
clf.fit(X, y)
clf.score(X, y)

# 10 Cross Validation
kf = KFold(n_splits=10)

acc_scores = cross_val_score(clf, X, y, scoring="accuracy", cv=kf)

print('Accuracy: %.3f (%.3f)' % (mean(acc_scores), std(acc_scores)))
```

```
============ UJI SKLEARN CROSS VALIDATION =================
Accuracy: 0.673 (0.444)
```

In [ ]:

```
# SAVE MODEL
# FULL TRAINING
nn = NeuralNetwork(n_layers=3, dataset=dataset, batch_size=50, n_neuron=[3, 2, 5], activati
nn.train()
nn.save_model("model1.txt")
```

In [ ]:

```
# LOAD MODEL
nn.load_model("model1_updated_weights.txt")
```

In [ ]:

```
# PREDIKSI INSTANCE BARU
new_data = [1, 3, 4, 5]
nn.set_predict([new_data])
nn.prediction2()
```

Out[161]:

```
array([[1., 1., 1.]])
```

ANALISIS HASIL DARI 2 DAN 3

Hasil 2

Pada nomor 2, kami melakukan full training dari dataset menggunakan model hasil implementasi dan model dari library sklearn. Dengan menggunakan beberapa metric uji, diperoleh hasil berikut

Confusion Matrix:
[[50 0 0]
[0 50 0]
[0 0 50]]
Accuracy: 1.0
Precision: [1. 1. 1.]
Recall: [1. 1. 1.]
F1: [1. 1. 1.]

Dapat dilihat hampir seluruh metric memberi skor 1 yang memberikan tanda bahwa model memberikan bobot akhir yang dapat memprediksi masukan training ke kelas target secara benar. Namun setelah melakukan pembelajaran dengan MLP solver "adam", diperoleh hasil:

Confusion Matrix:
[[50 0 0]
[0 38 12]
[0 0 50]]
Accuracy: 0.92
Precision: [1. 1. 0.80645161]
Recall: [1. 0.76 1.]
F1: [1. 0.86363636 0.89285714]

Berdasarkan hasil berikut, dapat dilihat bahwa pembelajaran yang dilakukan sklearn belum memberikan bobot baru yang optimal untuk melakukan prediksi kelas target.

Hasil 3

Pada nomor 3, ========== UJI IMPLEMENTASI SPLIT TEST ========== Confusion Matrix: [[6 0 0] [0 5 0] [0 0 4]] Accuracy: 1.0 Precision: [1. 1. 1.] Recall: [1. 1. 1.] F1: [1. 1. 1.] ========== UJI SKLEARN SPLIT TEST ========== [[6 0 0] [0 4 1] [0 0 4]] 0.09333333333333334 [1. 1. 0.8] [1. 0.8 1.] [1. 0.88888889 0.88888889]