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 [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
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 [ ]:
```
from sklearn.datasets import make_blobs
X, t = make_blobs(n_samples=[400,400,400, 400, 400], centers=[[0,1],[4,1],[8,1],[2,0],[6,0]],
                  n_features=2, random_state=2019, cluster_std=1.0)
```

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

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

In [ ]:
```
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.

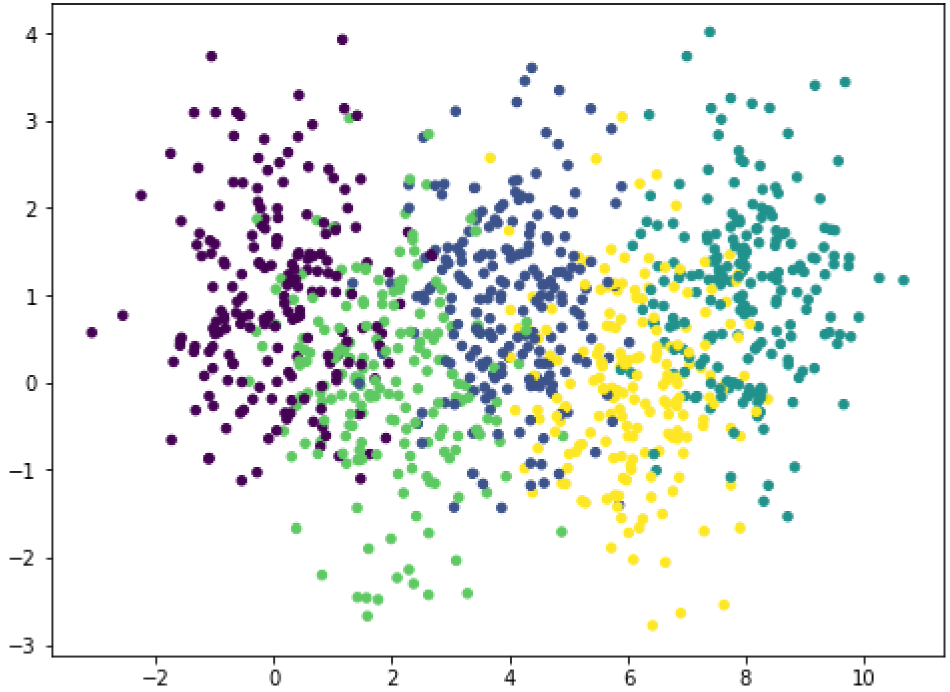
In [ ]:

```
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.

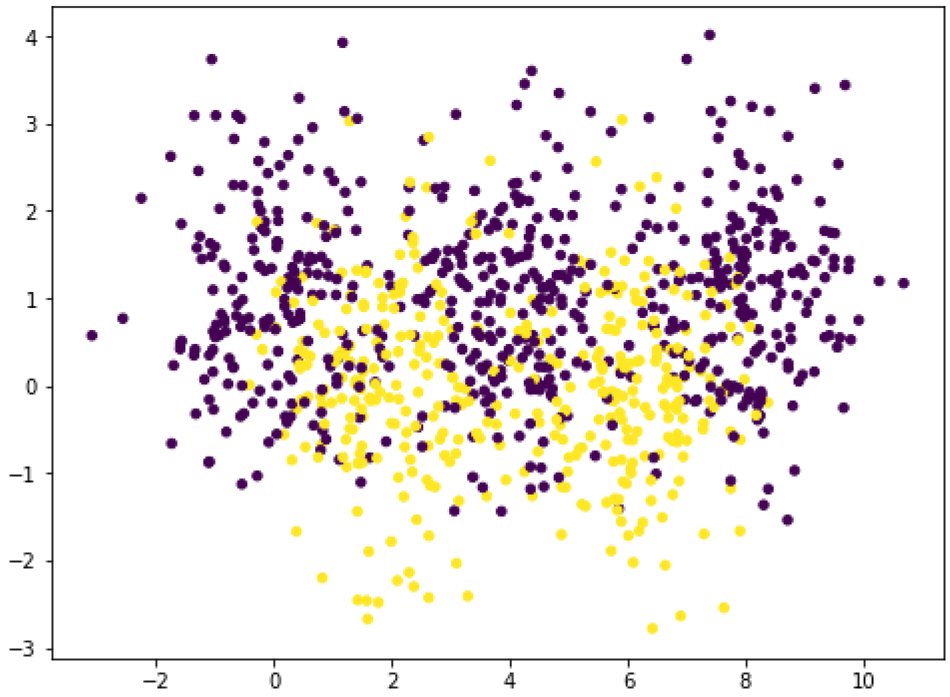
In [ ]:

```
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 [ ]:

```
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 [ ]:

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

In [ ]:

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

    epochs_used = 0

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

In [ ]:

```
class NumpyLinRegClass(NumpyClassifier):

    def fit(self, X_train, t_train, loss_diff, X_val = X_val, t_val = None, eta = 0.01, epochs=1000):
        """X_train is a Nxm matrix, N data points, m features
        t_train are the targets values for training data"""

        #cheking for optional args
        optional_args = True
        try:
            if X_val == None and t_val == None:
                optional_args = False
        except:
            optional_args = True

        if optional_args:
            check_acc = X_train
            check_acc_val= X_val
            X_val.shape
            X_val = add_bias(X_val)

        (k, m) = X_train.shape
        X_train = add_bias(X_train)
```

```
        self.weights = weights = np.zeros(m+1)

        training_loss = []
        validation_loss = []
        training_accuracy = []
        validation_accuracy = []

        #the following for loop could have been coded with less code, but I had another assignment to do.
        #basically: train, track loss if we do not have optional args, if we do have optional args then we do the same
        for e in range(epochs):
            #making sure we do not try loss diff unless we have something to compare current result to
            if not training_loss:
                weights -= eta / k *  X_train.T @ (X_train @ weights - t_train)
                error = mse(t_train, X_train @ weights)
                training_loss.append(error)
                if optional_args:
                    v_error = mse(t_val, X_val @ weights)
                    validation_loss.append(v_error)
                    training_accuracy.append(self.accuracy(check_acc, t_train))
                    validation_accuracy.append(self.accuracy(check_acc_val, t_val))
            #making sure we do not try loss diff unless we have something to compare current result to
            elif len(training_loss)==1:
                weights -= eta / k *  X_train.T @ (X_train @ weights - t2_train)
                error = mse(t_train, X_train @ self.weights)
                training_loss.append(error)
                if optional_args:
                    v_error = mse(t_val, X_val @ weights)
                    validation_loss.append(v_error)
                    training_accuracy.append(self.accuracy(check_acc, t_train))
                    validation_accuracy.append(self.accuracy(check_acc_val, t_val))
            #here we have enough values to start using loss_diff
            elif (training_loss[len(training_loss)-2]-training_loss[len(training_loss)-1])>loss_diff:
                weights -= eta / k *  X_train.T @ (X_train @ weights - t_train)
                error = mse(t_train, X_train @ self.weights)
                training_loss.append(error)
                if optional_args:
                    v_error = mse(t_val, X_val @ weights)
                    validation_loss.append(v_error)
                    training_accuracy.append(self.accuracy(check_acc, t_train))
                    validation_accuracy.append(self.accuracy(check_acc_val, t_val))
            else:
                #print("eta:", eta)
                #print("loss_diff:", loss_diff)
```

```
                return training_loss, validation_loss, training_accuracy, validation_accuracy

        #print("eta:", eta)
        #print("loss_diff:", loss_diff)
        return training_loss, validation_loss, training_accuracy, validation_accuracy

    def predict(self, x, threshold=0.5):
        z = add_bias(x)
        score = z @ self.weights
        return score>threshold
```

We can train and test a first classifier.

In [ ]:

```
cl = NumpyLinRegClass()
training_loss, validation_loss, training_accuracy, validation_accuracy = cl.fit(X_train, t2_train, 0.00001, X_val, t2_va
cl.epochs_used = len(training_loss)
print("Epochs:", len(training_loss))
print("Accuracy:", cl.accuracy(X_val, t2_val))

#plotting method
def plot(train_stats, title, xlab, ylab, val_stats = None):
    y_line = train_stats
    x_line = []
    for i in range(1, len(y_line)+1):
        x_line.append(i)
    plt.figure()
    plt.title(title)
    plt.xlabel(xlab)
    plt.ylabel(ylab)
    plt.plot(x_line, y_line)
    if val_stats:
        y_line = val_stats
        plt.plot(x_line, y_line)
        plt.legend(["Training set", "Validation set"])
    else:
        plt.legend(["Training set"])
```

```
eta: 0.01
loss_diff: 1e-05
Epochs: 646
Accuracy: 0.654
```

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.

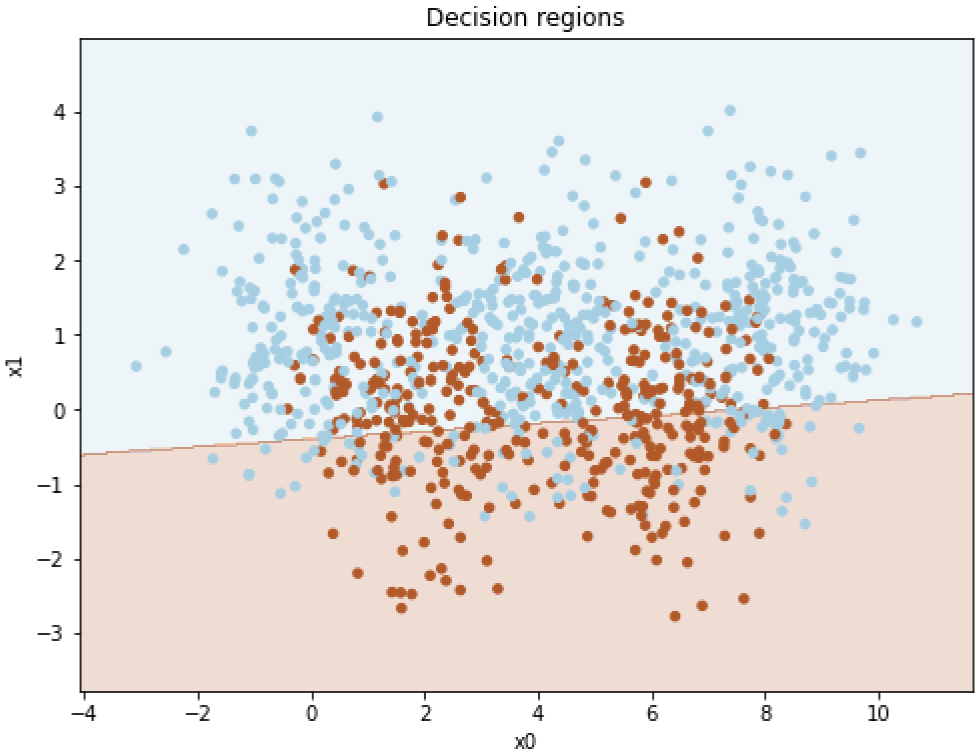
Feel free to improve the colors and the rest av of the graphics. We have chosen a simple set-up which can be applied to more than two classes without substanial modifications.

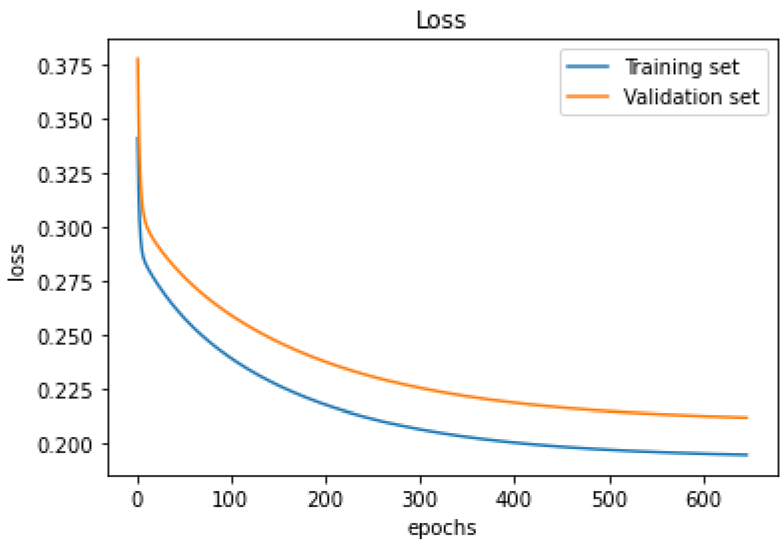
In [ ]:

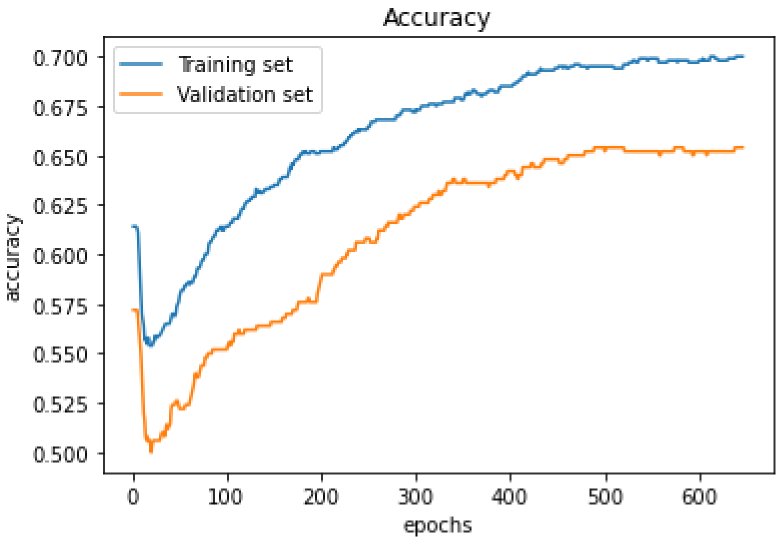
```
def plot_decision_regions(X, t, clf=[], size=(8,6)):
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    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[:,0], X[:,1], c=t, 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 [ ]:

```
plot_decision_regions(X_train, t2_train, cl)
plot(training_loss, "Loss", "epochs", "loss", validation_loss)
plot(training_accuracy, "Accuracy", "epochs", "accuracy", validation_accuracy)

#Choosing high ETA of 0.1, or low ETA of 0.00001 gives poor accuracy and high loss. A ETA of 0.01 gives a good balance.
```


Decision regions
x1
x0


Loss
Training set
Validation set
loss
epochs



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 [ ]:

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

#algorithm for measuring loss, modelled from lectures.
def log_loss(t, y):
  loss = -np.mean(t*(np.log(abs(y))) - (1-t)*np.log(1-abs(y)))
  return loss
```

In [ ]:

```
class NumpyLogReg(NumpyClassifier):

    def fit(self, X_train, t_train, loss_diff, X_val = None, t_val = None, eta = 0.01, epochs=1000):

        #cheking for optional args
        optional_args = True
        try:
            if X_val == None and t_val == None:
                optional_args = False
        except:
            optional_args = True
```

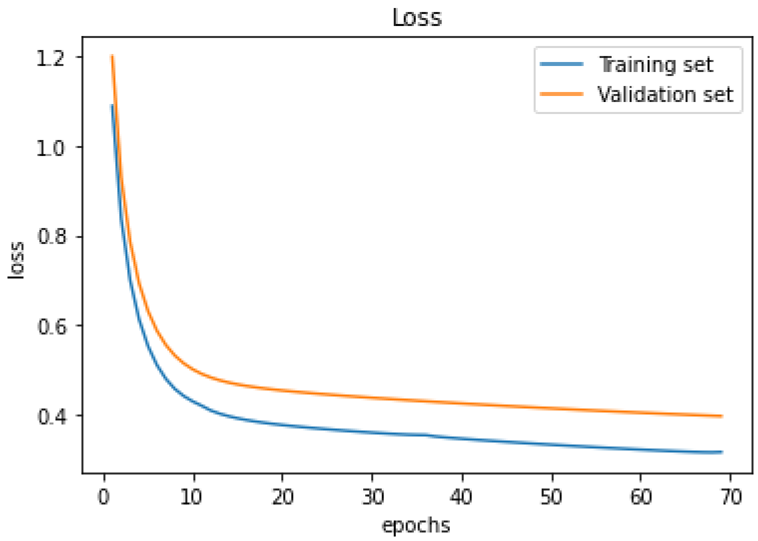
```
        if optional_args:
            check_acc = X_train
            check_acc_val= X_val
            X_val.shape
            X_val = add_bias(X_val)

        (k, m) = X_train.shape
        X_train = add_bias(X_train)

        self.weights = weights = np.zeros(m+1)

        training_log_loss = []
        validation_log_loss = []
        training_log_accuracy = []
        validation_log_accuracy = []
        for e in range(epochs):
            #making sure we do not try loss diff unless we have something to compare current result to
            if not training_log_loss:
                    weights -= eta / k *  X_train.T @ (X_train @ weights - t_train)
                    error = log_loss(t_train, X_train @ weights)
                    training_log_loss.append(error)
                    if optional_args:
                        v_error = log_loss(t_val, X_val @ weights)
                        validation_log_loss.append(v_error)
                        training_log_accuracy.append(self.accuracy(check_acc, t_train))
                        validation_log_accuracy.append(self.accuracy(check_acc_val, t_val))
            #making sure we do not try loss diff unless we have something to compare current result to
            elif len(training_log_loss)==1:
                    weights -= eta / k *  X_train.T @ (X_train @ weights - t_train)
                    error = log_loss(t_train, X_train @ weights)
                    training_log_loss.append(error)
                    if optional_args:
                        v_error = log_loss(t_val, X_val @ weights)
                        validation_log_loss.append(v_error)
                        training_log_accuracy.append(self.accuracy(check_acc, t_train))
                        validation_log_accuracy.append(self.accuracy(check_acc_val, t_val))
            #here we have enough values to start using loss_diff
            elif (training_log_loss[e-2]-training_log_loss[e-1])>loss_diff:
                    weights -= eta / k *  X_train.T @ (X_train @ weights - t_train)
                    error = log_loss(t_train, X_train @ weights)
                    training_log_loss.append(error)
                    if optional_args:
                        v_error = log_loss(t_val, X_val @ weights)
                        validation_log_loss.append(v_error)
```

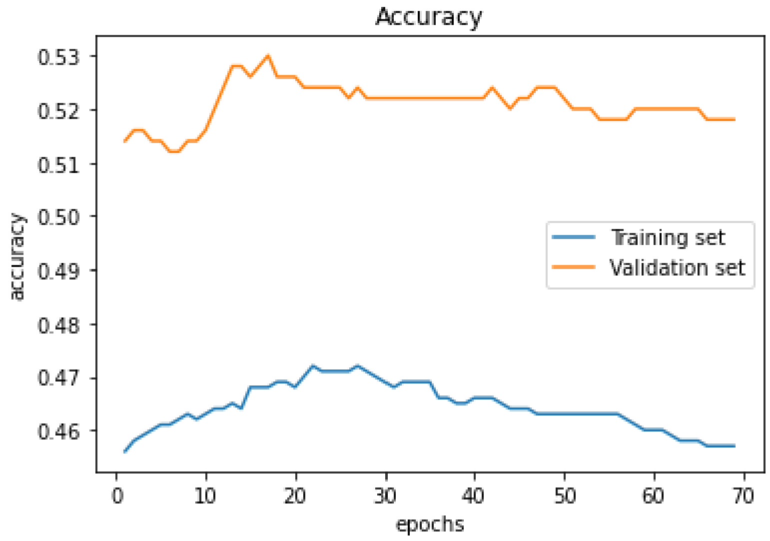
```
                        training_log_accuracy.append(self.accuracy(check_acc, t_train))
                        validation_log_accuracy.append(self.accuracy(check_acc_val, t_val))
                else:
                    #print("eta:", eta)
                    #print("loss_diff:", loss_diff)
                    return training_log_loss, validation_log_loss, training_log_accuracy, validation_log_accuracy
        return training_log_loss, validation_log_loss, training_log_accuracy, validation_log_accuracy

    def forward(self, X):
        return logistic(X @ self.weights)

    def score(self, x):
        z = add_bias(x)
        score = self.forward(z)
        return score

    def predict(self, x, threshold=0.5):
        z = add_bias(x)
        score = self.forward(z)
        return (score>threshold).astype('int')
```

In [ ]:

```
log_cl = NumpyLogReg()
training_log_loss, validation_log_loss, training_log_accuracy, validation_log_accuracy = log_cl.fit(X_train, t2_train, 
print("Epochs:", len(training_log_loss))
print("Accuracy:", log_cl.accuracy(X_val, t2_val))
plot(training_log_loss, "Loss", "epochs", "loss", validation_log_loss)
plot(training_log_accuracy, "Accuracy", "epochs", "accuracy", validation_log_accuracy)

#Accuracy for the logistic classifier varies more, but also appears to have a higher loss. Logistic classif. also finds
```

```
Epochs: 69
Accuracy: 0.518
```





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.

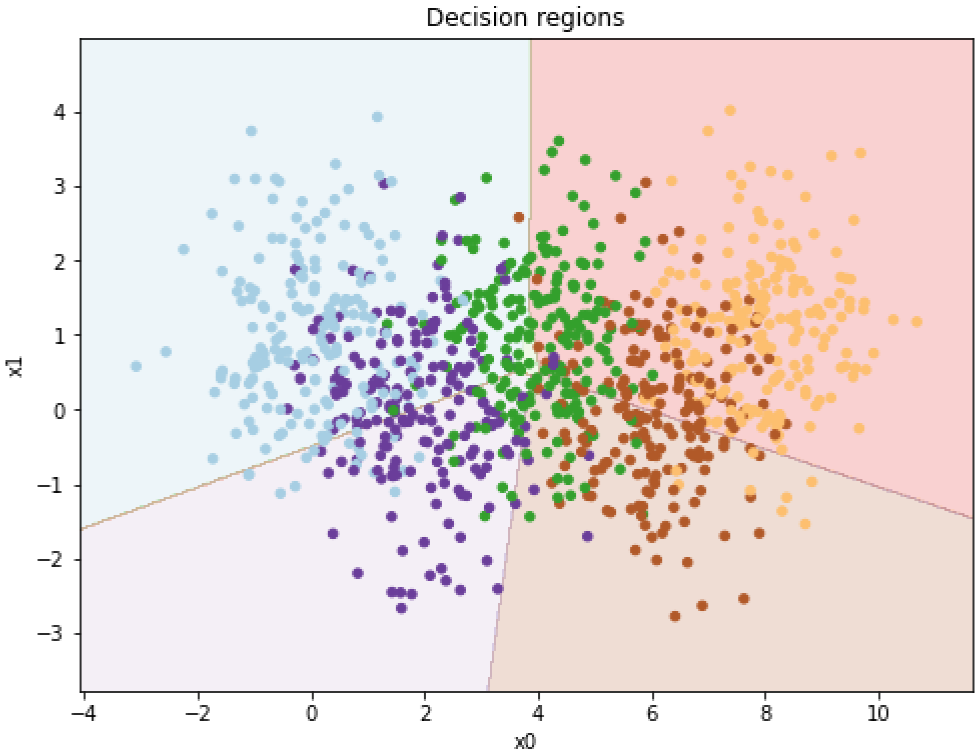
Also plot the decision boundaries for your best classifier similarly to the plots for the binary case.

In [ ]:

```
class NumpyLogRegMultiClass(NumpyClassifier):
    def fit(self, X_train, t_train, eta = 0.01, epochs=800):
        (k, m) = X_train.shape
        X_train = add_bias(X_train)
        (c, r) = t_train.shape

        self.weights = weights = np.zeros((m+1, r))

        for e in range(epochs):
            weights -= eta / k *  X_train.T @ (X_train @ weights - t_train)

    def forward(self, X):
        return X @ logistic(self.weights)

    def score(self, x):
        z = add_bias(x)
        score = self.forward(z)
        return score

    #could have used argmax further down, but did not have time to implement this
    def predict(self, x):
        z = add_bias(x)
        score = self.forward(z)
        (c, r) = x.shape
        output = np.zeros((c,), dtype=int)
        index = 0
        for s in score:
            best_p = -100
            for p in s:
                if p > best_p:
                    best_p = p
            label = s.tolist().index(best_p)
```

```
                output[index] = label
                index +=1
        return (output).astype('int')

def make_binary(label, classes):
    array = np.zeros((classes,), dtype=int)
    array[label] = 1
    return np.array(array)

def hot_encoding(data):
    vectors = []
    classes = np.unique(data)
    for c in data:
        array = []
        for l in range(0, len(classes)):
            array.append(0)
        array[c] = 1
        vector = np.array(array)
        vectors.append(vector)
    return np.array(vectors)

multi_cl = NumpyLogRegMultiClass()
multi_cl.fit(X_train, hot_encoding(t_train))
acc = multi_cl.accuracy(X_val, t_val)
print("Accuracy:", acc)
plot_decision_regions(X_train, t_train, multi_cl)
```

```
Accuracy: 0.564
```



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 [ ]:

```
class MMScaler():

    def fit(self, X_train):
        self.maxes = np.max(X_train, axis=0)
        self.mins = np.min(X_train, axis=0)

    def transform(self, X):
        return (X - self.mins)/(self.maxes - self.mins)

scaler = MMScaler()
scaler.fit(X_train)
scaled_data = scaler.transform(X_train)
```

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 [ ]:
```
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 [ ]:
```
# convert t_train
(p, d) = scaled_data.shape
#as explained in the task, dim_in must be d from line above
dim_in =  d
vectors = hot_encoding(t_train)
(c, r) = vectors.shape
dim_out = r
```

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 [ ]:
```
rng = np.random.RandomState(2022)
weights1 = (rng.rand(dim_in + 1, dim_hidden) * 2 - 1)/np.sqrt(dim_in)
weights2 = (rng.rand(dim_hidden+1, dim_out) * 2 - 1)/np.sqrt(dim_hidden)
```

In [ ]:
```
weights1
```

Out[ ]:
```
array([[-0.6938717 , -0.00133246, -0.54675803],
       [-0.63643285,  0.26220593, -0.01840165],
       [ 0.56237224,  0.20852872,  0.56139063]])
```

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 [ ]:

```
def add_negative_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([np.negative(bias), X], axis  = 1)

def activation(w, x_t):
    #w = weight
    #x = input
    bias = add_negative_bias(x_t)
    weighted_input = bias @ w
    sigmoid = logistic(weighted_input)
    return sigmoid

hidden_activations = activation(weights1, scaled_data)
```

In [ ]:

```
output_activations = activation(weights2, hidden_activations)
print(output_activations[0, :])
```

```
[0.26999006 0.4474561  0.4150512  0.37452743 0.44141406]
```

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 [ ]:

```
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 [ ]:
```
deltao = (vectors-output_activations)*output_activations*(1.0-output_activations)
print(deltao)
```

```
[[-0.05321381  0.13661048 -0.10076767 -0.08773554 -0.10883844]
 [-0.05211821 -0.11119401  0.14184521 -0.08687875 -0.10806852]
 [-0.05444775 -0.1101613  -0.10043682  0.14636927 -0.10913064]
 ...
 [ 0.14321548 -0.11063392 -0.10089049 -0.08808082 -0.1104096 ]
 [-0.05249266 -0.11066058 -0.10083636  0.14649455 -0.10951299]
 [-0.05492664  0.13680023 -0.10048647 -0.08791519 -0.10803311]]
```

Calculate the delta terms in the hidden layer.

In [ ]:
```
deltah = add_negative_bias(hidden_activations)*(1.0-add_negative_bias(hidden_activations))*(np.dot(deltao,np.transpose(v
print(deltah)
```

```
[[ 0.18133447 -0.010501    0.01879538  0.00825926]
 [ 0.18539593 -0.02087268 -0.0127441   0.02774492]
 [ 0.19051232 -0.00280232 -0.01180861  0.00054608]
 ...
 [ 0.27223574 -0.01566201 -0.02111423 -0.01574041]
 [ 0.19050525 -0.00282082 -0.01198568  0.00027344]
 [ 0.18113965 -0.01073821  0.01915115  0.00883349]]
```

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

In [ ]:
```
updatew1 = eta*(np.dot(np.transpose(add_negative_bias(scaled_data)),deltah[:,1:]))
updatew2 = eta*(np.dot(np.transpose(add_negative_bias(hidden_activations)),deltao))
weights1 += updatew1
weights2 += updatew2
```

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.

In [ ]:
```
print("New weights:")
print(weights1)
```

```
New weights:
[[-0.63398482  0.01265499 -0.56942272]
 [-0.66395765  0.268383    0.01561559]
 [ 0.5236028   0.19740452  0.5772698 ]]
```

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.

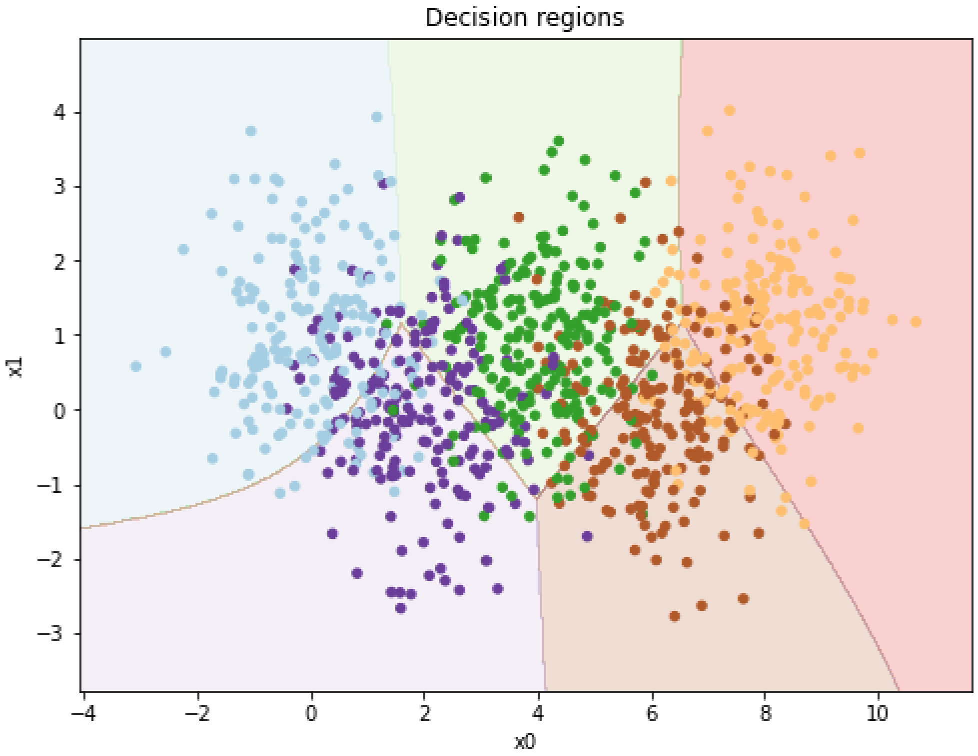
In [ ]:

```python
class MNNClassifier():
    """A multi-layer neural network with one hidden layer"""

    def __init__(self,eta = 0.1, dim_hidden = 6):
        """Initialize the hyperparameters"""
        self.eta = eta
        self.dim_hidden = dim_hidden
        self.scaler = MMScaler()

    def fit(self, X_train, t_train, epochs = 1000):
        """Initialize the weights. Train *epochs* many epochs."""

        self.scaler.fit(X_train)
        scaled_input = self.scaler.transform(X_train)
        # convert t_train
        (p, d) = scaled_input.shape
        #as explained in the task, dim_in must be d from line above
        dim_in =  d
        vectors = hot_encoding(t_train)
        (c, r) = vectors.shape
        dim_out = r

        self.weights1 = weights1 = (np.random.rand(dim_in + 1, dim_hidden) * 2 - 1)/np.sqrt(dim_in)
        self.weights2 = weights2 = (np.random.rand(dim_hidden+1, dim_out) * 2 - 1)/np.sqrt(dim_hidden)
```

```
        for e in range(epochs):
            # Run one epoch of forward-backward
            hidden_activations = activation(weights1, scaled_input)
            output_activations = activation(weights2, hidden_activations)
            deltao = (vectors-output_activations)*output_activations*(1.0-output_activations)
            deltah = add_negative_bias(hidden_activations)*(1.0-add_negative_bias(hidden_activations))*(np.dot(deltao,n
            updatew1 = eta*(np.dot(np.transpose(add_negative_bias(scaled_input)),deltah[:,1:]))
            updatew2 = eta*(np.dot(np.transpose(add_negative_bias(hidden_activations)),deltao))
            weights1 += updatew1
            weights2 += updatew2


    def forward(self, X):
        """Perform one forward step.
        Return a pair consisting of the outputs of the hidden_layer
        and the outputs on the final layer"""
        self.scaler.fit(X)
        scaled_input = self.scaler.transform(X)
        hidden_activations = activation(self.weights1, scaled_input)
        output_activations = activation(self.weights2, hidden_activations)
        return hidden_activations, output_activations

    #predict function so that we can plot, could have used argmax
    def predict(self, x):
        hidden_output, final_output = self.forward(x)
        (c, r) = final_output.shape
        output = np.zeros((c,), dtype=int)
        index = 0
        for s in final_output:
            best_p = -100
            for p in s:
                if p > best_p:
                    best_p = p
            label = s.tolist().index(best_p)
            output[index] = label
            index +=1
        return (output).astype('int')

    #custom accuracy func, could have used argmax
    def accuracy(self, X_test, t_test):
        """Calculate the accuracy of the classifier for the pair (X_test, t_test)
        Return the accuracy"""
```

```
        hidden_output, final_output = self.forward(X_test)
        (c, r) = final_output.shape
        output = np.zeros((c,), dtype=int)
        index = 0
        for s in final_output:
            best_p = -100
            for p in s:
                if p > best_p:
                    best_p = p
            label = s.tolist().index(best_p)
            output[index] = label
            index +=1
        pred = (output).astype('int')
        if len(pred.shape) > 1:
            pred = pred[:,0]
        return output, np.sum(pred==t_test)/len(pred)

mnn = MNNClassifier()
mnn.fit(X_train, t_train)
output, acc = mnn.accuracy(X_val, t_val)
print("Accuracy:", acc)
plot_decision_regions(X_train, t_train, mnn)
```

```
Accuracy: 0.726
```



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.

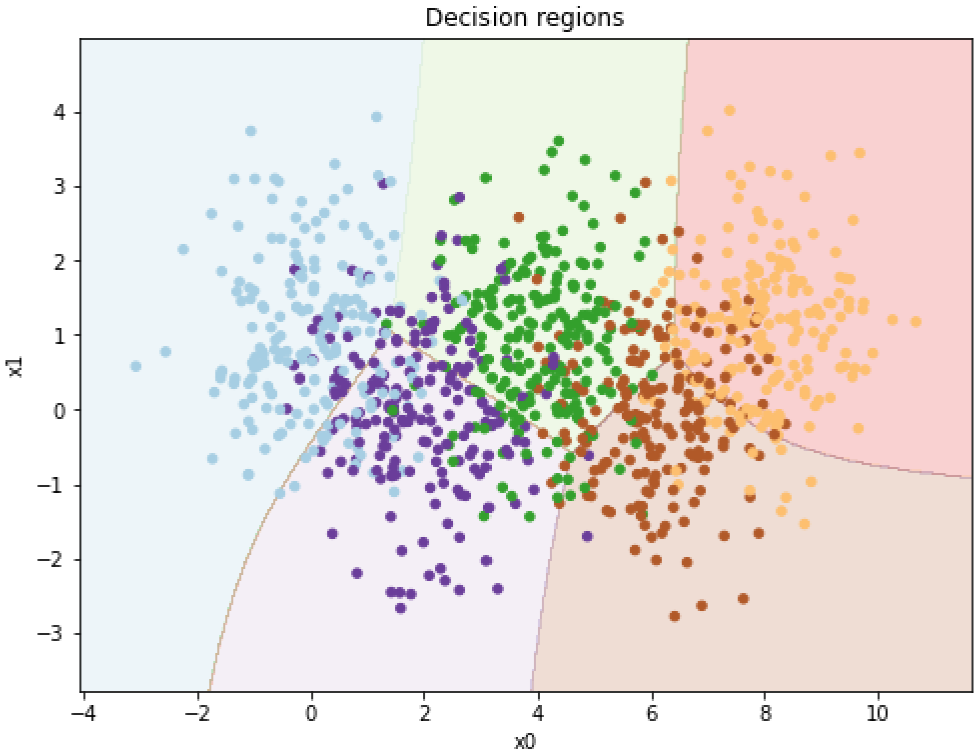
When you are content with the hyperparameters, you should run the same experiment 10 times, collect the accuracies and report the mean value and standard deviation of the accuracies across the experiments. This is common practise when you apply neural networks as the result may vary slightly between 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 [ ]:
```
import statistics
```

```
all_acc = []
for e in range(10):
    mnn = MNNClassifier()
    mnn.fit(X_train, t_train)
    output, acc = mnn.accuracy(X_val, t_val)
    all_acc.append(acc)
    print("Accuracy:", acc)
print("Mean value:", statistics.mean(all_acc))
print("Standard deviation:", statistics.stdev(all_acc))
plot_decision_regions(X_train, t_train, mnn)

#We see in that the accuracy is better for MNN than OVR, and we can see why in the decision boundaries. MNN is able to 
```

```
Accuracy: 0.74
Accuracy: 0.738
Accuracy: 0.736
Accuracy: 0.732
Accuracy: 0.738
Accuracy: 0.736
Accuracy: 0.718
Accuracy: 0.718
Accuracy: 0.742
Accuracy: 0.718
Mean value: 0.7316
Standard deviation: 0.009743373816770734
```



Binary class

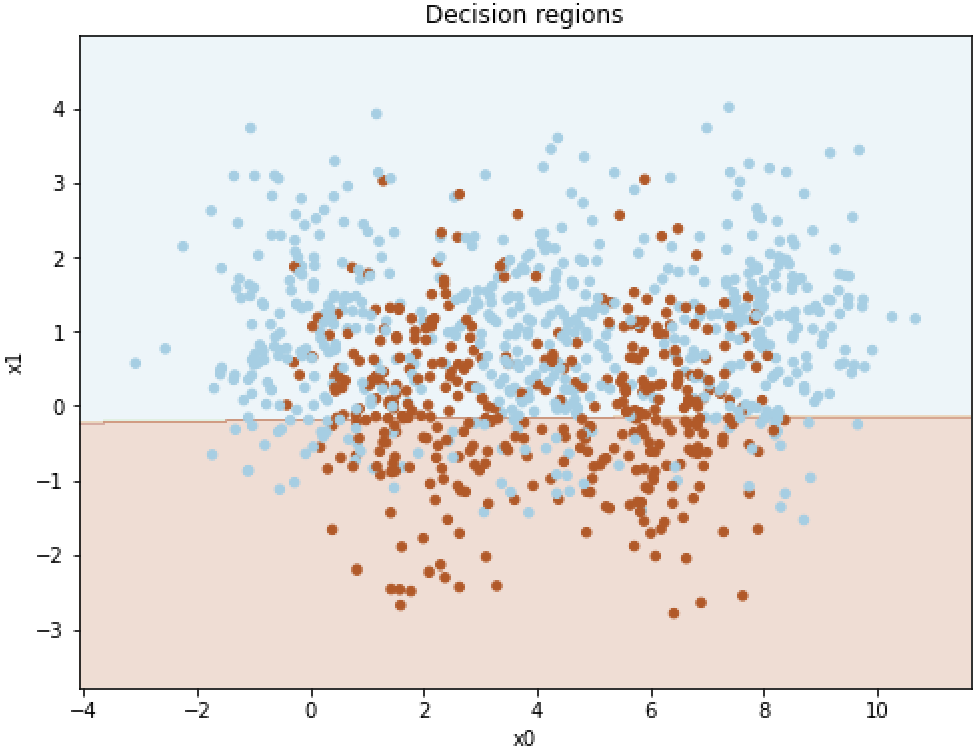
Let us see whether a multilayer neural network can learn a non-linear classifier. Train a classifier on (X_train, t2_train) and test it on (X_val, t2_val). Tune the hyper-parameters for the best result. Run ten times with the best setting and report mean and standard deviation. Plot the decision boundaries.

In [ ]:

```
import statistics

all_acc = []
for e in range(10):
    mnn = MNNClassifier()
    mnn.fit(X_train, t2_train)
    output, acc = mnn.accuracy(X_val, t2_val)
    all_acc.append(acc)
    print("Accuracy:", acc)
print("Mean value:", statistics.mean(all_acc))
```

```
print("Standard deviation:", statistics.stdev(all_acc))
plot_decision_regions(X_train, t2_train, mnn)
```

```
Accuracy: 0.67
Accuracy: 0.672
Accuracy: 0.67
Accuracy: 0.674
Accuracy: 0.674
Accuracy: 0.67
Accuracy: 0.674
Accuracy: 0.662
Accuracy: 0.672
Accuracy: 0.674
Mean value: 0.6712
Standard deviation: 0.0036757463338907 29
```



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.

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 [ ]:

```
#BINARY
#Linear
cl = NumpyLinRegClass()
training_loss, validation_loss, training_accuracy, validation_accuracy = cl.fit(X_train, t2_train, 0.00001, X_val, t2_va
lin_train_acc = cl.accuracy(X_train, t2_train)
lin_t2_acc = cl.accuracy(X_val, t2_val)
lin_test_acc = cl.accuracy(X_test, t2_test)

#Logistic
log_cl = NumpyLogReg()
training_log_loss, validation_log_loss, training_log_accuracy, validation_log_accuracy = log_cl.fit(X_train, t2_train, 
log_train_acc = log_cl.accuracy(X_train, t2_train)
log_t2_acc = log_cl.accuracy(X_val, t2_val)
log_test_acc = log_cl.accuracy(X_test, t2_test)

#MNN
mnn = MNNClassifier()
mnn.fit(X_train, t2_train)
output, mnn_train_acc = mnn.accuracy(X_train, t2_train)
output, mnn_t2_acc = mnn.accuracy(X_val, t2_val)
output, mnn_test_acc = mnn.accuracy(X_test, t2_test)

#MULTI-CLASS
#Logistic
multi_cl = NumpyLogRegMultiClass()
multi_cl.fit(X_train, hot_encoding(t_train))
multi_train_acc = multi_cl.accuracy(X_train, t_train)
multi_t_acc = multi_cl.accuracy(X_val, t_val)
multi_test_acc = multi_cl.accuracy(X_test, t_test)

#MNN
mnn = MNNClassifier()
mnn.fit(X_train, t_train)
output, mnn_train_acc = mnn.accuracy(X_train, t_train)
output, mnn_t_acc = mnn.accuracy(X_val, t_val)
output, mnn_test_acc = mnn.accuracy(X_test, t_test)
```

```python
names = ["linear", "logistic", "mnn"]
binary_train_acc = [lin_train_acc, log_train_acc, mnn_train_acc]
binary_t2_acc = [lin_t2_acc, log_t2_acc, mnn_t2_acc]
binary_test_acc = [lin_test_acc, log_test_acc, mnn_test_acc]

multi_train_acc = ["x", multi_train_acc, mnn_train_acc]
multi_t_acc = ["x", multi_t_acc, mnn_t_acc]
multi_test_acc = ["x", multi_test_acc, mnn_test_acc]

titles = ["classifier", "binary train", "binary validation", "binary test", "multiclass train", "multiclass validation"
data = [titles] + list(zip(names, binary_train_acc, binary_t2_acc, binary_test_acc, multi_train_acc, multi_t_acc, multi_

for i, d in enumerate(data):
    line = '|'.join(str(x).ljust(12) for x in d)
    print(line)
    if i == 0:
        print('-' * len(line))

#Linear: better results for test set than training set, usually we do not expect this.
#Usually test set will have accuracy closer to that which we see in the validation set.
#The training set must therefore be closer to trainin set than validation set.

#Logistic: the binary version gets worse accuracy for test set as expected.
#The OVR classifier gets better results than binary, and I am not sure as to why.
#The OVR gets similiar accuracy for sets.

#MNN: clearly the best and most versatile classifier. We get about equal accuracies for training and testing.
#This classifier works well for both binary and multiclass.
```

```
classifier  |binary train|binary validation|binary test |multiclass train|multiclass validation|multiclass test
----------------------------------------------------------------------------------------------------------------
linear      |0.7         |0.654       |0.716       |x           |x           |x
logistic    |0.464       |0.528       |0.482       |0.567       |0.564       |0.566
mnn         |0.78        |0.674       |0.788       |0.78        |0.738       |0.788
```

```
In [ ]:
```