# IN3050/IN4050 Mandatory Assignment 2, 2022: Supervised Learning

## Rules

Before you begin the exercise, review the rules at this website: https://www.uio.no/english/studies/examinations/compulsory-activities/mn-ifi-mandatory.html , in particular the paragraph on cooperation. This is an individual assignment. You are not allowed to deliver together or copy/share source-code/answers with others. By submitting this assignment, you confirm that you are familiar with the rules and the consequences of breaking them.

## Delivery

**Deadline**: Friday, March 25, 2022, 23:59

Your submission should be delivered in Devilry. You may redeliver in Devilry before the deadline, but include all files in the last delivery, as only the last delivery will be read. You are recommended to upload preliminary versions hours (or days) before the final deadline.

## What to deliver?

You are recommended to solve the exercise in a Jupyter notebook, but you might solve it in a Python program if you prefer.

If you choose Jupyter, you should deliver the notebook. You should answer all questions and explain what you are doing in Markdown. Still, the code should be properly commented. The notebook should contain results of your runs. In addition, you should make a pdf of your solution which shows the results of the runs. (If you can't export: notebook -> latex -> pdf on your own machine, you may do this on the IFI linux machines.)

If you prefer not to use notebooks, you should deliver the code, your run results, and a pdf-report where you answer all the questions and explain your work.

Your report/notebook should contain your name and username.

Deliver one single zipped folder (.zip, .tgz or .tar.gz) which contains your complete solution.

Important: if you weren't able to finish the assignment, use the PDF report/Markdown to elaborate on what you've tried and what problems you encountered. Students who have made an effort and attempted all parts of the assignment will get a second chance even if they fail initially. This exercise will be graded PASS/FAIL.

## Goals of the assignment

The goal of this assignment is to get a better understanding of supervised learning with gradient descent. It will, in particular, consider the similarities and differences between linear classifiers and multi-layer feed forward networks (multi-layer perceptron, MLP) and the differences and

similarities between binary and multi-class classification. A main part will be dedicated to implementing and understanding the backpropagation algorithm.

## Tools

The aim of the exercises is to give you a look inside the learning algorithms. You may freely use code from the weekly exercises and the published solutions. You should not use ML libraries like scikit-learn or tensorflow.

You may use tools like NumPy and Pandas, which are not specific ML-tools.

The given precode uses NumPy. You are recommended to use NumPy since it results in more compact code, but feel free to use pure python if you prefer.

## Beware

There might occur typos or ambiguities. This is a revised assignment compared to earlier years, and there might be new typos. If anything is unclear, do not hesitate to ask. Also, if you think some assumptions are missing, make your own and explain them!

## Initialization

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from tabulate import tabulate
import sklearn  #for datasets
```

# Part 1: Linear classifiers

# Datasets

We start by making a synthetic dataset of 2000 datapoints and five classes, with 400 individuals in each class. (See https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html regarding how the data are generated.) We choose to use a synthetic dataset---and not a set of natural occuring data---because we are mostly interested in properties of the various learning algorithms, in particular the differences between linear classifiers and multi-layer neural networks together with the difference between binary and multi-class data.

When we are doing experiments in supervised learning, and the data are not already split into training and test sets, we should start by splitting the data. Sometimes there are natural ways to split the data, say training on data from one year and testing on data from a later year, but if that is not the case, we should shuffle the data randomly before splitting. (OK, that is not necessary with this particular synthetic data set, since it is already shuffled by default by scikit, but that will not be the case with real-world data.) We should split the data so that we keep the alignment between X and t, which may be achieved by shuffling the indices. We split into 50% for training, 25% for validation, and 25% for final testing. The set for final testing *must not be used* till the end of the assignment in part 3.

We fix the seed both for data set generation and for shuffling, so that we work on the same datasets when we rerun the experiments. This is done by the `random_state` argument and the `rng = np.random.RandomState(2022)` .

In [2]:
```
from sklearn.datasets import make_blobs

X, t = make_blobs(n_samples=[400, 400, 400, 400, 400], centers=[[0, 1], [4, 1], [8,
                  n_features=2, random_state=2022, cluster_std=1.0)
```

In [3]:
```
indices = np.arange(X.shape[0])
rng = np.random.RandomState(2022)
rng.shuffle(indices)
indices[:10]
```

Out[3]:
```
array([1018, 1295,  643, 1842, 1669,   86,  164, 1653, 1174,  747])
```

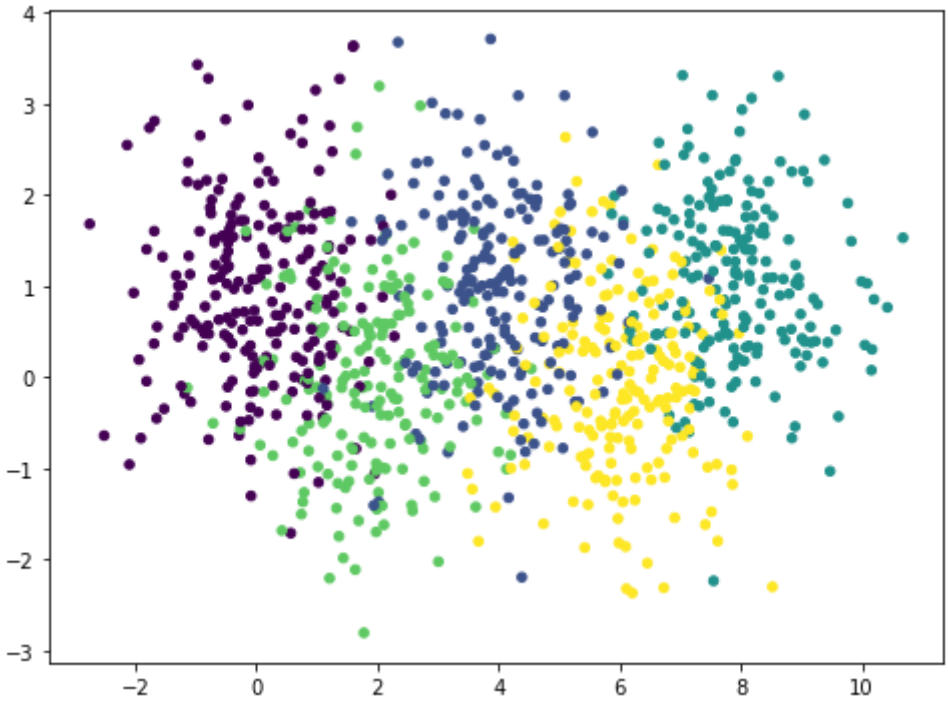
In [4]:
```
X_train = X[indices[:1000], :]
X_val = X[indices[1000:1500], :]
X_test = X[indices[1500:], :]
t_train = t[indices[:1000]]
t_val = t[indices[1000:1500]]
t_test = t[indices[1500:]]
```

Next, we will make a second dataset by merging the two smaller classes in (X,t) and call the new set (X, t2). This will be a binary set.

In [5]:
```
t2_train = t_train >= 3
t2_train = t2_train.astype('int')
t2_val = (t_val >= 3).astype('int')
t2_test = (t_test >= 3).astype('int')
```

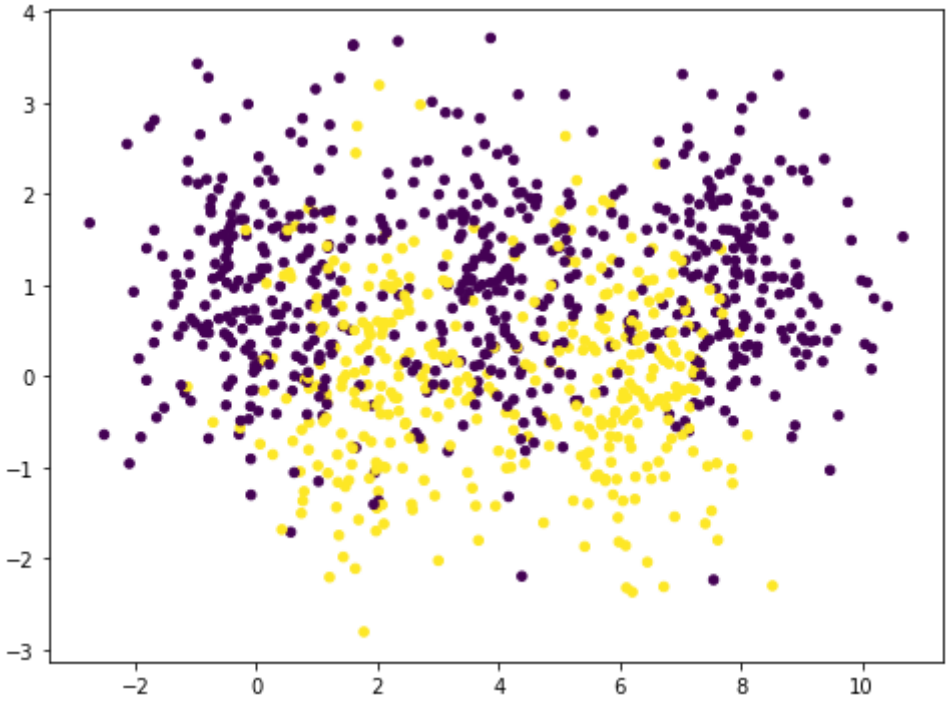
We can plot the two traing sets.

In [6]:
```
plt.figure(figsize=(8, 6))  # You may adjust the size
plt.scatter(X_train[:, 0], X_train[:, 1], c=t_train, s=20.0)
plt.show()
```




In [7]:

```
plt.figure(figsize=(8, 6))
plt.scatter(X_train[:, 0], X_train[:, 1], c=t2_train, s=20.0)
plt.show()
```




# Binary classifiers

## Linear regression

We see that that set (X, t2) is far from linearly separable, and we will explore how various classifiers are able to handle this. We start with linear regression. You may make your own implementation from scratch or start with the solution to the weekly exercise set 7, which we include here.

In [8]:

```
def add_bias(X):
    # Put bias in position 0
    sh = X.shape
    if len(sh) == 1:
        #X is a vector
        return np.concatenate([np.array([1]), X])
    else:
        # X is a matrix
        m = sh[0]
        bias = np.ones((m, 1))  # Makes a m*1 matrix of 1-s
        return np.concatenate([bias, X], axis=1)


def add_bias_minus_one(X):
    # Put bias in position 0
    sh = X.shape
    if len(sh) == 1:
        #X is a vector
        return np.concatenate([np.array([-1]), X])
    else:
        # X is a matrix
        m = sh[0]
        bias = np.ones((m, 1))  # Makes a m*1 matrix of 1-s
        bias = bias * -1
        return np.concatenate([bias, X], axis=1)
```

In [9]:

```
class NumpyClassifier:
    """Common methods to all numpy classifiers --- if any"""

    def accuracy(self, X_test_a, y_test_a, **kwargs):
        pred = self.predict(X_test_a, **kwargs)
        if len(pred.shape) > 1:
            pred = pred[:, 0]
        return np.sum(pred == y_test_a) / len(pred)

    def predict(self, X_test_p, param):
        pass
```

In [249…

```
class NumpyLinRegClass(NumpyClassifier):

    def __init__(self):
        self.X_train = []
        self.t_train = []
        self.X_val = None
        self.t_val = None
        self.e = 0
        self.mseTrainArray = []
        self.mseValArray = []
        self.weights = []
        self.accuracyTrainArray = []
        self.accuracyValArray = []

    @staticmethod
    def mse(y, y_pred):
        sum_errors = 0.
        for i in range(0, len(y)):
            sum_errors += (y[i] - y_pred[i]) ** 2
        mean_squared_error = sum_errors / len(y)
        return mean_squared_error

    def fit(self, X_train_f, t_train_f,
```

```
            eta_f=0.1, epochs=10, loss_diff=None,
            X_val_f=None, t_val_f=None):
        self.e = 0
        self.X_train = X_train_f
        self.t_train = t_train_f

        if X_val_f is not None and t_val_f is not None:
            self.X_val = X_val_f
            self.t_val = t_val_f
        """X_train is a Nxm matrix, N data points, m features
        t_train are the targets values for training data"""

        (k, m) = self.X_train.shape
        X_train_bias = add_bias(self.X_train)
        self.weights = np.zeros(m + 1)
        tryWeights = self.weights.copy()

        #for self.e in range(epochs):
        while True:
            if self.e >= epochs - 1:
                break
            self.e = self.e + 1

            tryWeights -= eta_f / k * X_train_bias.T @ (X_train_bias @ tryWeights -
            if len(self.mseTrainArray) > 1:
                thisTrainMse = self.mse(self.t_train, self.predict(self.X_train, wei
                prevTrainMse = self.mseTrainArray[-1]
                self.mseTrainArray.append(thisTrainMse)
                self.accuracyTrainArray.append(self.accuracy(self.X_train, self.t_tr
                if self.t_val is not None and self.X_val is not None:
                    self.mseValArray.append(self.mse(self.t_val, self.predict(self.X
                    self.accuracyValArray.append(self.accuracy(self.X_val, self.t_va
                if prevTrainMse > thisTrainMse:
                    difference = prevTrainMse - thisTrainMse
                    if difference > loss_diff:
                        self.weights = tryWeights.copy()
                    else:
                        break
            else:
                self.mseTrainArray.append(self.mse(self.t_train, self.predict(self.X
                if self.t_val is not None and self.X_val is not None:
                    self.mseValArray.append(self.mse(self.t_val, self.predict(self.X
        self.e += 1

    def predict(self, x, threshold=0.5, weights=None):
        z = add_bias(x)

        if weights is None:
            weights = self.weights

        score = z @ weights
        return score > threshold

    def confusionMatrix(self, X_c, t_c):
        prediction = self.predict(X_c)

        cm = np.zeros((2, 2))

        for index in range(2):
            for j in range(2):
                cm[index, j] = np.sum(np.where(prediction == index, 1, 0) * np.where

        return cm
```

```
    def printStats(self):
        mseFig, mseAx = plt.subplots(1)
        accuracyFig, accuracyAx = plt.subplots(1)

        print("Number of epochs -> " + str(self.e))
        mseAx.title.set_text("MSE Plot")
        accuracyAx.title.set_text("Accuracy Plot")

        accuracyAx.plot(range(0, len(self.accuracyTrainArray)), self.accuracyTrainAr
        mseAx.plot(range(0, len(self.mseTrainArray)), self.mseTrainArray, label="Tra

        if self.t_val is not None and self.X_val is not None:
            mseAx.plot(range(0, len(self.mseValArray)), self.mseValArray, label="Val
            accuracyAx.plot(range(0, len(self.accuracyValArray)), self.accuracyValAr

        mseAx.legend(loc="upper right")
        accuracyAx.legend(loc="lower right")
```

In [ ]:

```
for e in [1, 2, 5, 10, 50, 100, 1000, 10000, 100000]:
    cl = NumpyLinRegClass()
    # if you remove loss_diff it works much better
    cl.fit(X_train_f=X_train, t_train_f=t2_train, eta_f=0.01, loss_diff=0.00001,
           epochs=e, X_val_f=X_val, t_val_f=t2_val)
    print("epochs -> " + str(e))
    print("accuracy -> " + str(cl.accuracy(X_val, t2_val)))
    print("")
```

In [32]:

```
for eta in [1, 0.1, 0.01, 0.001, 0.0001, 0.00001]:
    cl = NumpyLinRegClass()
    # if you remove loss_diff it works much better
    cl.fit(X_train_f=X_train, t_train_f=t2_train, eta_f=eta, loss_diff=0.00001,
           epochs=1000, X_val_f=X_val, t_val_f=t2_val)
    print("eta -> " + str(eta))
    print("accuracy -> " + str(cl.accuracy(X_val, t2_val)))
    print("")
```

```
C:\Users\r2000\AppData\Local\Temp/ipykernel_10844/2705456577.py:44: RuntimeWarning:
invalid value encountered in matmul
  tryWeights -= eta_f / k * X_train_bias.T @ (X_train_bias @ tryWeights - self.t_tra
in)
C:\Users\r2000\AppData\Local\Temp/ipykernel_10844/2705456577.py:44: RuntimeWarning:
invalid value encountered in subtract
  tryWeights -= eta_f / k * X_train_bias.T @ (X_train_bias @ tryWeights - self.t_tra
in)
eta -> 1
accuracy -> 0.576

eta -> 0.1
accuracy -> 0.576

eta -> 0.01
accuracy -> 0.676

eta -> 0.001
accuracy -> 0.588

eta -> 0.0001
accuracy -> 0.576

eta -> 1e-05
```

```
accuracy -> 0.576
```

In [48]:

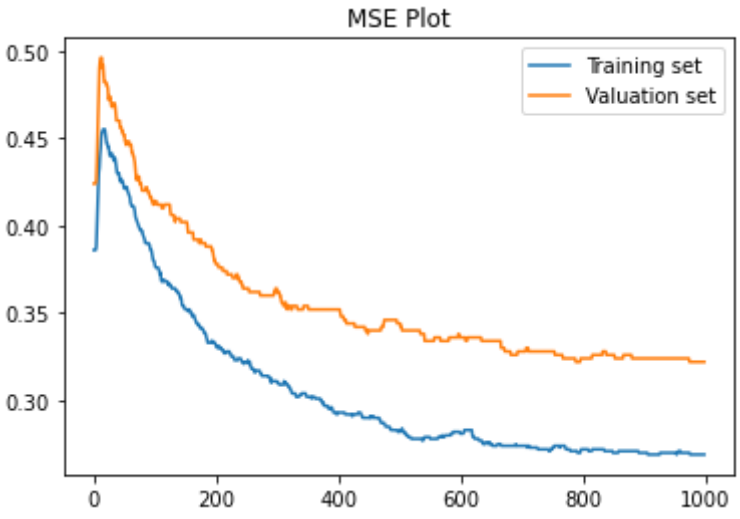
```
for loss_diff in [1, 0.1, 0.01, 0.001, 0.0001, 0.00001]:
    cl = NumpyLinRegClass()
    cl.fit(X_train_f=X_train, t_train_f=t2_train, eta_f=0.01, loss_diff=loss_diff,
           epochs=1000, X_val_f=X_val, t_val_f=t2_val)
    print("loss_diff -> " + str(loss_diff))
    print("accuracy -> " + str(cl.accuracy(X_val, t2_val)))
    print("epochs -> " + str(cl.e))
    print("")
```

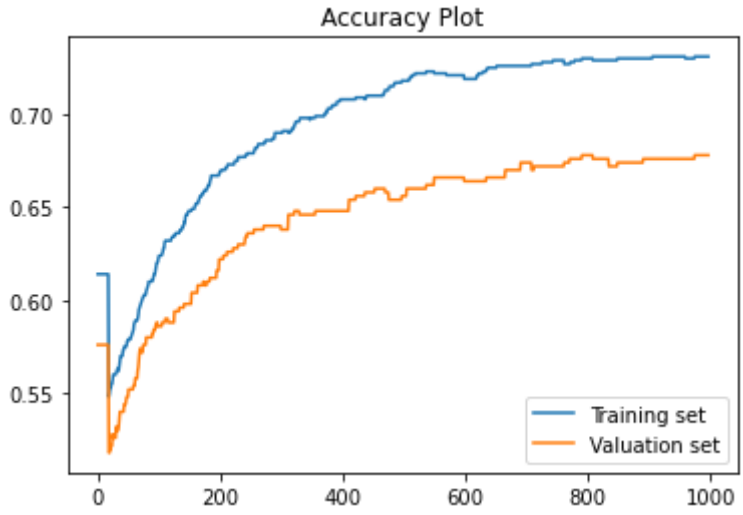
```
loss_diff -> 1
accuracy -> 0.576
epochs -> 21

loss_diff -> 0.1
accuracy -> 0.576
epochs -> 2

loss_diff -> 0.01
accuracy -> 0.576
epochs -> 21

loss_diff -> 0.001
accuracy -> 0.608
epochs -> 173

loss_diff -> 0.0001
accuracy -> 0.678
epochs -> 1000

loss_diff -> 1e-05
accuracy -> 0.678
epochs -> 1000
```

In [72]:

```
cl = NumpyLinRegClass()
cl.fit(X_train_f=X_train, t_train_f=t2_train, eta_f=0.01, loss_diff=0.0001,
       epochs=1000, X_val_f=X_val, t_val_f=t2_val)
cl.printStats()
```

```
Number of epochs -> 1000
```






**Answer**

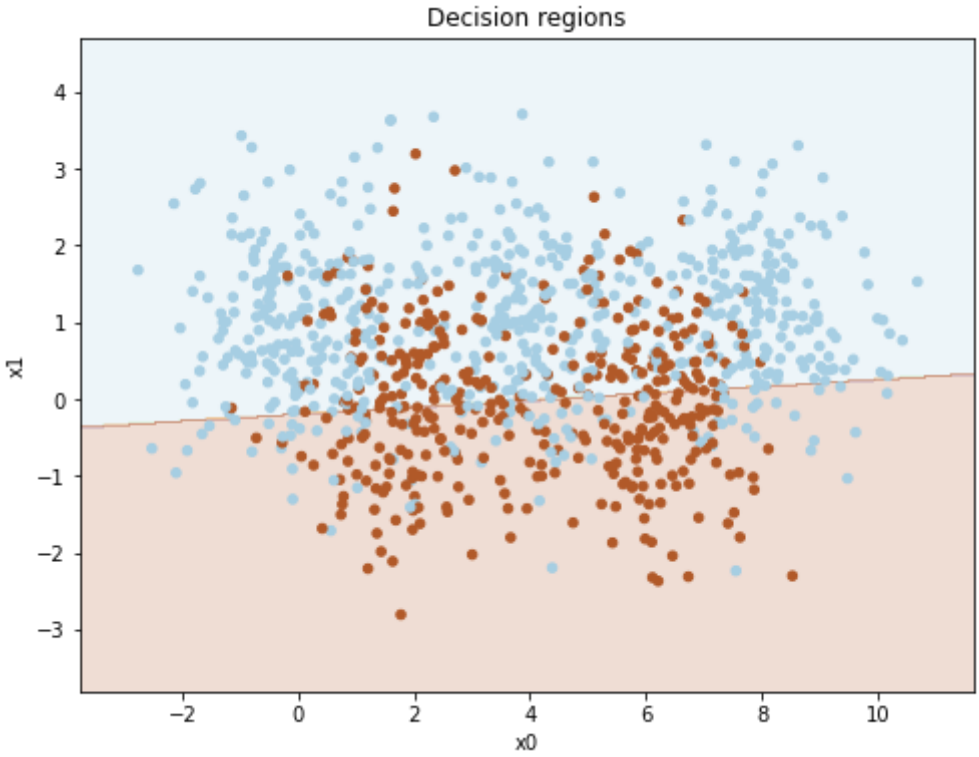
MSE and Accuracy are proportional inverse. Training and validation sets follow the same pattern. Training set has the better statistic because our classifier is trained on it.

The result is far from impressive. Experiment with various settings for the hyper-parameters, eta and epochs. Report how the accuracy vary with the hyper-parameter settings. When you are satisfied with the result, you may plot the decision boundaries, as below.

Feel free to improve the colors and the rest av of the graphics. We have chosen a simple set-up which can be applied to more than two classes without substanial modifications.

In [11]:
```
def plot_decision_regions(X_p, t_p, clf, size=(8, 6)):
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X_p[:, 0].min() - 1, X_p[:, 0].max() + 1
    y_min, y_max = X_p[:, 1].min() - 1, X_p[:, 1].max() + 1
    h = 0.02  # step size in the mesh
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    plt.figure(figsize=size)  # You may adjust this

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.2, cmap='Paired')

    plt.scatter(X_p[:, 0], X_p[:, 1], c=t_p, s=20.0, cmap='Paired')

    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title("Decision regions")
    plt.xlabel("x0")
    plt.ylabel("x1")

    plt.show()
```

In [12]:
```
plot_decision_regions(X_train, t2_train, cl)
```




## Loss

The linear regression classifier is trained with mean squared error loss. So far, we have not calculated the loss explicitly in the code. Extend the code to calculate the loss on the training set for each epoch and to store the losses such that the losses can be inspected after training.

Train a classifier with your best settings from last point. After training, plot the loss as a function of the number of epochs.

## Control training

The training runs for a number of epochs. We cannot know beforehand for how many epochs it is reasonable to run the training. One possibility is to run the training until the learning does not improve much. Extend the fit-method with a keyword argument, `loss_diff` , and stop training when the loss has not improved with more than loss_diff. Also add an attribute to the classifier which tells us after fitting how many epochs were ran.

In addition, extend the fit-method with optional arguments for a validation set (X_val, t_val). If a validation set is included in the call to fit, calculate the loss for the validation set, and the accuracy for both the training set and the validation set for each epoch.

Train classifiers with the best value for learning rate so far, and with varying values for `loss_diff` . For each run report, `loss_diff` , accuracy and number of epochs ran.

After a succesful training, plot both training loss snd vslidation loss as functions of the number of epochs in one figure, and both accuracies as functions of the number of epochs in another figure. Comment on what you see.

## Logistic regression

You should now do similarly for a logistic regression classifier. Calculate loss and accuracy for training set and, when provided, also for validation set.

Remember that logistic regression is trained with cross-entropy loss. Hence the loss function is calculated differently than for linear regression.

After a succesful training, plot both losses as functions of the number of epochs in one figure, and both accuracies as functions of the number of epochs in another figure.

Comment on what you see. Do you see any differences between the linear regression classifier and the logistic regression classifier on this dataset?

### Starting point: Code from weekly 7

In [13]:
```
def logistic(x):
    return 1 / (1 + np.exp(-x))
```

In [99]:
```
class NumpyLogReg(NumpyClassifier):
    def __init__(self):
        self.eta = 0

        self.celTrainArray = []
        self.trainWeights = []

        self.celValArray = []
        self.valWeights = []

        self.accuracyTrainArray = []
        self.accuracyValArray = []

        self.e = 0

        self.t_val = None
        self.X_val = None

    @staticmethod
    def cel(y, y_pred):
        #eps should be the smallest number you can get of the float type
        eps = np.finfo(float).eps
        loss = -np.sum(y * np.log(y_pred + eps))
        return loss / float(y_pred.shape[0])

    def fit(self, X_train_f, t_train_f, eta_f=0.1, epochs=10, X_val_f=None, t_val_f=
        """X_train is a Nxm matrix, N data points, m features
        t_train are the targets values for training data"""

        (k, m) = X_train_f.shape
        X_train_bias = add_bias(X_train_f)

        X_val_bias = None
        if X_val_f is not None and t_val_f is not None:
            self.X_val = X_val_f
            self.t_val = t_val_f
            self.valWeights = np.zeros(m + 1)
            X_val_bias = add_bias(X_val_f)

        self.trainWeights = np.zeros(m + 1)
        self.eta = eta_f
        for self.e in range(epochs):
```

```
            self.trainWeights -= eta_f / k * X_train_bias.T @ (
                    self.forward(X_train_bias, weights=self.trainWeights) - t_train_
            self.celTrainArray.append(self.cel(t_train_f, self.predict(X_train_f, we
            self.accuracyTrainArray.append(self.accuracy(X_train_f, t_train_f, weigh

            if X_val_f is not None and t_val_f is not None:
                self.valWeights -= eta_f / k * X_val_bias.T @ (
                        self.forward(X_val_bias, weights=self.valWeights) - t_val_f)
                self.celValArray.append(self.cel(t_val_f, self.predict(X_val_f, weig
                self.accuracyValArray.append(self.accuracy(X_val_f, t_val_f, weights
        self.e += 1

    def accuracy(self, X_test_a, y_test_a, weights=None):

        if weights is None:
            weights = self.trainWeights

        pred = self.predict(X_test_a, weights=weights)
        if len(pred.shape) > 1:
            pred = pred[:, 0]
        return np.sum(pred == y_test_a) / len(pred)

    def forward(self, X_f, weights=None):
        if weights is None:
            weights = self.trainWeights

        return logistic(X_f @ weights)

    def score(self, x, weights=None):
        z = add_bias(x)
        score = self.forward(z, weights)
        return score

    def predict(self, x, threshold=0.5, weights=None):
        z = add_bias(x)
        score = self.forward(z, weights)
        return score > threshold

    def printStats(self):

        mseFig, mseAx = plt.subplots(1)
        accuracyFig, accuracyAx = plt.subplots(1)

        print("Number of epochs -> " + str(self.e))
        mseAx.title.set_text("MSE Plot")
        accuracyAx.title.set_text("Accuracy Plot")

        accuracyAx.plot(range(0, len(self.accuracyTrainArray)), self.accuracyTrainAr
        mseAx.plot(range(0, len(self.celTrainArray)), self.celTrainArray, label="Tra

        if self.t_val is not None and self.X_val is not None:
            mseAx.plot(range(0, len(self.celValArray)), self.celValArray, label="Val
            accuracyAx.plot(range(0, len(self.accuracyValArray)), self.accuracyValAr

        mseAx.legend(loc="upper right")
        accuracyAx.legend(loc="lower right")
```

In [82]:

```
for e in [1, 2, 5, 10, 50, 100, 1000, 10000, 100000]:
    cl = NumpyLogReg()
    cl.fit(X_train_f=X_train, t_train_f=t2_train, eta_f=0.01,
           epochs=e, X_val_f=X_val, t_val_f=t2_val)
    print("epochs -> " + str(e))
```

```
    print("accuracy -> " + str(cl.accuracy(X_val, t2_val)))
    print("")
```

```
epochs -> 1
accuracy -> 0.558

epochs -> 2
accuracy -> 0.56

epochs -> 5
accuracy -> 0.562

epochs -> 10
accuracy -> 0.566

epochs -> 50
accuracy -> 0.592

epochs -> 100
accuracy -> 0.636

epochs -> 1000
accuracy -> 0.674

epochs -> 10000
accuracy -> 0.674

epochs -> 100000
accuracy -> 0.674
```

In [83]:

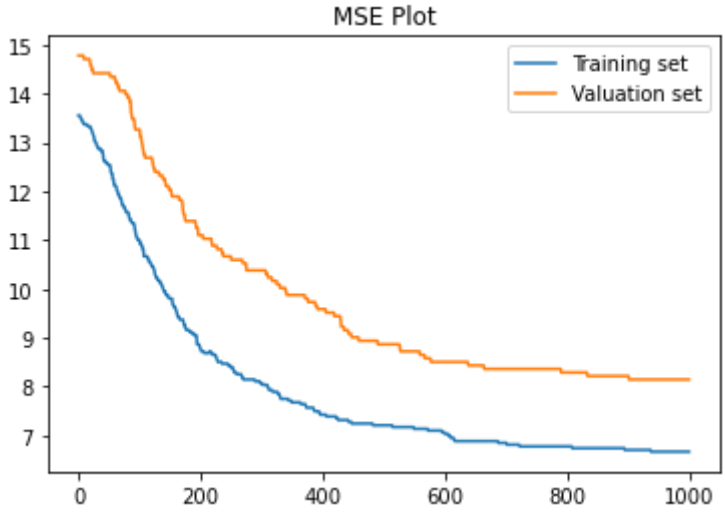
```
for eta in [1, 0.1, 0.01, 0.001, 0.0001, 0.00001]:
    cl = NumpyLogReg()
    cl.fit(X_train_f=X_train, t_train_f=t2_train, eta_f=eta,
           epochs=1000, X_val_f=X_val, t_val_f=t2_val)
    print("eta -> " + str(eta))
    print("accuracy -> " + str(cl.accuracy(X_val, t2_val)))
    print("")
```

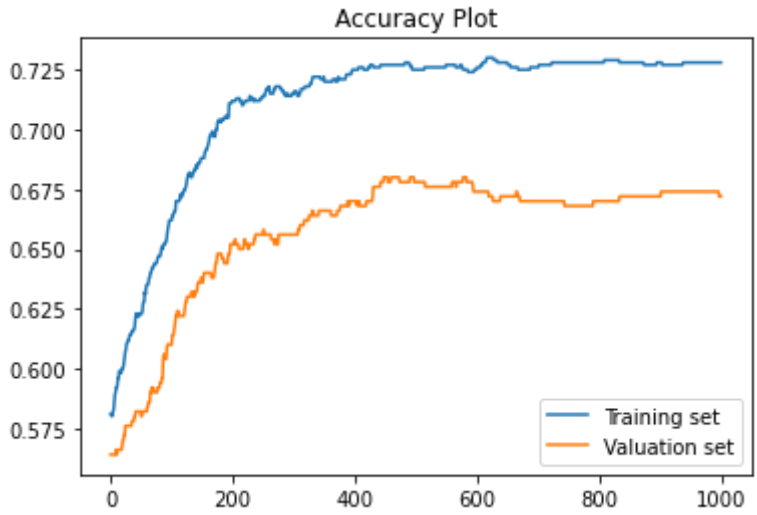
```
eta -> 1
accuracy -> 0.634

eta -> 0.1
accuracy -> 0.674

eta -> 0.01
accuracy -> 0.674

eta -> 0.001
accuracy -> 0.636

eta -> 0.0001
accuracy -> 0.566

eta -> 1e-05
accuracy -> 0.56
```

In [97]:

```
cl = NumpyLogReg()
cl.fit(X_train_f=X_train, t_train_f=t2_train, eta_f=0.01,
       epochs=1000, X_val_f=X_val, t_val_f=t2_val)
print(cl.accuracy(X_train, t2_train))
cl.printStats()
```

```
0.728
Number of epochs -> 1000
```




**Answer**

As it was before, the training set shows the best results. We can also see a small improvement from before, around 5% in accuracy

# Multi-class classifiers

We turn to the task of classifying when there are more than two classes, and the task is to ascribe one class to each input. We will now use the set (X, t).

## "One-vs-rest" with logistic regression

We saw in the lecture how a logistic regression classifier can be turned into a multi-class classifier using the one-vs-rest approach. We train one logistic regression classifier for each class. To predict the class of an item, we run all the binary classifiers and collect the probability score from each of them. We assign the class which ascribes the highest probability.

Build such a classifier. Train the resulting classifier on (X_train, t_train), test it on (X_val, t_val), tune the hyper-parameters and report the accuracy.

Also plot the decision boundaries for your best classifier similarly to the plots for the binary case.

In [146…

```
class OneVsRestClass(NumpyClassifier):
    def __init__(self):
        self.weights = []
        self.celTrainArray = []

        self.X_train = None
        self.t_train = None

    def forward(self, X_f, weights=None):
        if weights is None:
            weights = self.weights

        return logistic(X_f @ weights)

    def score(self, x, weights=None):
        z = add_bias(x)
        score = self.forward(z, weights)
        return score

    def predict(self, x, nLabels, threshold=0.5, weights=None):
        z = add_bias(x)
        if weights is None:
            weights = self.weights

        scores = np.zeros((len(x), nLabels))

        for i in range(0, len(weights)):
            scores[:, i] = self.forward(z, weights[i])
        predict = np.zeros(len(scores))
        for i in range(0, len(scores)):
            predict[i] = np.argmax(scores[i])
        return predict

    def accuracy(self, X_test_a, y_test_a, weights=None):
        pred = self.predict(X_test_a, len(np.unique(y_test_a)), weights=weights)
        if len(pred.shape) > 1:
            pred = pred[:, 0]
        return np.sum(pred == y_test_a) / len(pred)

    def fit(self, X_train_f, t_train_f, epochs, eta_f):
        self.X_train = X_train_f
        self.t_train = t_train_f

        (k, m) = self.X_train.shape
        X_train_bias_f = add_bias(self.X_train)
        self.weights = np.zeros((len(np.unique(self.t_train)), (m + 1)))

        for e in range(0, epochs):
            self.celTrainArray.append([])
            for i in range(0, len(np.unique(self.t_train))):
                t_train_of_i = self.t_train == i
                self.weights[i] -= eta_f / k * X_train_bias_f.T @ (
                        self.forward(X_train_bias_f, weights=self.weights[i]) - t_tr

    def printStats(self, size=(8, 6)):
        # Plot the decision boundary. For that, we will assign a color to each
        # point in the mesh [x_min, x_max]x[y_min, y_max].
        x_min, x_max = self.X_train[:, 0].min() - 1, self.X_train[:, 0].max() + 1
        y_min, y_max = self.X_train[:, 1].min() - 1, self.X_train[:, 1].max() + 1
        h = 0.02  # step size in the mesh
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

        Z = self.predict(np.c_[xx.ravel(), yy.ravel()], len(np.unique(self.t_train))
```

```
        plt.figure(figsize=size)  # You may adjust this

        # Put the result into a color plot
        Z = Z.reshape(xx.shape)

        plt.contourf(xx, yy, Z, alpha=0.2, cmap='Paired')

        plt.scatter(self.X_train[:, 0], self.X_train[:, 1], c=self.t_train, s=20.0,

        plt.xlim(xx.min(), xx.max())
        plt.ylim(yy.min(), yy.max())
        accuracy = self.accuracy(self.X_train, self.t_train)
        plt.title("Accuracy -> " + str(accuracy) + "\n\nDecision regions")
        plt.xlabel("x0")
        plt.ylabel("x1")

        plt.show()
```

In [149…

```
for e in [1, 2, 10, 50, 100, 1000, 10000, 15000, 50000]:
    cl = OneVsRestClass()
    cl.fit(X_train_f=X_train, t_train_f=t2_train, eta_f=0.01, epochs=e)
    print("epochs -> " + str(e))
    print("accuracy -> " + str(cl.accuracy(X_val, t2_val)))
    print("")
```

```
epochs -> 1
accuracy -> 0.558

epochs -> 2
accuracy -> 0.56

epochs -> 10
accuracy -> 0.566

epochs -> 50
accuracy -> 0.592

epochs -> 100
accuracy -> 0.636

epochs -> 1000
accuracy -> 0.674

epochs -> 10000
accuracy -> 0.674

epochs -> 15000
accuracy -> 0.674

epochs -> 50000
accuracy -> 0.674
```

In [126…

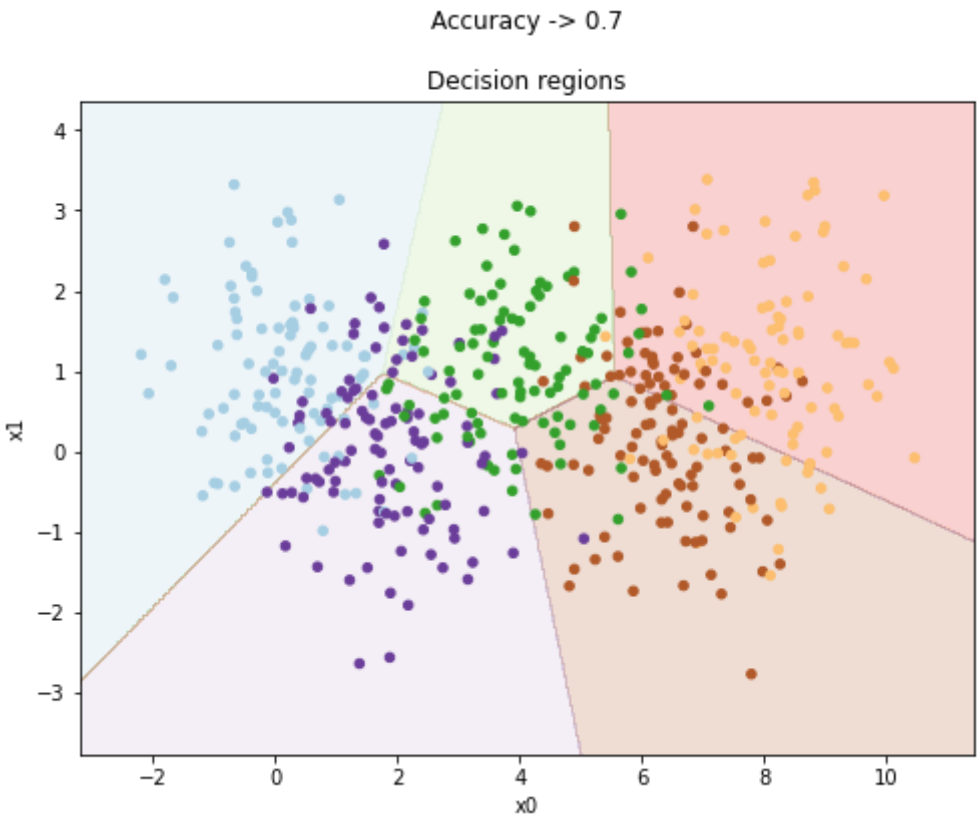
```
for eta in [1, 0.1, 0.01, 0.001, 0.0001, 0.00001]:
    cl = OneVsRestClass()
    cl.fit(X_train_f=X_train, t_train_f=t2_train, eta_f=eta, epochs=1000)
    print("eta -> " + str(eta))
    print("accuracy -> " + str(cl.accuracy(X_val, t2_val)))
    print("")
```

```
eta -> 1
accuracy -> 0.634
```

```
eta -> 0.1
accuracy -> 0.674

eta -> 0.01
accuracy -> 0.674

eta -> 0.001
accuracy -> 0.636

eta -> 0.0001
accuracy -> 0.566

eta -> 1e-05
accuracy -> 0.56
```

In [147…

```
cl = OneVsRestClass()
cl.fit(X_val, t_val, 15000, 0.01)
cl.printStats()
```




## For in4050-students: Multi-nominal logistic regression

The following part is only mandatory for in4050-students. In3050-students are also welcome to make it a try. Everybody has to return for the part 2 on multi-layer neural networks.

In the lecture, we contrasted the one-vs-rest approach with the multinomial logistic regression, also called softmax classifier. Implement also this classifier, tune the parameters, and compare the results to the one-vs-rest classifier. (Don't expect a large difference on a simple task like this.)

Remember that this classifier uses exponetiation followed by softmax in the forward phase. For loss, it uses cross-entropy loss. The loss has a somewhat simpler form than in the binary case. To

calculate the gradient is a little more complicated. The actual gradient and update rule is simple, however, as long as you have calculated the forward values correctly.

# Part II

## Multi-layer neural networks

We will implement the Multi-layer feed forward network (MLP, Marsland sec. 4.2.1), where we use mean squared loss together with logistic activation in both the hidden and the last layer.

Since this part is more complex, we will do it in two rounds. In the first round, we will go stepwise through the algorithm with the dataset (X, t). We will initialize the network and run a first round of training, i.e. one pass through the algorithm at p. 78 in Marsland.

In the second round, we will turn this code into a more general classifier. We can train and test this on (X, t) and (X, t2), but also on other datasets.

## Round 1: One epoch of training

### Scaling

First we have to scale our data. Make a standard scaler (normalizer) and scale the data. Remember, not to follow Marsland on this point. The scaler should be constructed from the training data only, but be applied both to training data and later on to validation and test data.

In [16]:
```
# Your code
mu = X_train.mean(axis=0)
sigma = X_train.std(axis=0)
Y_train = (X_train - mu) / sigma
Y_val = (X_val - mu) / sigma
Y_test = (X_train - mu) / sigma
```

### Initialization

We will only use one hidden layer. The number of nodes in the hidden layer will be a hyper-parameter provided by the user; let's call it *dim_hidden*. (*dim_hidden* is called *M* by Marsland.) Initially, we will set it to 3. This is a hyper-parameter where other values may give better results, and the hyper-parameter could be tuned.

Another hyper-parameter set by the user, is the learning rate. We set the initial value to 0.01, but also this may need tuning.

In [17]:
```
eta = 0.01  #Learning rate
dim_hidden = 3
```

We assume that the input *X_train* (after scaling) is a matrix of dimension *P x dim_in*, where *P* is the number of training instances, and *dim_in* is the number of features in the training instances (*L* in Marsland). Hence we can read *dim_in* off from *X_train*.

The target values have to be converted from simple numbers, *0, 2, ...* to "one-hot-encoded" vectors similarly to the multi-class task. After the conversion, we can read *dim_out* off from *t_train*.

In [18]:
```
X_train = X[indices[:1000], :]
X_val = X[indices[1000:1500], :]
X_test = X[indices[1500:], :]
t_train = t[indices[:1000]]
t_val = t[indices[1000:1500]]
t_test = t[indices[1500:]]

# convert t_train
values_of_t = len(np.unique(t_train))
t_train_encoded = np.zeros((len(t_train), values_of_t))

for i in range(0, len(t_train)):
    t_train_encoded[i][t_train[i]] = 1

t_train = t_train_encoded

dim_in = np.shape(Y_train)[1]  #len(Y_train[0])  # Calculate the correct value from
dim_out = np.shape(t_train)[1]  #len(t_train[0])  # Calculate the correct value from

print(dim_in)
print(dim_out)
```

```
2
5
```

We need two sets of weights: weights1 between the input and the hidden layer, and weights2, between the hidden layer and the output. Make sure that you take the bias terms into consideration and get the correct dimensions. The weight matrices should be initialized to small random numbers, not to zeros. It is important that they are initialized randomly, both to ensure that different neurons start with different initial values and to generate different results when you rerun the classifier. In this introductory part, we have chosen to fix the random state to make it easier for you to control your calculations. But this should not be part of your final classifier.

In [19]:
```
rng = np.random.RandomState(2022)
weightsInputHidden = (rng.rand(dim_in + 1, dim_hidden) * 2 - 1) / np.sqrt(dim_in)
weightsHiddenOutput = (rng.rand(dim_hidden + 1, dim_out) * 2 - 1) / np.sqrt(dim_hidd
```

In [20]:
```
print(weightsInputHidden.shape)
print()
print(weightsHiddenOutput.shape)
```

```
(3, 3)

(4, 5)
```

## Forwards phase

We will run the first step in the training, and start with the forward phase. Calculate the activations after the hidden layer and after the output layer. We will follow Marsland and use the logistic (sigmoid) activation function in both layers. Inspect whether the results seem reasonable with respect to format and values.

In [21]:
```
X_train_bias = add_bias_minus_one(Y_train)
print(X_train_bias)
z_input = X_train_bias @ weightsInputHidden
hidden_activations = logistic(z_input)
print(hidden_activations)
print("")

y_output_bias = add_bias_minus_one(hidden_activations)
z_output = y_output_bias @ weightsHiddenOutput
y_output = logistic(z_output)

output_activations = y_output
print(output_activations.shape)
```

```
[[-1.         -0.82679052 -0.64854735]
 [-1.          1.05291582  0.50410212]
 [-1.         -1.1052071   0.44135955]
 ...
 [-1.         -1.50242349  1.32757475]
 [-1.          0.95421139 -0.85006076]
 [-1.          0.77335446  0.2625872 ]]
[[0.70168682 0.41321325 0.54931018]
 [0.57621498 0.59448389 0.69219065]
 [0.83827789 0.45105146 0.69314647]
 ...
 [0.91657184 0.47108868 0.78913065]
 [0.4033634  0.51855904 0.51299278]
 [0.58646123 0.56435789 0.66372553]]

(1000, 5)
```

To control that you are on the right track, you may compare your first output value with our result. We have put the bias term -1 in position 0 in both layers. If you have done anything differently from us, you will not get the same numbers. But you may still be on the right track!

In [22]:
```
#outputs[0, :]
```

## Backwards phase

Calculate the delta terms at the output. We assume, like Marsland, that we use sum of squared errors. (This amounts to the same as using the mean square error).

In [23]:
```
deltaO = (output_activations - t_train) * output_activations * (1 - output_activatio
print(deltaO.shape)
eps = np.finfo(float).eps
loss = np.sum((output_activations - t_train) ** 2)
print(loss)
```

```
(1000, 5)
1010.1721716021603
```

Calculate the delta terms in the hidden layer.

In [24]:
```
print(weightsHiddenOutput)
print(weightsHiddenOutput.shape)
print("")
print(weightsInputHidden)
```

```
deltaH = y_output_bias * (1 - y_output_bias) * (deltaO @ weightsHiddenOutput.T)
print(deltaH)
```

```
[[ 0.25534462  0.38261397  0.37824303  0.38518453  0.52774934]
 [-0.15236916 -0.00596099 -0.1853185   0.13790511  0.55140373]
 [-0.46599894  0.28198504 -0.23960097 -0.23246968  0.29153125]
 [-0.55579925  0.02740963  0.42081427 -0.12835323 -0.33233223]]
(4, 5)

[[-0.6938717  -0.00133246 -0.54675803]
 [-0.63643285  0.26220593 -0.01840165]
 [ 0.56237224  0.20852872  0.56139063]]
[[-0.19365797  0.00271542  0.01066704 -0.00211523]
 [-0.18373476  0.02154314  0.01251571 -0.02818547]
 [-0.19469635  0.00207332  0.01196504 -0.00089222]
 ...
 [-0.2795743   0.00598042  0.02117345  0.01246094]
 [-0.10957271 -0.02276339 -0.02095291  0.01134536]
 [-0.10938841 -0.02234888 -0.01941509  0.01120068]]
```

Update the weights in both layers... See whether the weights have changed.

In [25]:

```
updateHiddenInput = eta * X_train_bias.T @ deltaH[:, :-1]
updateHiddenOutput = eta * y_output_bias.T @ deltaO

weightsInputHidden = weightsInputHidden - updateHiddenInput
weightsHiddenOutput = weightsHiddenOutput - updateHiddenOutput

print(weightsInputHidden)
print("")
print(weightsHiddenOutput)
```

```
[[-2.60372287  0.04198881 -0.53560097]
 [-1.01678414  0.27013581  0.0517494 ]
 [ 0.71208072  0.16253452  0.52970656]]

[[ 0.38835268  0.97979196  0.90803671  0.82761915  1.1372514 ]
 [-0.14956166 -0.36602762 -0.58673967 -0.13450645  0.07076574]
 [-0.55859937 -0.00789667 -0.45470182 -0.48339179 -0.00534288]
 [-0.61207289 -0.32495106  0.11154118 -0.43403304 -0.75454667]]
```

As an aid, you may compare your new weights with our results. But again, you may have done everything correctly even though you get a different result. For example, there are several ways to introduce the mean squared error. They may give different results after one epoch. But if you run sufficiently many epochs, you will get about the same classifier.

# Step 2: A Multi-layer neural network classifier

## Make the classifier

You want to train and test a classifier on (X, t). You could have put some parts of the code in the last step into a loop and run it through some iterations. But instead of copying code for every network we want to train, we will build a general Multi-layer neural network classfier as a class. This class will have some of the same structure as the classifiers we made for linear and logistic regression. The task consists mainly in copying in parts from what you did in step 1 into the template below. Remember to add the *self-* prefix where needed, and be careful in your use of variable names. And don't fix the random numbers within the classifier.

In [176...

```
from sklearn.datasets import make_blobs

X, t = make_blobs(n_samples=[400, 400, 400, 400, 400], centers=[[0, 1], [4, 1], [8,
                  n_features=2, random_state=2022, cluster_std=1.0)

indices = np.arange(X.shape[0])
rng = np.random.RandomState(2022)
rng.shuffle(indices)

X_train = X[indices[:1000], :]
X_val = X[indices[1000:1500], :]
X_test = X[indices[1500:], :]
t_train = t[indices[:1000]]
t_val = t[indices[1000:1500]]
t_test = t[indices[1500:]]


class MNNClassifier(NumpyClassifier):
    def __init__(self, eta_i=0.001, dim_hidden_i=3):
        """Initialize the hyperparameters"""
        self.eta = eta_i
        self.dim_hidden = dim_hidden_i
        self.X_train = None
        self.t_train = None
        self.t_train_not_encoded = None

        self.weightsInputHidden = None
        self.weightsHiddenOutput = None

        self.dim_in = 0
        self.dim_out = 0

        self.losses = []

        self.values_of_t = None

        self.X_train_bias = None
        self.y_output_bias = None

    def fit(self, X_train_f, t_train_f, epochs_f=100):
        """Initialize the weights. Train *epochs* many epochs."""

        mu_f = X_train.mean(axis=0)
        sigma_f = X_train.std(axis=0)
        self.X_train = (X_train_f - mu_f) / sigma_f
        self.t_train_not_encoded = t_train
        self.values_of_t = len(np.unique(t_train_f))
        t_train_encoded_f = np.zeros((len(t_train_f), self.values_of_t))

        for index in range(0, len(t_train_f)):
            t_train_encoded_f[index][t_train_f[index]] = 1

        self.t_train = t_train_encoded_f

        self.dim_in = np.shape(self.X_train)[1]
        self.dim_out = np.shape(self.t_train)[1]

        self.weightsInputHidden = (np.random.rand(self.dim_in + 1, self.dim_hidden)
        self.weightsHiddenOutput = (np.random.rand(self.dim_hidden + 1, self.dim_out
            self.dim_hidden)

        for e in range(epochs_f):
            output_activations_f = self.forward(self.X_train)
            deltaO_f = (output_activations_f - self.t_train) * output_activations_f
```

```
            self.losses.append(np.sum((output_activations_f - self.t_train) ** 2))
            deltaH_f = self.y_output_bias * (1 - self.y_output_bias) * (deltaO_f @ s

            updateHiddenInput_f = self.eta * self.X_train_bias.T @ deltaH_f[:, :-1]
            updateHiddenOutput_f = self.eta * self.y_output_bias.T @ deltaO_f

            self.weightsInputHidden = self.weightsInputHidden - updateHiddenInput_f
            self.weightsHiddenOutput = self.weightsHiddenOutput - updateHiddenOutput

    def forward(self, X_f):
        """Perform one forward step.
        Return a pair consisting of the outputs of the hidden_layer
        and the outputs on the final layer"""
        self.X_train_bias = add_bias_minus_one(X_f)
        z_input_f = self.X_train_bias @ self.weightsInputHidden
        hidden_activations_f = logistic(z_input_f)

        self.y_output_bias = add_bias_minus_one(hidden_activations_f)
        z_output_f = self.y_output_bias @ self.weightsHiddenOutput
        y_output_f = logistic(z_output_f)

        output_activations_f = y_output_f
        return output_activations_f

    def predict(self, X_p, **kwargs):

        X_train_bias_p = add_bias_minus_one(X_p)
        z_input_p = X_train_bias_p @ self.weightsInputHidden
        hidden_activations_p = logistic(z_input_p)
        y_output_bias_p = add_bias_minus_one(hidden_activations_p)
        z_output_f = y_output_bias_p @ self.weightsHiddenOutput

        predictions_sigmoid = logistic(z_output_f)
        predict = np.ones(len(predictions_sigmoid))
        predict *= -1

        for index in range(0, len(predictions_sigmoid)):
            predict[index] = np.argmax(predictions_sigmoid[index])

        return predict

    def accuracy(self, X_test_a, t_test_a, **kwargs):
        mu_f = X_train.mean(axis=0)
        sigma_f = X_train.std(axis=0)
        Y_test_a = (X_test_a - mu_f) / sigma_f

        p = self.predict(Y_test_a)
        return np.sum(p == t_test_a) / len(t_test_a)

    def printStats(self):
        plt.plot(range(0, len(self.losses)), self.losses)

        t_stats = self.t_train_not_encoded
        if self.values_of_t == 2:
            t_stats = t_stats > 0.5

        plt.title("Loss plot")

        plot_decision_regions(self.X_train, t_stats, self)
```

## Multi-class

Train the network on (X_train, t_train) (after scaling), and test on (X_val, t_val). Tune the hyperparameters to get the best result:

- number of epochs
- learning rate
- number of hidden nodes.

When you are content with the hyperparameters, you should run the same experiment 10 times, collect the accuracies and report the mean value and standard deviation of the accuracies across the experiments. This is common practise when you apply neural networks as the result may vary slightly between the runs. You may plot the decision boundaries for one of the runs.

Discuss shortly how the results and decsion boundaries compare to the "one-vs-rest" classifier.

In [156…

```
for e in [1, 2, 10, 50, 100, 1000, 10000, 15000, 50000]:
    mnn = MNNClassifier(eta_i=0.01, dim_hidden_i=3)
    mnn.fit(X_train, t_train, epochs_f=e)
    print("epochs -> " + str(e))
    print("accuracy -> " + str(mnn.accuracy(X_val, t_val)))
    print("")
```

```
epochs -> 1
accuracy -> 0.252

epochs -> 2
accuracy -> 0.464

epochs -> 10
accuracy -> 0.432

epochs -> 50
accuracy -> 0.52

epochs -> 100
accuracy -> 0.572

epochs -> 1000
accuracy -> 0.41

C:\Users\r2000\AppData\Local\Temp/ipykernel_10844/3472383605.py:2: RuntimeWarning: o
verflow encountered in exp
  return 1 / (1 + np.exp(-x))
epochs -> 10000
accuracy -> 0.506

epochs -> 15000
accuracy -> 0.554

epochs -> 50000
accuracy -> 0.436
```

In [157…

```
for eta in [1, 0.1, 0.01, 0.001, 0.0001, 0.00001]:
    mnn = MNNClassifier(eta_i=eta, dim_hidden_i=3)
    mnn.fit(X_train, t_train, epochs_f=100)
    print("eta -> " + str(eta))
    print("accuracy -> " + str(mnn.accuracy(X_val, t_val)))
    print("")
```

```
eta -> 1
accuracy -> 0.206

eta -> 0.1
accuracy -> 0.206

eta -> 0.01
accuracy -> 0.554

eta -> 0.001
accuracy -> 0.336

eta -> 0.0001
accuracy -> 0.378

eta -> 1e-05
accuracy -> 0.15
```

In [158…

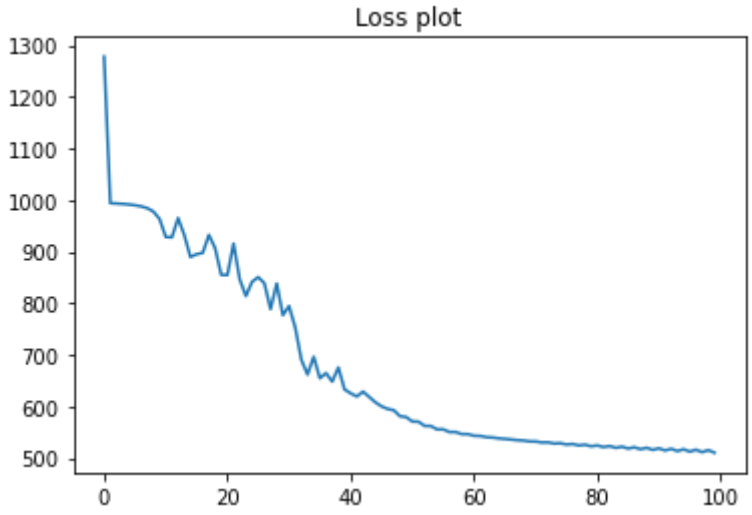
```
for hid in [1, 3, 6, 10, 30, 75]:
    mnn = MNNClassifier(eta_i=0.01, dim_hidden_i=hid)
    mnn.fit(X_train, t_train, epochs_f=100)
    print("dim_hidden -> " + str(hid))
    print("accuracy -> " + str(mnn.accuracy(X_val, t_val)))
    print("")
```

```
dim_hidden -> 1
accuracy -> 0.186

dim_hidden -> 3
accuracy -> 0.552

dim_hidden -> 6
accuracy -> 0.622

dim_hidden -> 10
accuracy -> 0.656

dim_hidden -> 30
accuracy -> 0.702

dim_hidden -> 75
accuracy -> 0.218
```

In [177…

```
accuracies = []
for rounds in range(0, 10):
    mnn = MNNClassifier(eta_i=0.01, dim_hidden_i=30)
    mnn.fit(X_train, t_train, epochs_f=100)
    accuracies.append(mnn.accuracy(X_val, t_val))

print("Mean accuracy - > " + str(np.mean(accuracies)))
print("Standard deviation accuracy - > " + str(np.std(accuracies)))
mnn.printStats()
```

```
Mean accuracy - > 0.581
Standard deviation accuracy - > 0.1372392072259236
```





**Answer**

The difference is that in the one-vs-rest the boundaries are straight lines, here, instead, the boundaries could be not straight also.

## Binary class

Let us see whether a multilayer neural network can learn a non-linear classifier. Train a classifier on (X_train, t2_train) and test it on (X_val, t2_val). Tune the hyper-parameters for the best result. Run ten times with the best setting and report mean and standard deviation. Plot the decision boundaries.

In [178…

```
for e in [1, 2, 10, 50, 100, 1000, 10000, 15000, 50000]:
    mnn = MNNClassifier(eta_i=0.01, dim_hidden_i=3)
    mnn.fit(X_train, t2_train, epochs_f=e)
    print("epochs -> " + str(e))
    print("accuracy -> " + str(mnn.accuracy(X_val, t2_val)))
    print("")
```

```
epochs -> 1
```

```
accuracy -> 0.576

epochs -> 2
accuracy -> 0.576

epochs -> 10
accuracy -> 0.628

epochs -> 50
accuracy -> 0.676

epochs -> 100
accuracy -> 0.684

epochs -> 1000
accuracy -> 0.63

epochs -> 10000
accuracy -> 0.576

C:\Users\r2000\AppData\Local\Temp/ipykernel_10844/3472383605.py:2: RuntimeWarning: o
verflow encountered in exp
  return 1 / (1 + np.exp(-x))
epochs -> 15000
accuracy -> 0.684

epochs -> 50000
accuracy -> 0.664
```

In [179…

```
for eta in [1, 0.1, 0.01, 0.001, 0.0001, 0.00001]:
    mnn = MNNClassifier(eta_i=eta, dim_hidden_i=3)
    mnn.fit(X_train, t2_train, epochs_f=100)
    print("eta -> " + str(eta))
    print("accuracy -> " + str(mnn.accuracy(X_val, t2_val)))
    print("")
```

```
eta -> 1
accuracy -> 0.576

eta -> 0.1
accuracy -> 0.67

eta -> 0.01
accuracy -> 0.704

eta -> 0.001
accuracy -> 0.578

eta -> 0.0001
accuracy -> 0.576

eta -> 1e-05
accuracy -> 0.616

C:\Users\r2000\AppData\Local\Temp/ipykernel_10844/3472383605.py:2: RuntimeWarning: o
verflow encountered in exp
  return 1 / (1 + np.exp(-x))
```

In [181…

```
for hid in [1, 3, 6, 10, 30, 75]:
    mnn = MNNClassifier(eta_i=0.01, dim_hidden_i=hid)
    mnn.fit(X_train, t2_train, epochs_f=100)
```

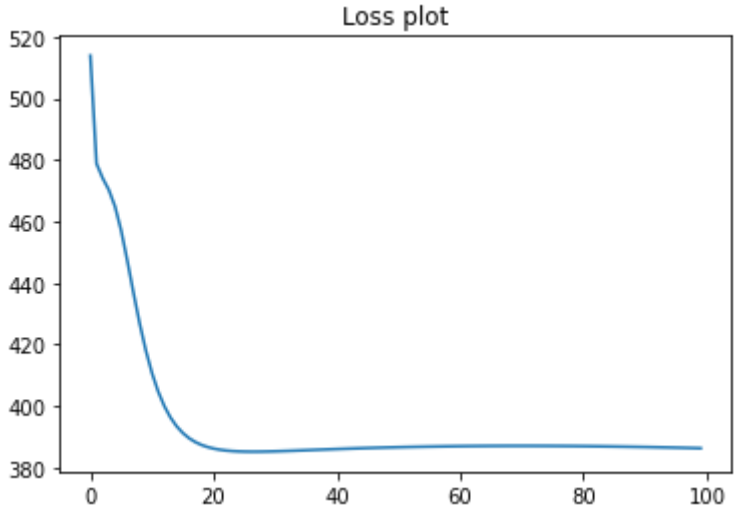
```
    print("dim_hidden -> " + str(hid))
    print("accuracy -> " + str(mnn.accuracy(X_val, t2_val)))
    print("")
```

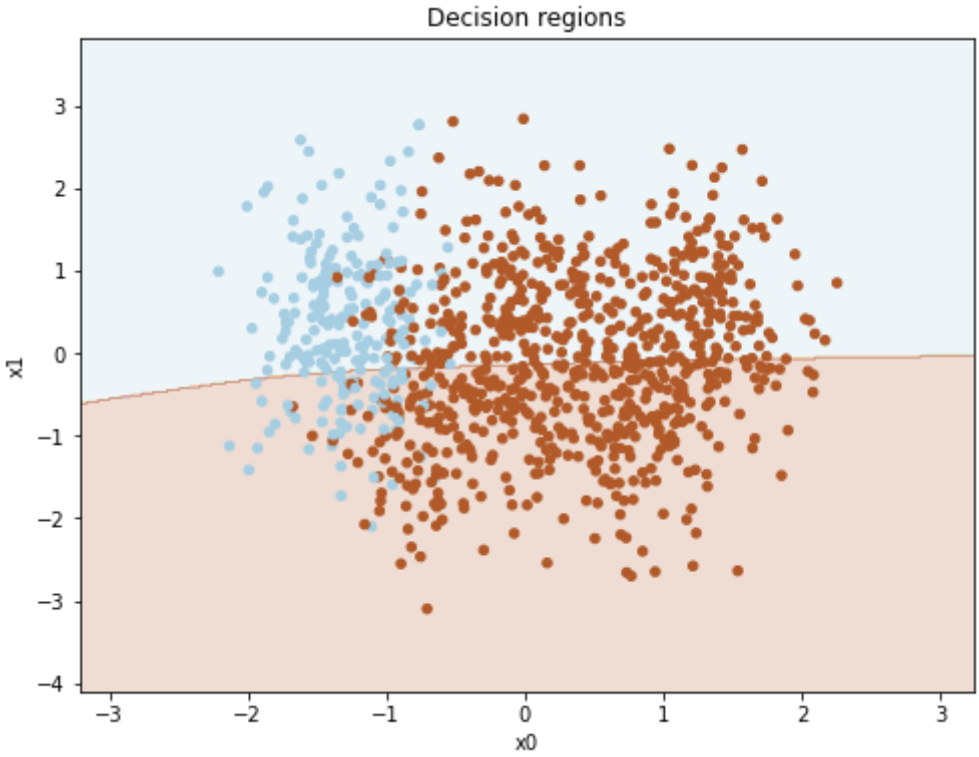
```
dim_hidden -> 1
accuracy -> 0.576

dim_hidden -> 3
accuracy -> 0.688

dim_hidden -> 6
accuracy -> 0.68

dim_hidden -> 10
accuracy -> 0.684

dim_hidden -> 30
accuracy -> 0.674

dim_hidden -> 75
accuracy -> 0.582
```

In [184…

```
accuracies = []
for rounds in range(0, 10):
    mnn = MNNClassifier(eta_i=0.01, dim_hidden_i=3)
    mnn.fit(X_train, t2_train, epochs_f=100)
    accuracies.append(mnn.accuracy(X_val, t2_val))

print("Mean accuracy - > " + str(np.mean(accuracies)))
print("Standard deviation accuracy - > " + str(np.std(accuracies)))
mnn.printStats()
```

```
Mean accuracy - > 0.6836000000000001
Standard deviation accuracy - > 0.009156418513807644
```






## For in4050-students: Early stopping

The following part is only mandatory for in4050-students. In3050-students are also welcome to make it a try. Everybody has to return for the part 2 on multi-layer neural networks.

There is a danger of overfitting if we run too many epochs of training. One way to control that is to use early stopping. We can use (X_val, t_val) as valuation set when training on (X_train, t_train).

Let *e=50* or *e=10* (You may try both or choose some other number) After *e* number of epochs, calculate the loss for both the training set (X_train, t_train) and the validation set (X_val, t_val), and store them.

Train a classifier for many epochs. Plot the losses for both the training set and the validation set in the same figure and see whether you get the same effect as in figure 4.11 in Marsland.

Modify the code so that the training stops if the loss on the validation set is not reduced by more than *t* after *e* many epochs, where *t* is a threshold you provide as a parameter.

Run the classifier with various values for *t* and report the accuracy and the numberof epochs ran.

# Part III: Final testing

We can now perform a final testing on the held-out test set.

## Binary task (X, t2)

Consider the linear regression classifier, the logistic regression classifier and the multi-layer network with the best settings you found. Train each of them on the training set and evaluate on the held-out test set, but also on the validation set and the training set. Report in a 3 by 3 table.

Comment on what you see. How do the three different algorithms compare? Also, compare the result between the different data sets. In cases like these, one might expect slightly inferior results on the held-out test data compared to the validation data. Is so the case?

Also report precision and recall for class 1.

In [3]:

```
cl = NumpyLinRegClass()
cl.fit(X_train_f=X_train, t_train_f=t2_train, eta_f=0.01, loss_diff=0.0001, epochs=1
cl_train = cl.accuracy(X_train, t2_train)
cl_val = cl.accuracy(X_val, t2_val)
cl_test = cl.accuracy(X_test, t2_test)

lr = NumpyLogReg()
lr.fit(X_train_f=X_train, t_train_f=t2_train, eta_f=0.01, epochs=1000)
lr_train = lr.accuracy(X_train, t2_train)
lr_val = lr.accuracy(X_val, t2_val)
lr_test = lr.accuracy(X_test, t2_test)

mnn = MNNClassifier(eta_i=0.01, dim_hidden_i=3)
mnn.fit(X_train, t2_train, epochs_f=100)
mnn_train = mnn.accuracy(X_train, t2_train)
mnn_val = mnn.accuracy(X_val, t2_val)
mnn_test = mnn.accuracy(X_test, t2_test)

table_array = np.array([["Train", cl_train, lr_train, mnn_train],
                        ["Valuate", cl_val, lr_val, mnn_val],
                        ["Test", cl_test, lr_test, mnn_test]])
headers = ["", "Linear regression", "Logistic regression", "Multi-layer neural netwo
table = tabulate(table_array, headers, tablefmt="fancy_grid")
print(table)

confusion = cl.confusionMatrix(X_train, t2_train)
tp = confusion[0, 0]
fp = confusion[0, 1]
fn = confusion[1, 0]
tn = confusion[1, 1]

print("Precision of Linear regression train set -> " + str(tp / (tp + fp)))
print("Recall of Linear regression train set -> " + str(tp / (tp + fn)))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_17908/2435135107.py in <module>
----> 1 cl = NumpyLinRegClass()
      2 cl.fit(X_train_f=X_train, t_train_f=t2_train, eta_f=0.01, loss_diff=0.0001,
 epochs=1000)
      3 cl_train = cl.accuracy(X_train, t2_train)
      4 cl_val = cl.accuracy(X_val, t2_val)
      5 cl_test = cl.accuracy(X_test, t2_test)

NameError: name 'NumpyLinRegClass' is not defined
```

**Answer**

All the data are pretty similar. In this case the test has good results.

# Multi-class task (X, t)

For IN3050 students compare the one-vs-rest classifier to the multi-layer preceptron. Evaluate on test, validation and training set as above. In4050-students should also include results from

the multi-nomial logistic regression.

Comment on the results.

In [228…

```
orc = OneVsRestClass()
orc.fit(X_val, t_val, 15000, 0.01)
orc_train = orc.accuracy(X_train, t_train)
orc_val = orc.accuracy(X_val, t_val)
orc_test = orc.accuracy(X_test, t_test)

mnn = MNNClassifier(eta_i=0.01, dim_hidden_i=30)
mnn.fit(X_train, t_train, epochs_f=10000)
mnn_train = mnn.accuracy(X_train, t_train)
mnn_val = mnn.accuracy(X_val, t_val)
mnn_test = mnn.accuracy(X_test, t_test)

table_array = np.array([["Train", orc_train, mnn_train],
                        ["Valuate", orc_val, mnn_val],
                        ["Test", orc_test, mnn_test]])

headers = ["", "One vs Rest", "Multi-layer neural network"]
table = tabulate(table_array, headers, tablefmt="fancy_grid")
print(table)
```

| | One vs Rest | Multi-layer neural network |
|---|---|---|
| Train | 0.748 | 0.783 |
| Valuate | 0.7 | 0.72 |
| Test | 0.732 | 0.746 |

**Answer**

In this case the MMN beats OvR. Also, the test set doesn't show as mush error as we can expect.