IN3050/IN4050 Mandatory Assignment 2, 2025: 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. Read also the "Routines for handling suspicion of cheating and attempted cheating at the University of Oslo": https://www.uio.no/english/studies/examinations/cheating/index.html We do not entirely prohibit the use of generative language models ("smart assistants" like ChatGPT, Llama, Claude or Copilot), but you must clearly acknowledge this at all times, following the UiO guidelines: https://www.uio.no/english/studies/resources/ai_student.html Note also that you must fully understand *all* the parts of you submissions, even if you got some help from a generative model. This will be tested during your peer review sessions (https://www.uio.no/studier/emner/matnat/ifi/IN3050/v25/Peer%20review/). By submitting this assignment, you confirm that you are familiar with the rules and the consequences of breaking them.

Delivery

Deadline: Friday, March 28, 2025, 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 regular Python script if you prefer.

Alternative 1

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.

Alternative 2

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.)

Here is a list of *absolutely necessary* (but not sufficient) conditions to get the assignment marked as passed:

- You must deliver your code (Python script or Jupyter notebook) you used to solve the assignment.
- The code used for making the output and plots must be included in the assignment.
- You must include example runs that clearly shows how to run all implemented functions and methods.
- All the code (in notebook cells or python main-blocks) must be runnable. If you have unfinished code that crashes, please comment it out and document what you think causes it to crash.
- You must also deliver a PDF of the code, outputs, comments and plots as explained above.

Your report/notebook should contain your name and username.

Deliver one single compressed 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 neural networks (multi-layer perceptrons, MLP) and the differences and similarities between binary and multi-class classification.

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 machine learning libraries like Scikit-Learn or PyTorch, because the point of this assignment is for you to implement things from scratch. You, however, are encouraged to use tools like NumPy, Pandas and MatPlotLib, which are not ML-specific.

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.

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
import sklearn # This is only to generate a dataset
```

Datasets

We start by making a synthetic dataset of 5000 instances and ten classes, with 500 instances 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. In addition, we would like a dataset with instances represented with only two numerical features, so that it is easy to visualize the data. It would be rather difficult (although not impossible) to find a real-world dataset of the same nature. Anyway, you surely can use the code in this assignment for training machine learning models on real-world datasets.

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-Learn, but that will not be the case with real-world data) We should split the data so that we keep the alignment between X (features) and t (class labels), which may be achieved by shuffling the indices. We split into 60% for training, 20% for validation, and 20% 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(424242)`.

In [2]:
```
# Generating the dataset
from sklearn.datasets import make_blobs
X, t_multi = make_blobs(n_samples=[500, 500, 500, 500, 500, 500, 500, 500
                  n_features=2, random_state=424242, cluster_std=[1.0, 2.
```

In [3]:
```
# Shuffling the dataset
indices = np.arange(X.shape[0])
rng = np.random.RandomState(424242)
```

```
rng.shuffle(indices)
indices[:10]
```

Out[3]:
```
array([3560, 4674,   49, 1257,  661, 3066, 3834, 4792,  570, 3855])
```

In [4]:
```
# Splitting into train, dev and test
X_train = X[indices[:3000],:]
X_val = X[indices[3000:4000],:]
X_test = X[indices[4000:],:]
t_multi_train = t_multi[indices[:3000]]
t_multi_val = t_multi[indices[3000:4000]]
t_multi_test = t_multi[indices[4000:]]
```

Next, we will make a second dataset with only two classes by merging the existing labels in (X,t), so that `0-5` become the new `0` and `6-9` become the new `1`. Let's call the new set (X, t2). This will be a binary set. We now have two datasets:

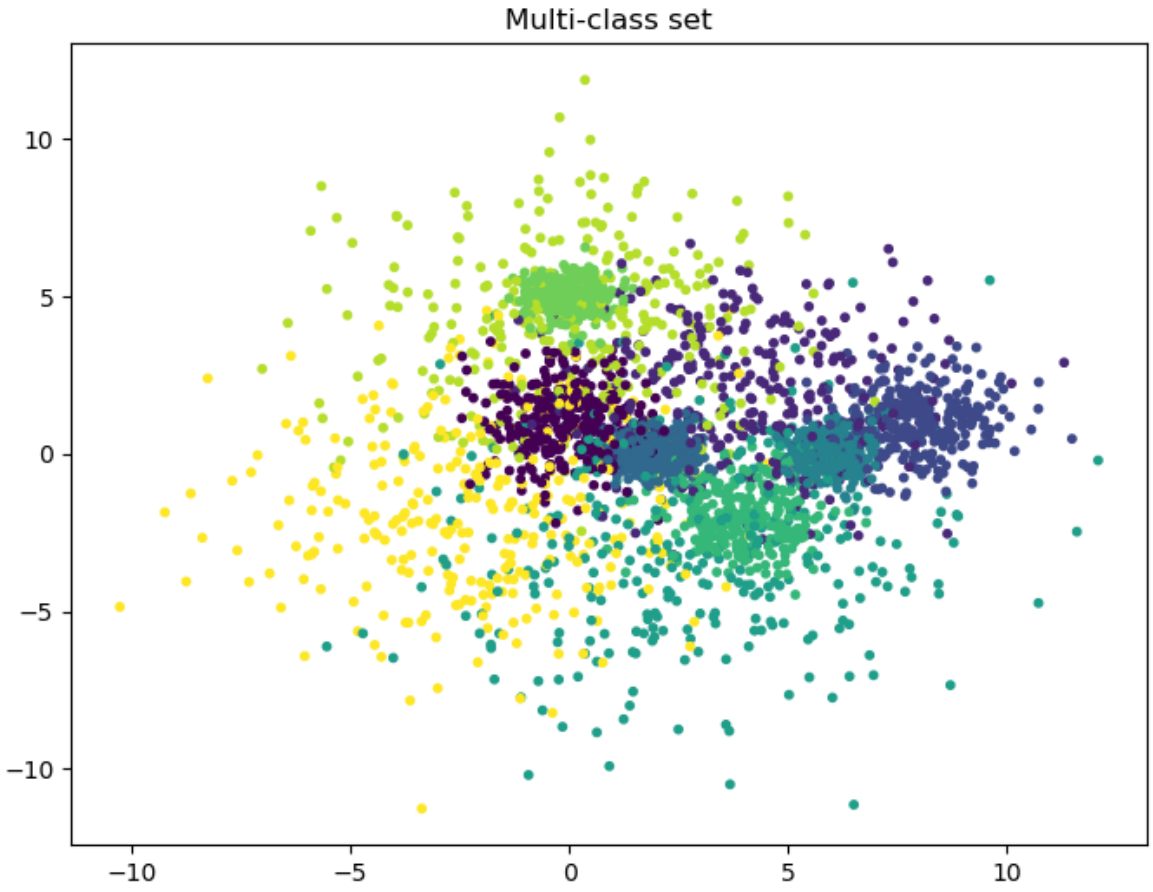
- Binary set: `(X, t2)`
- Multi-class set: `(X, t_multi)`

In [5]:
```
t2_train = t_multi_train >= 6
t2_train = t2_train.astype("int")
t2_val = (t_multi_val >= 6).astype("int")
t2_test = (t_multi_test >= 6).astype("int")
```

We can plot the two traning sets.

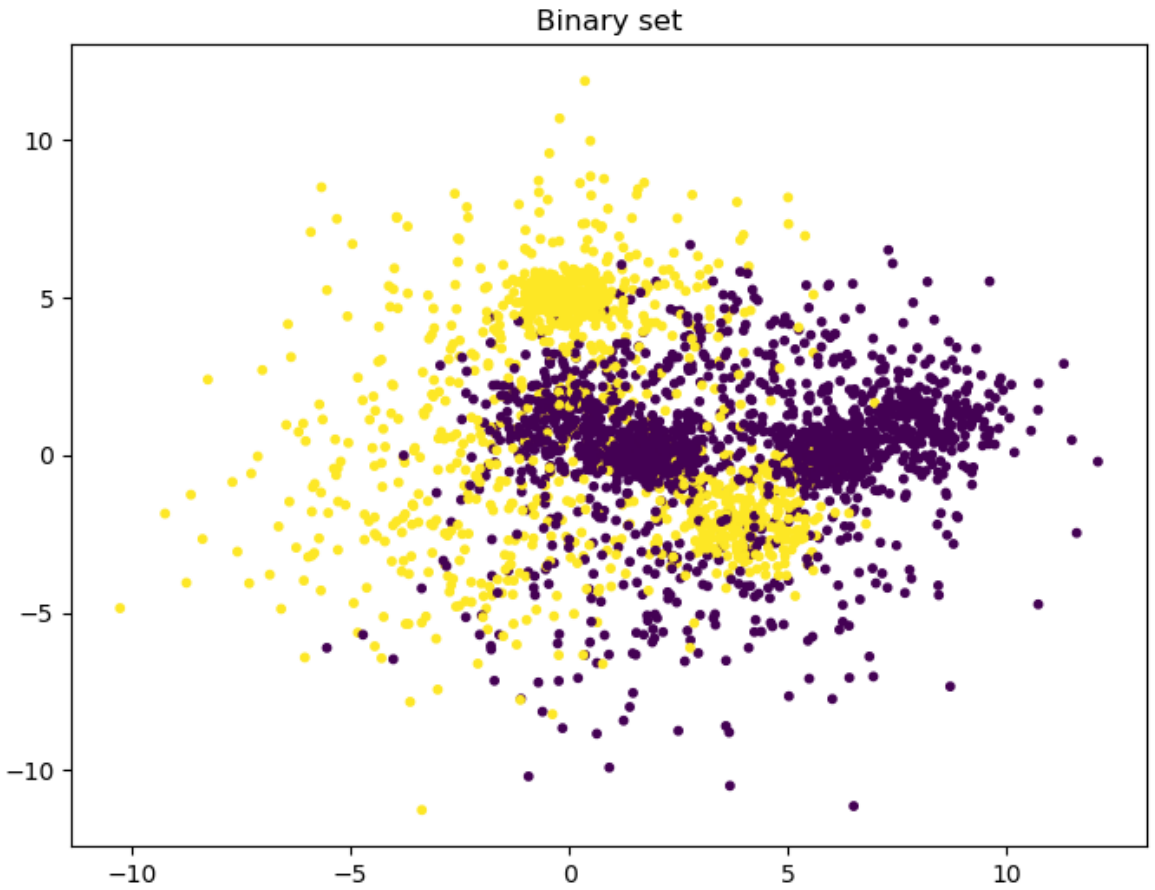
In [6]:
```
plt.figure(figsize=(8,6)) # You may adjust the size
plt.scatter(X_train[:, 0], X_train[:, 1], c=t_multi_train, s=10.0)
plt.title("Multi-class set")
```

Out[6]:
```
Text(0.5, 1.0, 'Multi-class set')
```


Multi-class set
10
5
0
−5
−10
−10
−5
0
5
10

In [7]:
```
plt.figure(figsize=(8,6))
plt.scatter(X_train[:, 0], X_train[:, 1], c=t2_train, s=10.0)
plt.title("Binary set")
```

Out[7]:
```
Text(0.5, 1.0, 'Binary set')
```



Part 1: Linear classifiers

Linear regression

We see that even the binary 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 with the Mean Squared Error (MSE) loss, although it is not the most widely used approach for classification tasks: but we are interested. You may make your own implementation from scratch or start with the solution to the weekly exercise set 6. We include it here with a little added flexibility.

In [8]:
```
def add_bias(X, bias):
    """X is a NxM matrix: N datapoints, M features
    bias is a bias term, -1 or 1, or any other scalar. Use 0 for no bias
    Return a Nx(M+1) matrix with added bias in position zero
    """
    N = X.shape[0]
    biases = np.ones((N, 1)) * bias # Make a N*1 matrix of biases
    # Concatenate the column of biases in front of the columns of X.
    return np.concatenate((biases, X), axis  = 1)
```

In [9]:
```
class NumpyClassifier():
    """Common methods to all Numpy classifiers --- if any"""
```

In [10]:
```
class NumpyLinRegClass(NumpyClassifier):

    def __init__(self, bias=-1):
        self.bias=bias

    def fit(self, X_train, t_train, eta = 0.1, epochs=10):
        """X_train is a NxM matrix, N data points, M features
        t_train is avector of length N,
        the target class values for the training data
        lr is our learning rate
        """

        if self.bias:
            X_train = add_bias(X_train, self.bias)

        (N, M) = X_train.shape

        self.weights = weights = np.zeros(M)

        for epoch in range(epochs):
            # print("Epoch", epoch)
            weights -= eta / N *  X_train.T @ (X_train @ weights - t_trai

    def predict(self, X, threshold=0.5):
        """X is a KxM matrix for some K>=1
        predict the value for each point in X"""
        if self.bias:
            X = add_bias(X, self.bias)
        ys = X @ self.weights
        return ys > threshold
```

We can train and test a first classifier (on the binary dataset).

In [11]:
```
def accuracy(predicted, gold):
    return np.mean(predicted == gold)
```

In [12]:
```
cl = NumpyLinRegClass()


etas = np.linspace(0.01, 1.0, num=100)
epochs = np.arange(5, 205, 5)
# print(etas)
# print(epochs)
max_acc = 0

for eta in etas:
    for epoch in epochs:
        epoch = int(epoch)
        cl.fit(X_train, t2_train, eta=eta, epochs=epoch)
        acc = accuracy(cl.predict(X_val), t2_val)
        # print(acc)
        if acc > max_acc:
            max_acc = acc
            max_eta = eta
            max_epoch = epoch
```

```
            print("NEW MAX")
            print(f"Accuracy {acc} with eta: {eta} and epoch {epoch}")

cl.fit(X_train, t2_train, eta=max_eta, epochs=max_epoch)
```

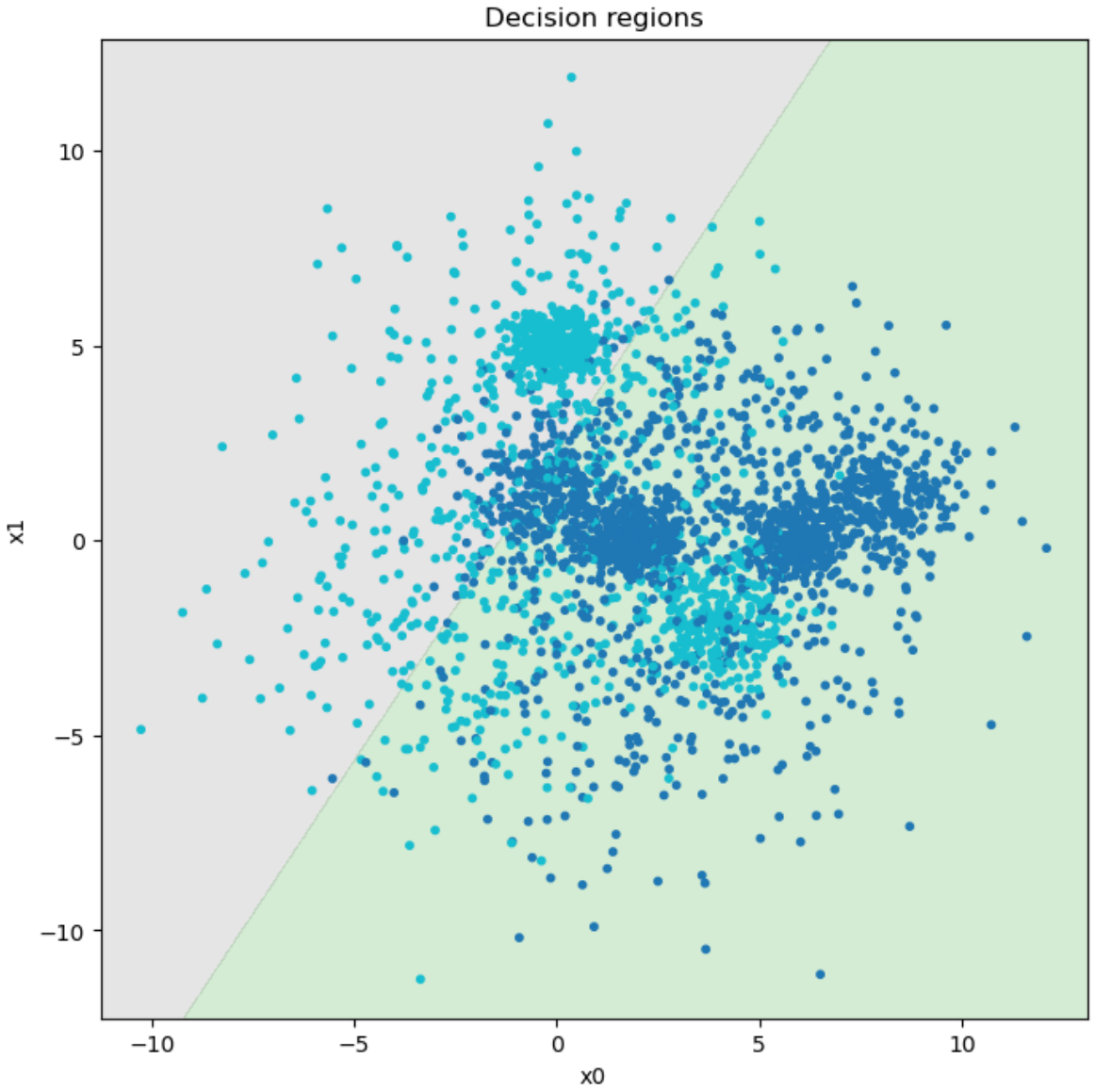
```
NEW MAX
Accuracy 0.597 with eta: 0.01 and epoch 5
NEW MAX
Accuracy 0.599 with eta: 0.01 and epoch 25
NEW MAX
Accuracy 0.602 with eta: 0.01 and epoch 30
NEW MAX
Accuracy 0.605 with eta: 0.01 and epoch 35
NEW MAX
Accuracy 0.608 with eta: 0.01 and epoch 40
NEW MAX
Accuracy 0.612 with eta: 0.01 and epoch 45
NEW MAX
Accuracy 0.613 with eta: 0.01 and epoch 55
NEW MAX
Accuracy 0.614 with eta: 0.01 and epoch 60
NEW MAX
Accuracy 0.617 with eta: 0.01 and epoch 70
NEW MAX
Accuracy 0.618 with eta: 0.01 and epoch 75
NEW MAX
Accuracy 0.619 with eta: 0.01 and epoch 85
NEW MAX
Accuracy 0.621 with eta: 0.01 and epoch 90
NEW MAX
Accuracy 0.626 with eta: 0.01 and epoch 95
NEW MAX
Accuracy 0.634 with eta: 0.01 and epoch 100
NEW MAX
Accuracy 0.639 with eta: 0.01 and epoch 105
NEW MAX
Accuracy 0.645 with eta: 0.01 and epoch 110
NEW MAX
Accuracy 0.654 with eta: 0.01 and epoch 115
NEW MAX
Accuracy 0.661 with eta: 0.01 and epoch 120
NEW MAX
Accuracy 0.674 with eta: 0.01 and epoch 125
NEW MAX
Accuracy 0.683 with eta: 0.01 and epoch 130
NEW MAX
Accuracy 0.693 with eta: 0.01 and epoch 135
NEW MAX
Accuracy 0.702 with eta: 0.01 and epoch 140
NEW MAX
Accuracy 0.709 with eta: 0.01 and epoch 145
NEW MAX
Accuracy 0.717 with eta: 0.01 and epoch 150
NEW MAX
Accuracy 0.725 with eta: 0.01 and epoch 155
NEW MAX
Accuracy 0.733 with eta: 0.01 and epoch 160
NEW MAX
Accuracy 0.738 with eta: 0.01 and epoch 165
NEW MAX
Accuracy 0.744 with eta: 0.01 and epoch 170
NEW MAX
Accuracy 0.751 with eta: 0.01 and epoch 175
NEW MAX
Accuracy 0.754 with eta: 0.01 and epoch 180
```

```
NEW MAX
Accuracy 0.757 with eta: 0.01 and epoch 185
NEW MAX
Accuracy 0.758 with eta: 0.01 and epoch 190
NEW MAX
Accuracy 0.761 with eta: 0.01 and epoch 195
NEW MAX
Accuracy 0.762 with eta: 0.01 and epoch 200
NEW MAX
Accuracy 0.764 with eta: 0.02 and epoch 110
NEW MAX
Accuracy 0.769 with eta: 0.02 and epoch 120
NEW MAX
Accuracy 0.771 with eta: 0.02 and epoch 125
NEW MAX
Accuracy 0.775 with eta: 0.02 and epoch 130
```

The following is a small procedure which plots the data set together with the decision boundaries. You may modify the colors and the rest of the graphics as you like. The procedure will also work for multi-class classifiers

In [13]:
```
def plot_decision_regions(X, t, clf=[], size=(8,8)):
    """Plot the data set (X,t) together with the decision boundary of the
    # The region of the plane to consider determined by X
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    # Make a prediction of the whole region
    h = 0.02  # step size in the mesh
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_m
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    # Classify each meshpoint.
    Z = Z.reshape(xx.shape)

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

    # Put the result into a color plot
    plt.contourf(xx, yy, Z, alpha=0.2, cmap = 'tab10')

    plt.scatter(X[:,0], X[:,1], c=t, s=10.0, cmap='tab10')

    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 [14]:
```
plot_decision_regions(X_train, t2_train, cl)
```



Task: Tuning

The result is far from impressive. Remember that a classifier which always chooses the majority class will have an accuracy of 0.6 on this data set.

Your task is to try various settings for the two training hyper-parameters, learning rate and the number of epochs, to get the best accuracy on the validation set.

Report how the accuracy varies with the hyper-parameter settings. It it not sufficient to give the final hyperparameters. You must also show how you found then and results for alternative values you tried aout.

When you are satisfied with the result, you may plot the decision boundaries, as above.

Answer

eta = learning rate.

After trying out different epochs and etas I discoverd that epochs that are odd has a lower accuracy in general than epochs that are even. With that I tried the epochs from 1 - 200 on eta from 0.01 - 2.0 and found that etas lower than 0.1 gave more accurate answers. The best was an accuracy 0.775 and eta 0.02 and epochs 130.

This might have to do something with how the weights shifts back and forth when going between odd numbers and even numbers where even numbers are better in this case. Also lower eta makes more slight and accurate changes in the right direction.

Task: Scaling

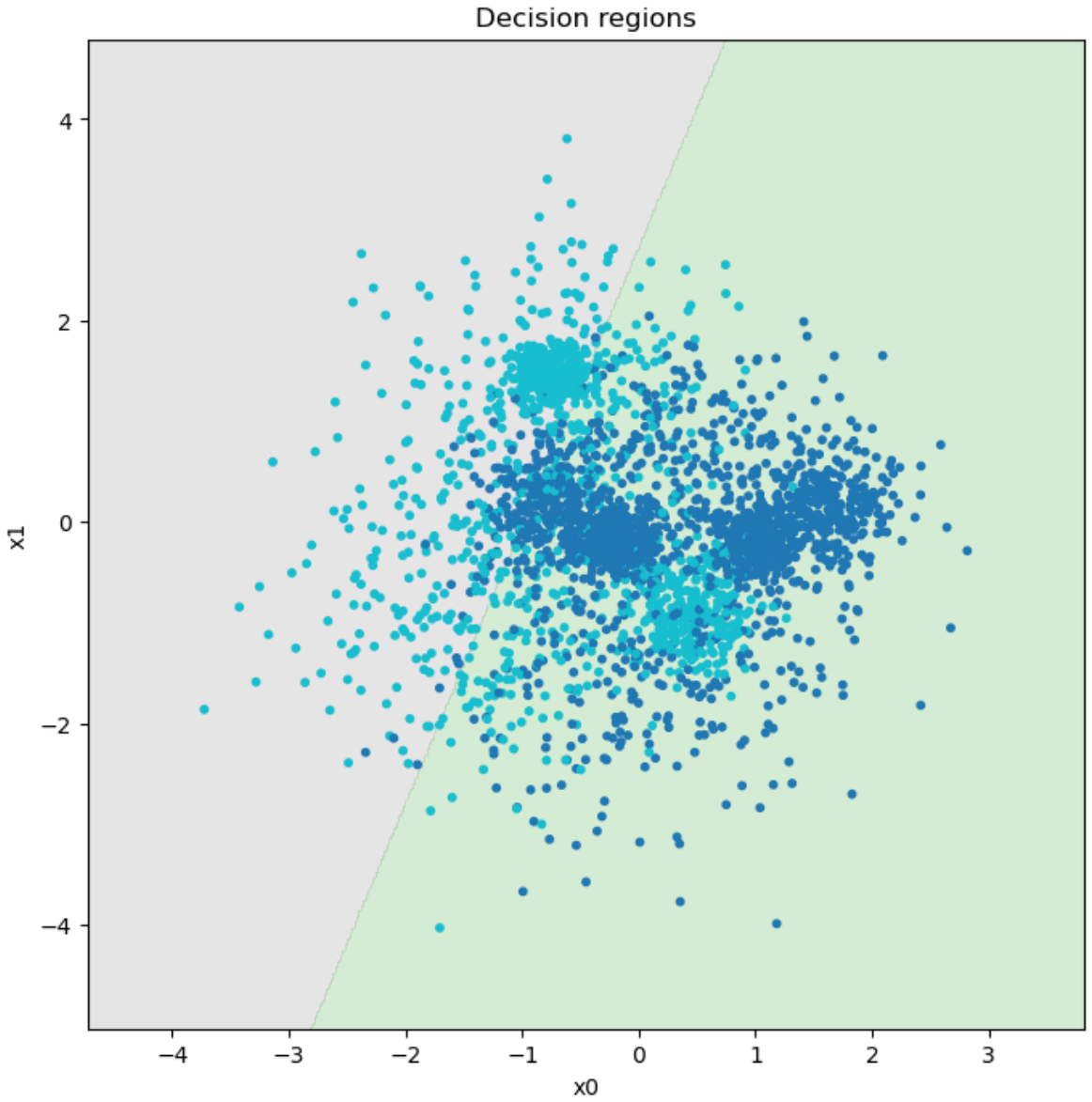
We have seen in the lectures that scaling the data may improve training speed and sometimes the performance.

- Implement the standard scaler (normalizer); you can also try other scaling techniques
- Scale the data
- Train the model on the scaled data
- Experiment with hyper-parameter settings and see whether you can speed up or improve the training.
- Report final hyper-parameter settings and show how you found them.

In [15]:
```
def normalize(X):
    mu = [np.mean(x_i) for x_i in X.T]
    sigma = [np.std(x_i) for x_i in X.T]
    return (X - mu) / sigma

cl = NumpyLinRegClass()

etas = np.linspace(0.01, 1.0, num=100)
epochs = np.arange(5, 205, 5)
# print(etas)
# print(epochs)
max_acc = 0

X_train_scaled = normalize(X_train)
X_val_scaled = normalize(X_val)

for eta in etas:
    for epoch in epochs:
        epoch = int(epoch)
        cl.fit(X_train_scaled, t2_train, eta=eta, epochs=epoch)
        acc = accuracy(cl.predict(X_val_scaled), t2_val)
        # print(acc)
        if acc > max_acc:
            max_acc = acc
            max_eta = eta
            max_epoch = epoch
            print("NEW MAX")
            print(f"Accuracy {acc} with eta: {eta} and epoch {epoch}")

cl.fit(X_train_scaled, t2_train, eta=max_eta, epochs=max_epoch)
```

```
NEW MAX
Accuracy 0.597 with eta: 0.01 and epoch 5
NEW MAX
Accuracy 0.599 with eta: 0.01 and epoch 65
NEW MAX
Accuracy 0.601 with eta: 0.01 and epoch 70
NEW MAX
Accuracy 0.604 with eta: 0.01 and epoch 75
NEW MAX
Accuracy 0.614 with eta: 0.01 and epoch 80
NEW MAX
Accuracy 0.618 with eta: 0.01 and epoch 85
NEW MAX
Accuracy 0.621 with eta: 0.01 and epoch 90
NEW MAX
Accuracy 0.629 with eta: 0.01 and epoch 95
NEW MAX
Accuracy 0.634 with eta: 0.01 and epoch 100
NEW MAX
Accuracy 0.643 with eta: 0.01 and epoch 105
NEW MAX
Accuracy 0.654 with eta: 0.01 and epoch 110
NEW MAX
Accuracy 0.663 with eta: 0.01 and epoch 115
NEW MAX
Accuracy 0.678 with eta: 0.01 and epoch 120
NEW MAX
Accuracy 0.693 with eta: 0.01 and epoch 125
NEW MAX
Accuracy 0.715 with eta: 0.01 and epoch 130
NEW MAX
Accuracy 0.729 with eta: 0.01 and epoch 135
NEW MAX
Accuracy 0.741 with eta: 0.01 and epoch 140
NEW MAX
Accuracy 0.753 with eta: 0.01 and epoch 145
NEW MAX
Accuracy 0.757 with eta: 0.01 and epoch 150
NEW MAX
Accuracy 0.767 with eta: 0.01 and epoch 155
NEW MAX
Accuracy 0.772 with eta: 0.01 and epoch 160
NEW MAX
Accuracy 0.774 with eta: 0.01 and epoch 165
NEW MAX
Accuracy 0.776 with eta: 0.08 and epoch 20
```

In [16]:
```
plot_decision_regions(X_train_scaled, t2_train, cl)
```



Answer

Much like the last task I just tested the classifier on a lot of different variations of eta and epochs and found the best with eta = 0.08 and epoch = 20. Also in the picture you can see that the line has shifted quite a lot due to the different spacing between each point.

Logistic regression

a) You should now implement a logistic regression classifier similarly to the classifier based on linear regression. You may use the code from the solution to weekly exercise set week06.

b) In addition to the method `predict()` which predicts a class for the data, include a method `predict_probabilities()` which predicts the probability of the data belonging to the positive class.

c) 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. The prefered loss for logistic regression is binary cross-entropy, but you can also try mean squared error. The most important is that your implementation of the loss corresponds to your implementation of the gradient descent. Also, calculate and store accuracies after each epoch.

d) 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 and the accuracy for the validation set after each epoch.

e) 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 two keyword arguments, `tol` (tolerance) and `n_epochs_no_update` and stop training when the loss has not improved with more than `tol` after `n_epochs_no_update` (to save compute and potentially avoid over-fitting). A possible default value for `n_epochs_no_update` is 2. Also, add an attribute to the classifier which tells us after fitting how many epochs it was trained for.

f) Train classifiers with various learning rates, and with varying values for `tol` for finding the optimal values. Also consider the effect of scaling the data.

g) After a succesful training, for your best model, plot both training loss and validation loss as functions of the number of epochs in one figure, and both training and validation accuracies as functions of the number of epochs in another figure. Comment on what you see. Are the curves monotone? Is this as expected?

In [17]:
```
class NumpyLogRegClass(NumpyClassifier):

    def __init__(self, bias=-1):
        self.bias=bias
        self.log_func = lambda x: 1/(1+np.exp(-x))

    def fit(self, X_train, t_train, eta = 0.1, epochs = 10, X_val = [], t
        """X_train is a NxM matrix, N data points, M features
        t_train is avector of length N,
        the target class values for the training data
        lr is our learning rate
        """

        if self.bias:
            X_train_bias = add_bias(X_train, self.bias)

        (N, M) = X_train_bias.shape

        self.weights = W = np.zeros(M)
        self.losses = []
        self.accuracies = []
        self.epochs = 0

        if len(X_val) and len(t_val):
```

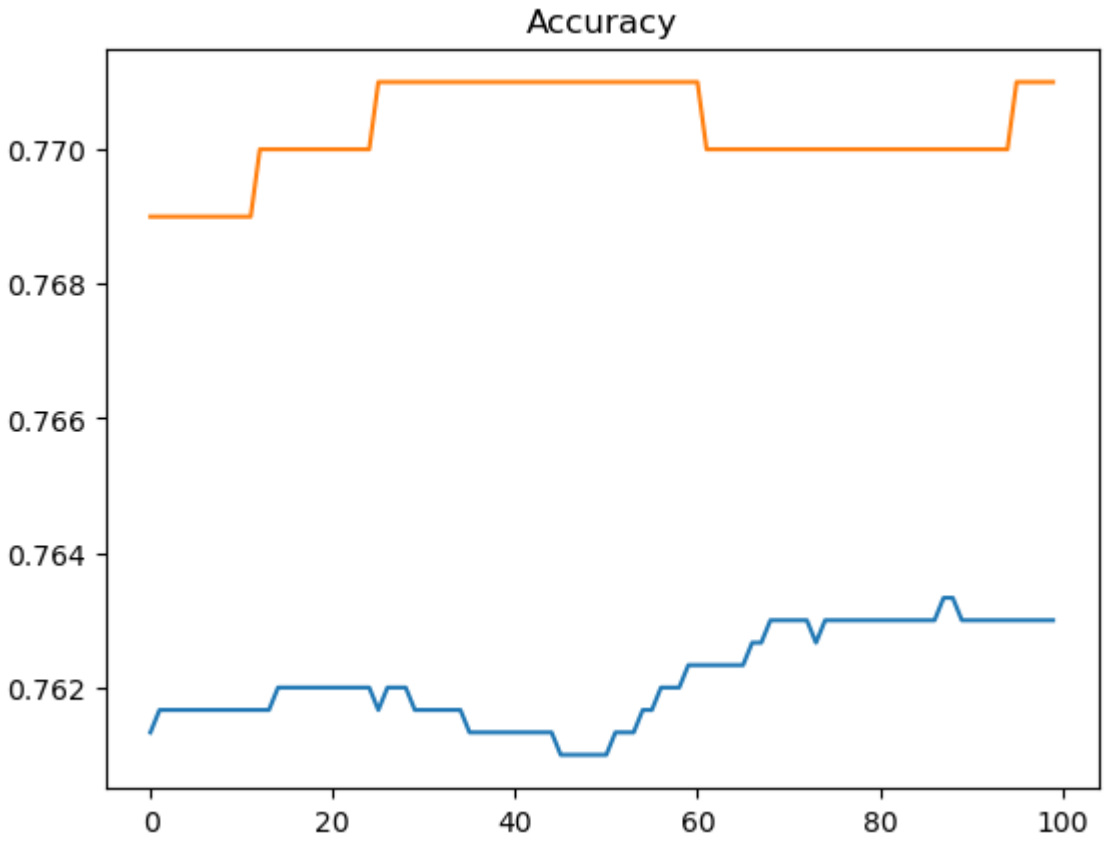
```
            self.losses_val = []
            self.accuracies_val = []

        for epoch in range(epochs):
            # print("Epoch", epoch, end=" ")
            Y = self.log_func(X_train_bias @ W)
            W -= eta / N *  X_train_bias.T @ (Y - t_train)

            # Use X_train, without bias
            loss = self.binary_cross_entropy(X_train, t_train)
            self.losses.append(loss)
            # print("Training Loss:", loss, end=" ")

            acc = accuracy(self.predict(X_train), t_train)
            self.accuracies.append(acc)
            # print("Training Accuracy: ", acc, end=" ")

            if len(X_val) and len(t_val):
                loss = self.binary_cross_entropy(X_val, t_val)
                self.losses_val.append(loss)
                # print("Validation Loss:", loss, end=" ")

                acc = accuracy(self.predict(X_val), t_val)
                self.accuracies_val.append(acc)
                # print("Validation Accuracy:", acc, end=" ")
            # print()

            if n_epochs_no_update and tol:
                if epoch >= n_epochs_no_update and abs(self.losses[-n_epo
                    break

            self.epochs += 1
        if len(X_val) and len(t_val):
            return self.epochs, self.accuracies, self.losses, self.accura
        return self.epochs, self.accuracies, self.losses

    def predict(self, X, threshold=0.5):
        """X is a KxM matrix for some K>=1
        predict the value for each point in X"""
        if self.bias:
            X = add_bias(X, self.bias)
        Y = X @ self.weights
        return self.log_func(Y) > threshold

    def predict_probabilities(self, X):
        if self.bias:
            X = add_bias(X, self.bias)

        Y = X @ self.weights
        return self.log_func(Y)

    def binary_cross_entropy(self, X, t):
        Y = self.predict_probabilities(X)

        P = -np.mean(t * np.log(Y) + (1 - t) * np.log(1 - Y))
        return P
```

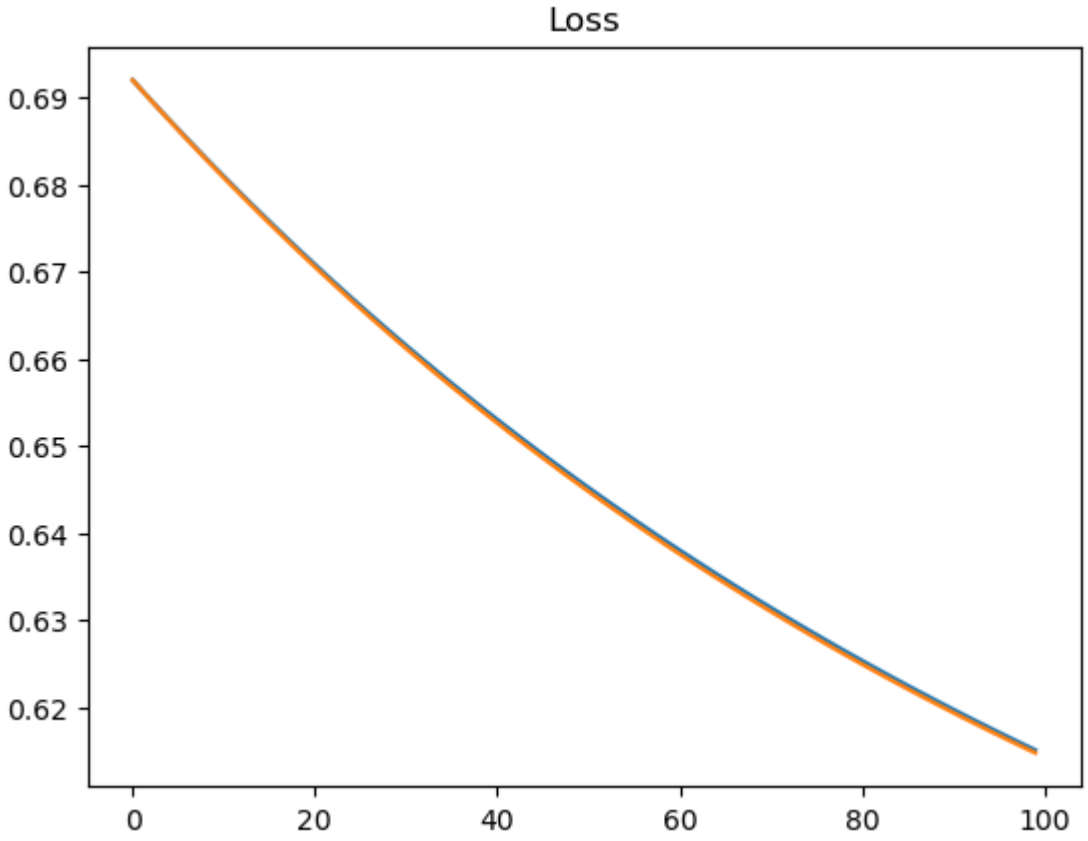
In [45]:
```
lr = NumpyLogRegClass()
```

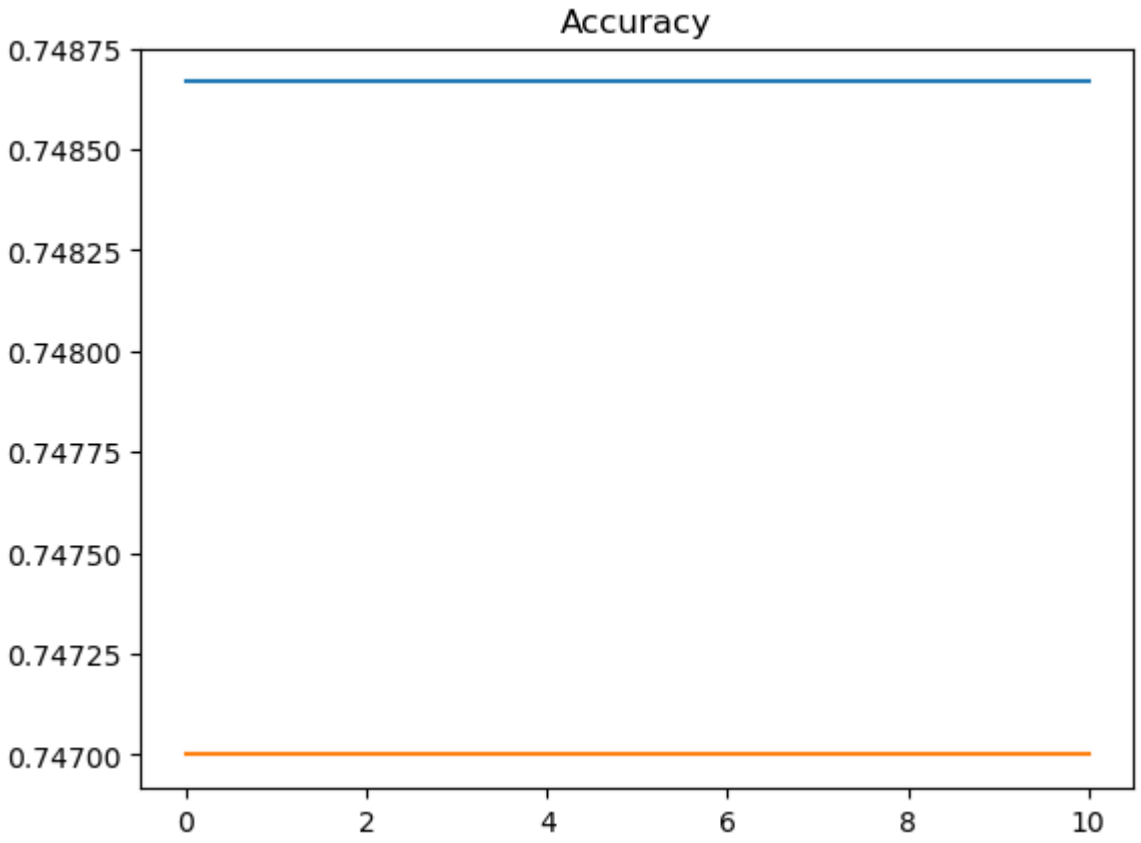
```
X_train_scaled = normalize(X_train)
X_val_scaled = normalize(X_val)

eta = 0.001
epochs = 100
tol = 0.005
n_epochs_no_update = 10
epochs_break, accuracies, losses, accuracies_val, losses_val = lr.fit(X_t

print(f"Accuracy: {accuracies[-1]} Loss: {losses[-1]} Accuracy_val: {accu
plot_decision_regions(X_train, t2_train, lr)
epochs_break += 1
f1 = plt.figure()
# plt.ylim([0.6,0.9])
plt.title("Accuracy")
plt.plot(np.array(range(min(epochs_break, epochs))), accuracies, label =
if len(X_val) and len(t2_val):
    plt.plot(np.array(range(min(epochs_break, epochs))), accuracies_val,

f2 = plt.figure()
# plt.ylim([0.4,0.8])
plt.title("Loss")
plt.plot(np.array(range(min(epochs_break, epochs))), losses, label = "Tra
if len(X_val) and len(t2_val):
    plt.plot(np.array(range(min(epochs_break, epochs))), losses_val, labe


lr = NumpyLogRegClass()
epochs_break, accuracies, losses, accuracies_val, losses_val = lr.fit(X_t

print(f"Accuracy: {accuracies[-1]} Loss: {losses[-1]} Accuracy_val: {accu

epochs_break += 1
f1 = plt.figure()
# plt.ylim([0.6,0.9])
plt.title("Accuracy")
plt.plot(np.array(range(min(epochs_break, epochs))), accuracies, label =
if len(X_val) and len(t2_val):
    plt.plot(np.array(range(min(epochs_break, epochs))), accuracies_val,

f2 = plt.figure()
# plt.ylim([0.4,0.8])
plt.title("Loss")
plt.plot(np.array(range(min(epochs_break, epochs))), losses, label = "Tra
if len(X_val) and len(t2_val):
    plt.plot(np.array(range(min(epochs_break, epochs))), losses_val, labe
```

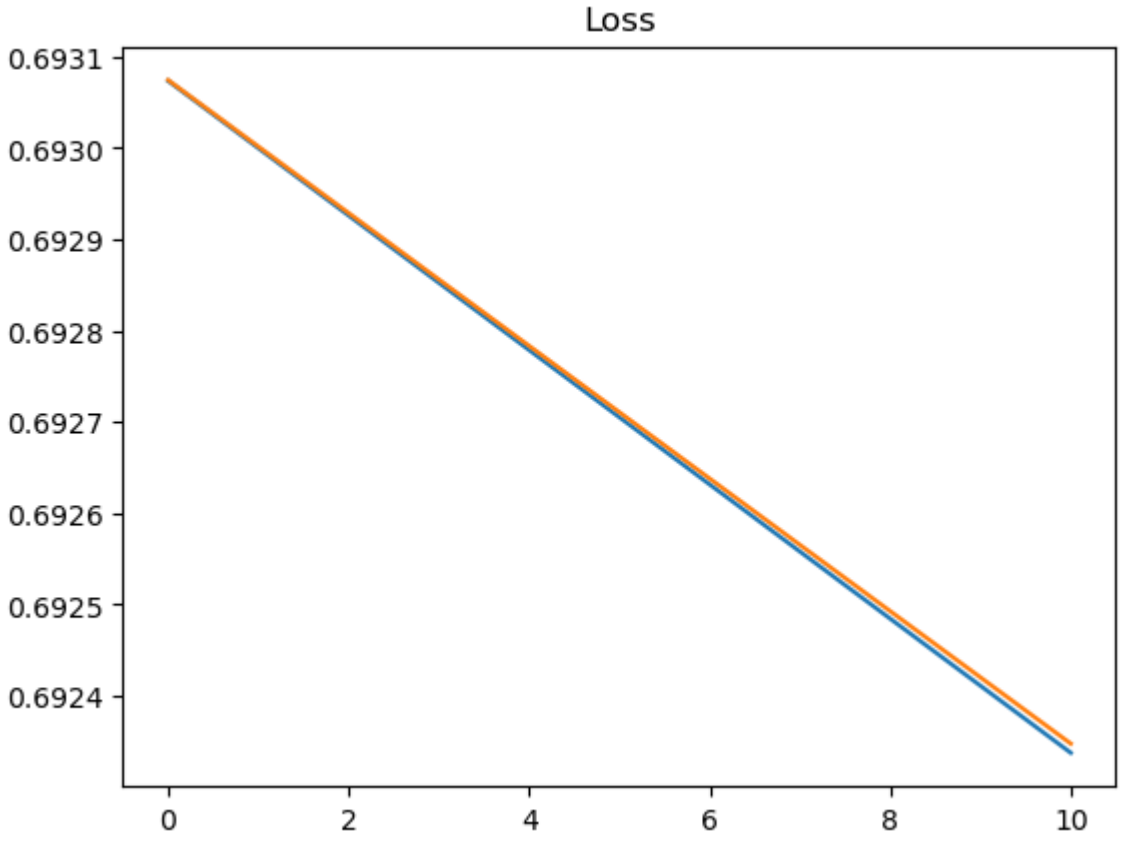
```
Accuracy: 0.763 Loss: 0.6150786792855825 Accuracy_val: 0.771 Epochs: 100
```


```
Accuracy: 0.7486666666666667 Loss: 0.6923374912025982 Accuracy_val: 0.747
Epochs: 10
```


Accuracy
0.770
0.768
0.766
0.764
0.762
0
20
40
60
80
100


Loss
0.69
0.68
0.67
0.66
0.65
0.64
0.63
0.62
0
20
40
60
80
100


Accuracy
0.74875
0.74850
0.74825
0.74800
0.74775
0.74750
0.74725
0.74700
0
2
4
6
8
10


Loss
0.6931
0.6930
0.6929
0.6928
0.6927
0.6926
0.6925
0.6924
0
2
4
6
8
10

Findings

I found the loss results to be monotone however the accuracy is not. The accuracy goes both up and down in the same training. The loss goes down which is unexpected since the accuracy is going both up and down while the loss is not.

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_multi)`.

Multi-class with logistic regression

We saw in the lectures 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_multi_train)`, test it on `(X_val, t_multi_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.

For IN4050 students: Multinomial logistic regression

The following part is only mandatory for IN4050 students. IN3050 students are also welcome to make it a try. Everybody has to do the part 2 on multi-layer neural networks.

In the lectures, 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 classifiers above. (Don't expect a large difference on a simple task like this.)

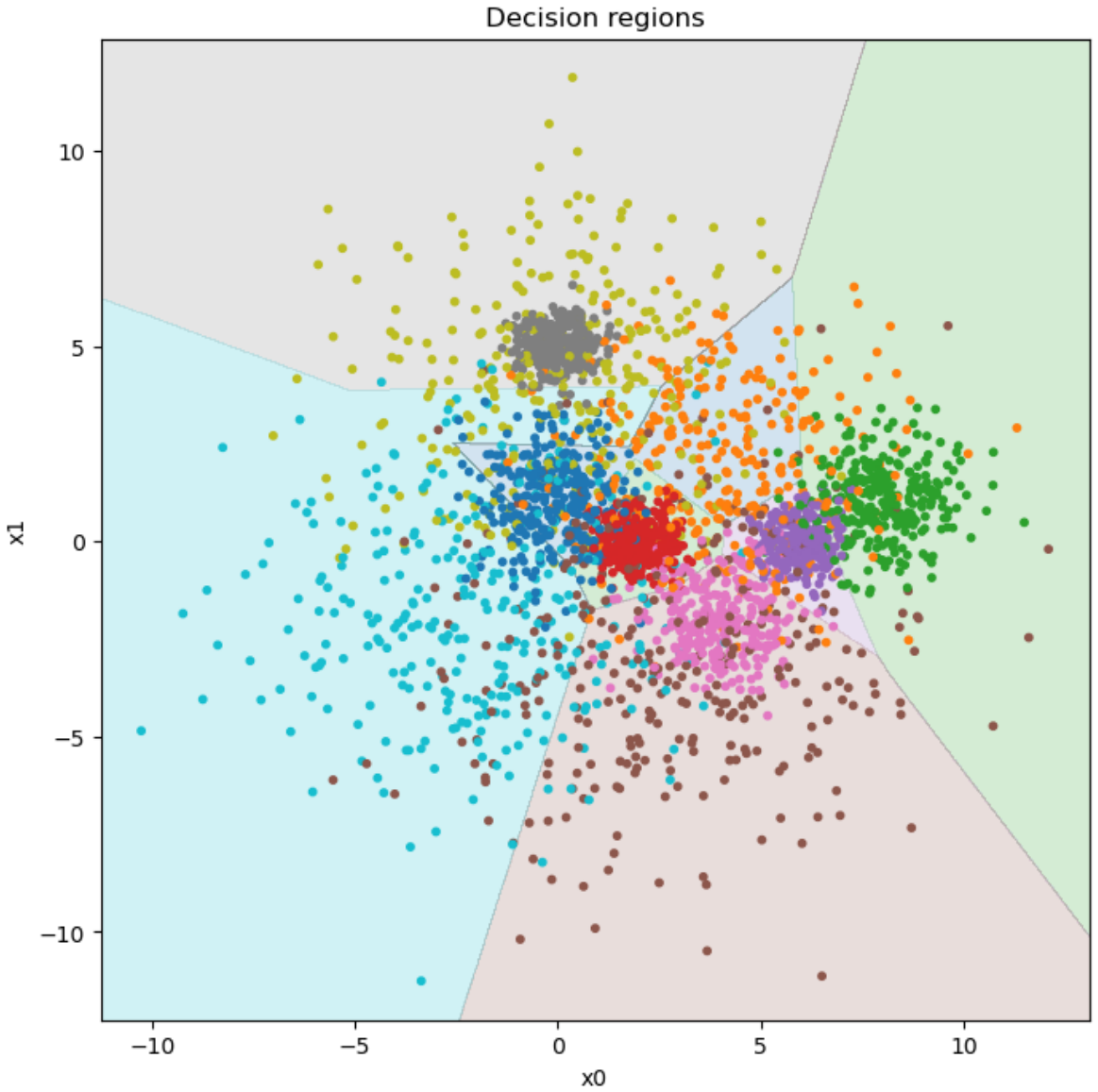
Remember that this classifier uses softmax in the forward phase. For loss, it uses categorical cross-entropy loss. The loss has a somewhat simpler form than in the binary case. To calculate the gradient is a little more complicated. The actual gradient and update rule is simple, however, as long as you have calculated the forward values correctly.

In [19]:
```
class MultiLogRegClass(NumpyClassifier):

    def fit(self, X_train, t_train, eta=0.1, epochs=10, tol=0, n_epochs_n
```

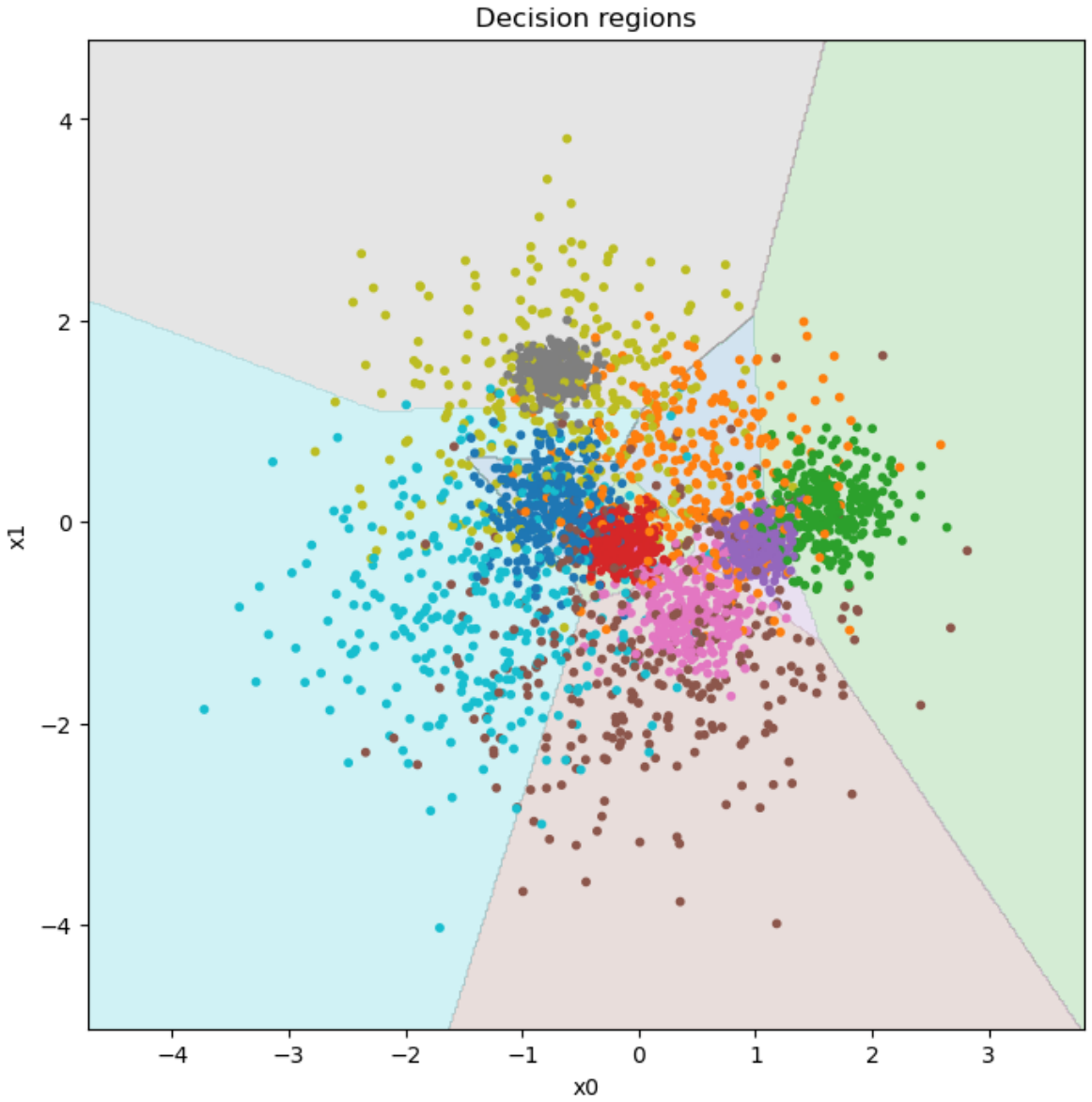
```
        self.classifiers = []

        classes = np.unique(t_train)
        #print(classes)
        for c in classes:
            lrc = NumpyLogRegClass()
            t_class = np.array([1 if t == c else 0 for t in t_train])
            lrc.fit(X_train, t_class, eta=eta, epochs=epochs, tol=tol, n_
            #print(c)

            self.classifiers.append(lrc)

    def predict(self, X):
        # Gpt squisha det til 1 linje, men hadde det orginalt på 3
        return np.argmax(np.array([lrc.predict_probabilities(X) for lrc i


def accuracy(pred, gold):
    return np.mean(pred == gold)
```

In [20]:
```
mlrc = MultiLogRegClass()

eta = 0.62
epochs = 1800
tol = 0.000005
n_epochs_no_update = 10

mlrc.fit(X_train, t_multi_train, eta=eta, epochs=epochs, tol=tol, n_epoch
pred = mlrc.predict(X_val)

print(accuracy(pred, t_multi_val))

plot_decision_regions(X_train, t_multi_train, mlrc)

X_train_scaled = normalize(X_train)
t2_val_scaled = normalize(t2_val)

mlrc = MultiLogRegClass()
mlrc.fit(X_train_scaled, t_multi_train, eta=eta, epochs=epochs, tol=tol,
pred = mlrc.predict(X_val_scaled)

print(accuracy(pred, t_multi_val))

plot_decision_regions(X_train_scaled, t_multi_train, mlrc)
```

```
0.646
```




```
/tmp/ipykernel_11508/3903138766.py:4: RuntimeWarning: invalid value encoun
tered in divide
  return (X - mu) / sigma
0.658
```



Our classifier is not ideal. But is it because all the 10 classes are equally diffucult to predict? You should evaluate the multinomial model predictions for each class separately and report your findings. Please also report whether there is any difference in this respect between the training and the validation sets.

Part 2 Multi-layer neural networks

A first non-linear classifier

The following code is a simple implementation of a multi-layer perceptron or feed-forward neural network. For now, it is quite restricted. There is only one hidden layer with 6 neurons. It can only handle binary classification. In addition, it uses a simple final layer similar to the linear regression classifier above. One way to look at it is what happens when we add a hidden layer to the linear regression classifier.

The MLP class below misses the implementation of the `forward()` function. Your first task is to implement it.

Remember that in the forward pass, we "feed" the input to the model, the model processes it and produces the output. The function should make use of the logistic activation function and bias.

In [21]:
```
# First, we define the logistic function and its derivative:
def logistic(x):
    return 1/(1+np.exp(-x))

def logistic_diff(y):
    return y * (1 - y)
```

In [22]:
```
class MLPBinaryLinRegClass(NumpyClassifier):
    """A multi-layer neural network with one hidden layer"""

    def __init__(self, bias=-1, dim_hidden = 6):
        """Intialize the hyperparameters"""
        self.bias = bias
        # Dimensionality of the hidden layer
        self.dim_hidden = dim_hidden

        self.activ = logistic

        self.activ_diff = logistic_diff

    def forward(self, X):
        """TODO:
        Perform one forward step.
        Return a pair consisting of the outputs of the hidden_layer
        and the outputs on the final layer"""


        hidden_outs = self.activ(add_bias(X @ self.weights1, self.bias))
        outputs = self.activ(hidden_outs @ self.weights2)

        return hidden_outs, outputs

    def fit(self, X_train, t_train, lr=0.02, epochs = 100, X_val = [], t_
        """Initialize the weights. Train *epochs* many epochs.

        X_train is a NxM matrix, N data points, M features
        t_train is a vector of length N of targets values for the trainin
        where the values are 0 or 1.
        lr is the learning rate
        """
        self.lr = lr
        self.epochs = 0

        # Turn t_train into a column vector, a N*1 matrix:
        T_train = t_train.reshape(-1,1)
        if len(t_val) and len(X_val):
            T_val = t_val.reshape(-1,1)
            self.accuracies_val = []
            self.losses_val = []

        self.accuracies = []
        self.losses = []

        dim_in = X_train.shape[1]
```

```
        dim_out = T_train.shape[1]

        # Initialize the weights
        self.weights1 = (np.random.rand(
            dim_in + 1,
            self.dim_hidden) * 2 - 1)/np.sqrt(dim_in)
        self.weights2 = (np.random.rand(
            self.dim_hidden+1,
            dim_out) * 2 - 1)/np.sqrt(self.dim_hidden)
        X_train_bias = add_bias(X_train, self.bias)
        if len(X_val):
            X_val_bias = add_bias(X_val, self.bias)

        for e in range(epochs):
            # print("Epoch: ", e, end=" ")
            # One epoch
            # The forward step:
            hidden_outs, outputs = self.forward(X_train_bias)
            if len(X_val):
                hidden_outs_val, outputs_val = self.forward(X_val_bias)
            # The delta term on the output node:
            out_deltas = (outputs - T_train)
            # The delta terms at the output of the hidden layer:
            hiddenout_diffs = out_deltas @ self.weights2.T
            # The deltas at the input to the hidden layer:
            hiddenact_deltas = (hiddenout_diffs[:, 1:] *
                                self.activ_diff(hidden_outs[:, 1:]))


            # Use X_train, without bias
            loss = self.binary_cross_entropy(outputs, T_train)
            self.losses.append(loss)
            #print("Training Loss:", loss, end=" ")

            acc = accuracy(self.predict(X_train), t_train)
            self.accuracies.append(acc)
            #print("Training Accuracy: ", acc, end=" ")

            if len(X_val) and len(t_val):
                loss = self.binary_cross_entropy(outputs_val, T_val)
                self.losses_val.append(loss)
                #print("Validation Loss:", loss, end=" ")

                acc = accuracy(self.predict(X_val), t_val)
                self.accuracies_val.append(acc)
                #print("Validation Accuracy:", acc, end=" ")
            #print()

            # Update the weights:
            self.weights2 -= self.lr * hidden_outs.T @ out_deltas
            self.weights1 -= self.lr * X_train_bias.T @ hiddenact_deltas


            if n_epochs_no_update and tol:
                if e >= n_epochs_no_update:
                    if abs(self.losses[-n_epochs_no_update - 1] - self.lo
                        break

            self.epochs += 1
        if len(X_val) and len(t_val):
```

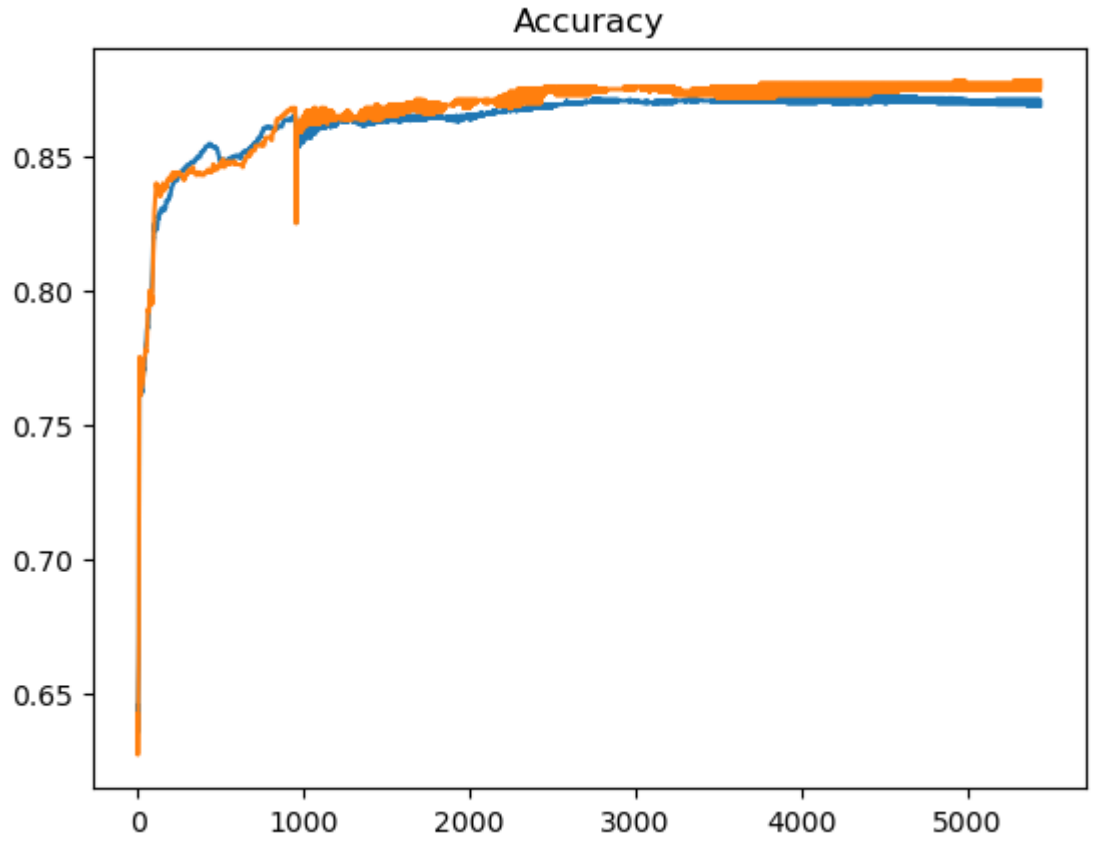
```
            return self.epochs, self.accuracies, self.losses, self.accura
        return self.epochs, self.accuracies, self.losses


    def predict(self, X):
        """Predict the class for the members of X"""
        Z = add_bias(X, self.bias)
        forw = self.forward(Z)[1]
        score= forw[:, 0]
        return (score > 0.5)

    def predict_probabilities(self, X):
        if self.bias:
            X = add_bias(X, self.bias)

        return self.forward(X)[1][0:, 0]

    def binary_cross_entropy(self, Y, t):

        P = -np.mean(t * np.log(Y) + (1 - t) * np.log(1 - Y))
        return P
```

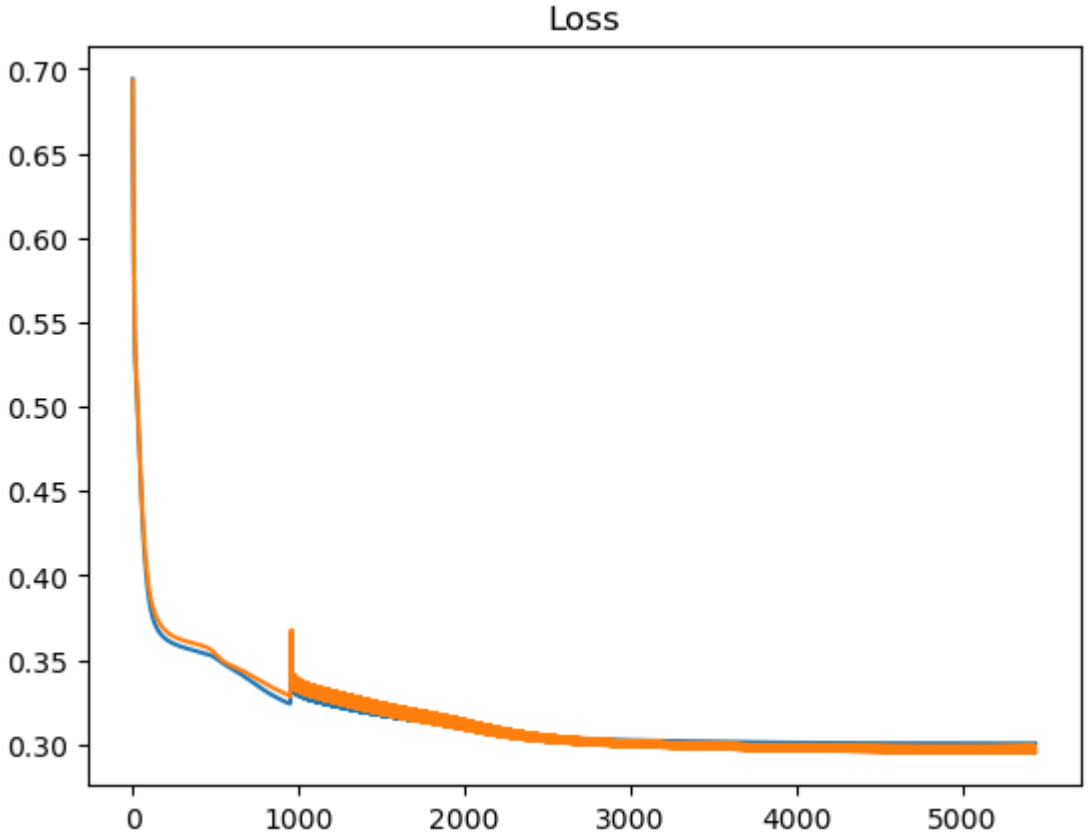
In [41]:

```
dim_hidden = 10

mln = MLPBinaryLinRegClass(dim_hidden = dim_hidden)

lr = 0.0005
epochs = 10000
tol = 0.000001
n_epochs_no_update = 10

epochs_break, accuracies, losses, accuracies_val, losses_val = mln.fit(X_
pred = mln.predict(X_val)
acc = accuracy(pred, t2_val)
print("Accuracy:", acc, "Epcohs:", epochs_break)
plot_decision_regions(X_val, t2_val, mln)

epochs_break += 1
f1 = plt.figure()
# plt.ylim([0.6,0.9])
plt.title("Accuracy")
plt.plot(np.array(range(min(epochs_break, epochs))), accuracies, label =
if len(X_val) and len(t2_val):
    plt.plot(np.array(range(min(epochs_break, epochs))), accuracies_val,

f2 = plt.figure()
# plt.ylim([0.4,0.8])
plt.title("Loss")
plt.plot(np.array(range(min(epochs_break, epochs))), losses, label = "Tra
if len(X_val) and len(t2_val):
    plt.plot(np.array(range(min(epochs_break, epochs))), losses_val, labe
```

```
Accuracy: 0.875 Epcohs: 5438
```


Decision regions
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
x1
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x0


Accuracy
0.85
0.80
0.75
0.70
0.65
0
1000
2000
3000
4000
5000



In [42]:
```
accs = []
for _ in range(3):
    mln = MLPBinaryLinRegClass(dim_hidden = dim_hidden)
    mln.fit(X_train, t2_train, lr=lr, epochs=epochs, X_val=X_val, t_val =
    pred = mln.predict(X_val)
    acc = accuracy(pred, t2_val)
    accs.append(acc)
    print("Accuracy:", acc)

print(accs)
print("Mean: ", np.mean(accs))
print("Standard deviation: ", np.std(accs))
```

```
Accuracy: 0.879
Accuracy: 0.876
Accuracy: 0.873
[0.879, 0.876, 0.873]
Mean:  0.876
Standard deviation:  0.00244948974278318
```

When implemented, this model can be used to make a non-linear classifier for the set `(X, t2)`. Experiment with settings for learning rate and epochs and see how good results you can get. Report results for various settings. Be prepared to train for a long time (but you can control it via the number of epochs and hidden size).

Plot the training set together with the decision regions as in Part I.

Improving the MLP classifier

You should now make changes to the classifier similarly to what you did with the

logistic regression classifier in part 1.

a) In addition to the `predict()` method, which predicts a class for the data, include the `predict_probabilities()` method which predict the probability of the data belonging to the positive class. The training should be based on these values, as with logistic regression.

b) Calculate the loss and the accuracy after each epoch and store them for inspection after training.

c) 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 and the accuracy for the validation set after each epoch.

d) Extend the `fit()` method with two keyword arguments, `tol` (tolerance) and `n_epochs_no_update` and stop training when the loss has not improved for more than `tol` after `n_epochs_no_update`. A possible default value for `n_epochs_no_update` is 2. Add an attribute to the classifier which tells us after fitting how many epochs it was trained on.

e) Tune the hyper-parameters: `lr`, `tol` and `dim-hidden` (size of the hidden layer). Also, consider the effect of scaling the data.

f) After a succesful training with the best setting for the hyper-parameters, plot both training loss and validation loss as functions of the number of epochs in one figure, and both training and validation accuracies as functions of the number of epochs in another figure. Comment on what you see.

g) The MLP algorithm contains an element of non-determinism. Hence, train the classifier 3 times with the optimal hyper-parameters and report the mean and standard deviation of the accuracies over the 3 runs.

For IN4050-students: Multi-class neural network

The following part is only mandatory for IN4050 students. IN3050 students are also welcome to make it a try. This is the most fun part of the set :))

The goal is to use a feed-forward neural network for non-linear multi-class classfication and apply it to the set `(X, t_multi)`.

Modify the network to become a multi-class multinomial classifier. As a sanity check of your implementation, you may apply it to `(X, t_2)` and see whether you get similar results as above.

Train the resulting classifier on `(X_train, t_multi_train)`, test it on `(X_val, t_multi_val)`, tune the hyper-parameters and report the accuracy.

Plot the decision boundaries for your best classifier. Evaluate the best model

predictions for each class separately (same as in part 1) and report your findings. Please also report whether there is any difference in this respect between the training and the validation sets.

Part III: Final testing

We can now perform a final testing on the held-out test set we created in the beginning.

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 the performance in a 3 by 3 table.

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

Also report precision and recall for class 1.

In [48]:
```
linreg = NumpyLinRegClass()

eta = 0.08
epochs = 100
tol = 0.00001
n_epochs_no_update = 2

linreg.fit(X_train, t2_train, eta=0.08, epochs=20)

pred_linreg_train = linreg.predict(normalize(X_train))
pred_linreg_val = linreg.predict(normalize(X_val))
pred_linreg_test = linreg.predict(normalize(X_test))

acc_linreg_train = accuracy(pred_linreg_train, t2_train)
acc_linreg_val = accuracy(pred_linreg_val, t2_val)
acc_linreg_test = accuracy(pred_linreg_test, t2_test)
print("LinReg Train Accuracy:", acc_linreg_train)
print("LinReg Val Accuracy:", acc_linreg_val)
print("LinReg Test Accuracy:", acc_linreg_test)
print()

logreg = NumpyLogRegClass()

eta = 0.001
epochs = 100
tol = 0.005
n_epochs_no_update = 10
logreg.fit(X_train, t2_train, eta=eta, epochs=epochs, tol=tol, n_epochs_n
```

```
pred_logreg_train = logreg.predict(X_train)
pred_logreg_val = logreg.predict(X_val)
pred_logreg_test = logreg.predict(X_test)

acc_logreg_train = accuracy(pred_logreg_train, t2_train)
acc_logreg_val = accuracy(pred_logreg_val, t2_val)
acc_logreg_test = accuracy(pred_logreg_test, t2_test)
print("LogReg Train Accuracy:", acc_logreg_train)
print("LogReg Val Accuracy:", acc_logreg_val)
print("LogReg Test Accuracy:", acc_logreg_test)
print()

dim_hidden = 10
multin = MLPBinaryLinRegClass(dim_hidden=dim_hidden)

lr = 0.0005
epochs = 10000
tol = 0.000001
n_epochs_no_update = 10

multin.fit(X_train, t2_train, lr=eta, epochs=epochs, tol=tol, n_epochs_no

pred_multin_train = multin.predict(X_train)
pred_multin_val = multin.predict(X_val)
pred_multin_test = multin.predict(X_test)

acc_multin_train = accuracy(pred_multin_train, t2_train)
acc_multin_val = accuracy(pred_multin_val, t2_val)
acc_multin_test = accuracy(pred_multin_test, t2_test)
print("Multin Train Accuracy:", acc_multin_train)
print("Multin Val Accuracy:", acc_multin_val)
print("Multin Test Accuracy:", acc_multin_test)
print()
```

```
LinReg Train Accuracy: 0.608
LinReg Val Accuracy: 0.598
LinReg Test Accuracy: 0.589

LogReg Train Accuracy: 0.763
LogReg Val Accuracy: 0.771
LogReg Test Accuracy: 0.764

Multin Train Accuracy: 0.8716666666666667
Multin Val Accuracy: 0.881
Multin Test Accuracy: 0.857
```

Results

	LinReg	LogReg	Multin
X_val	0.608	0.763	0.872
X_test	0.598	0.771	0.881
X_train	0.589	0.764	0.857

In general the test data has worse accuracy. This might be because of overfitting for

the training set. In addition the multin scored best, but had the slowest time. Multin is more flexible and LogReg is also a bit more flexible, while LinReg is just a straight line.

For IN4050-students: Multi-class task (X, t_multi)

The following part is only mandatory for IN4050-students. IN3050-students are also welcome to give it a try though.

Compare the three multi-class classifiers: the standard multi-class and the multinomial logistic regression from part one and the multi-class neural network from part two. Evaluate on test, validation and training set as above.

Comment on the results.