IN3050/IN4050 Mandatory Assignment 2, 2023: 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

By submitting this assignment, you confirm that you are familiar with the rules and the consequences of breaking them.

Delivery

Deadline: Friday, March 24, 2023, 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.

Alternative 1

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.)

Alternative 2

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.

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

- You must deliver your code (python file or 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 run. 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 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
```

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_multi = make_blobs(n_samples=[400,400,400, 400, 400],
                        centers=[[0,1],[4,2],[8,1],[2,0],[6,0]],
                        cluster_std=[1.0, 2.0, 1.0, 0.5, 0.5],
                        n_features=2, random_state=2022)
#Her lager vi clusterene med til sammen 2000
```

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_multi_train = t_multi[indices[:1000]]
t_multi_val = t_multi[indices[1000:1500]]
t_multi_test = t_multi[indices[1500:]]
```

Next, we will make a second dataset by merging classes in (X,t) into two classes and call the new set (X, t2). This will be a binary set. We now have two datasets:

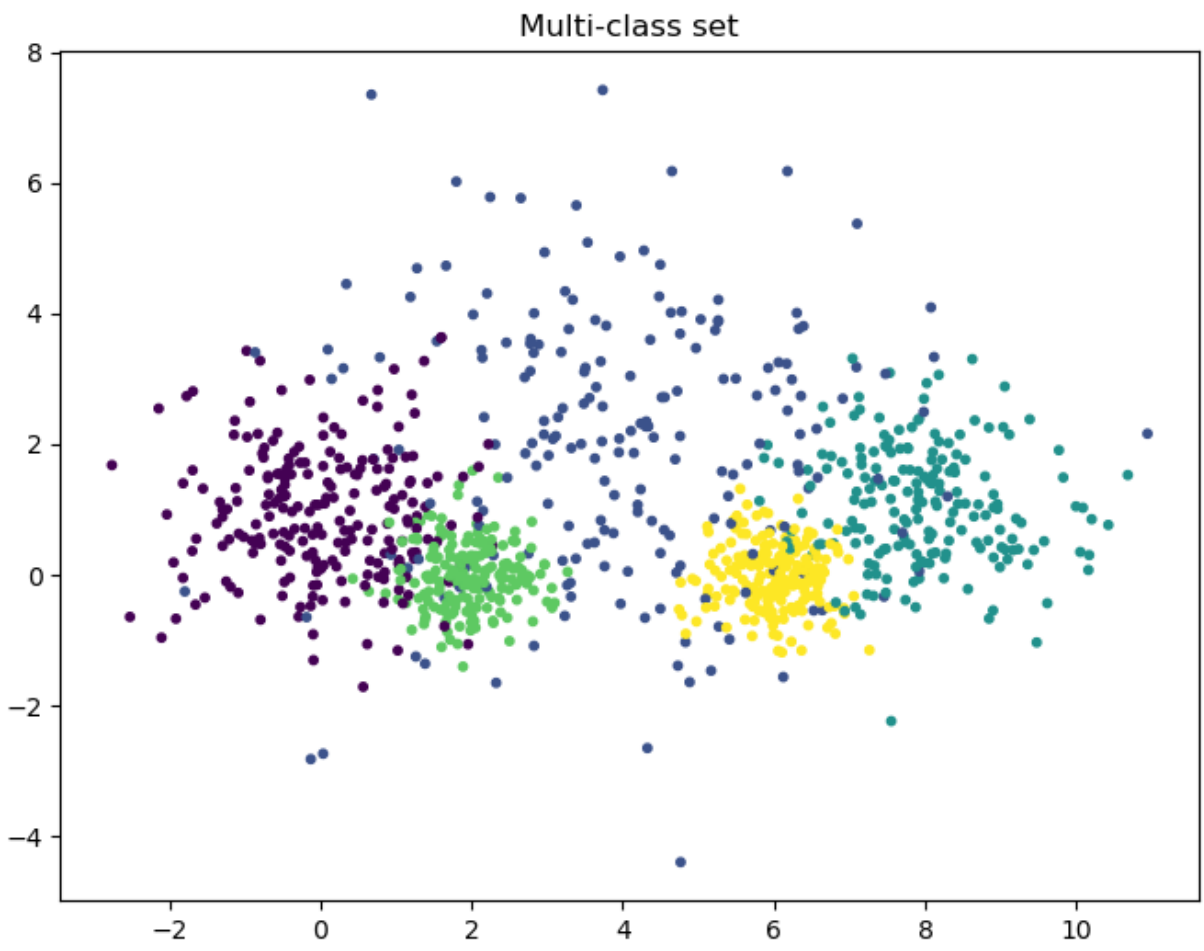
- Binary set: `(X, t2)`
- Multi-class set: `(X, t_multi)`

In [ ]:
```
t2_train = t_multi_train >= 3
t2_train = t2_train.astype('int')
t2_val = (t_multi_val >= 3).astype('int')
t2_test = (t_multi_test >= 3).astype('int')
```

We can plot the two traning sets.

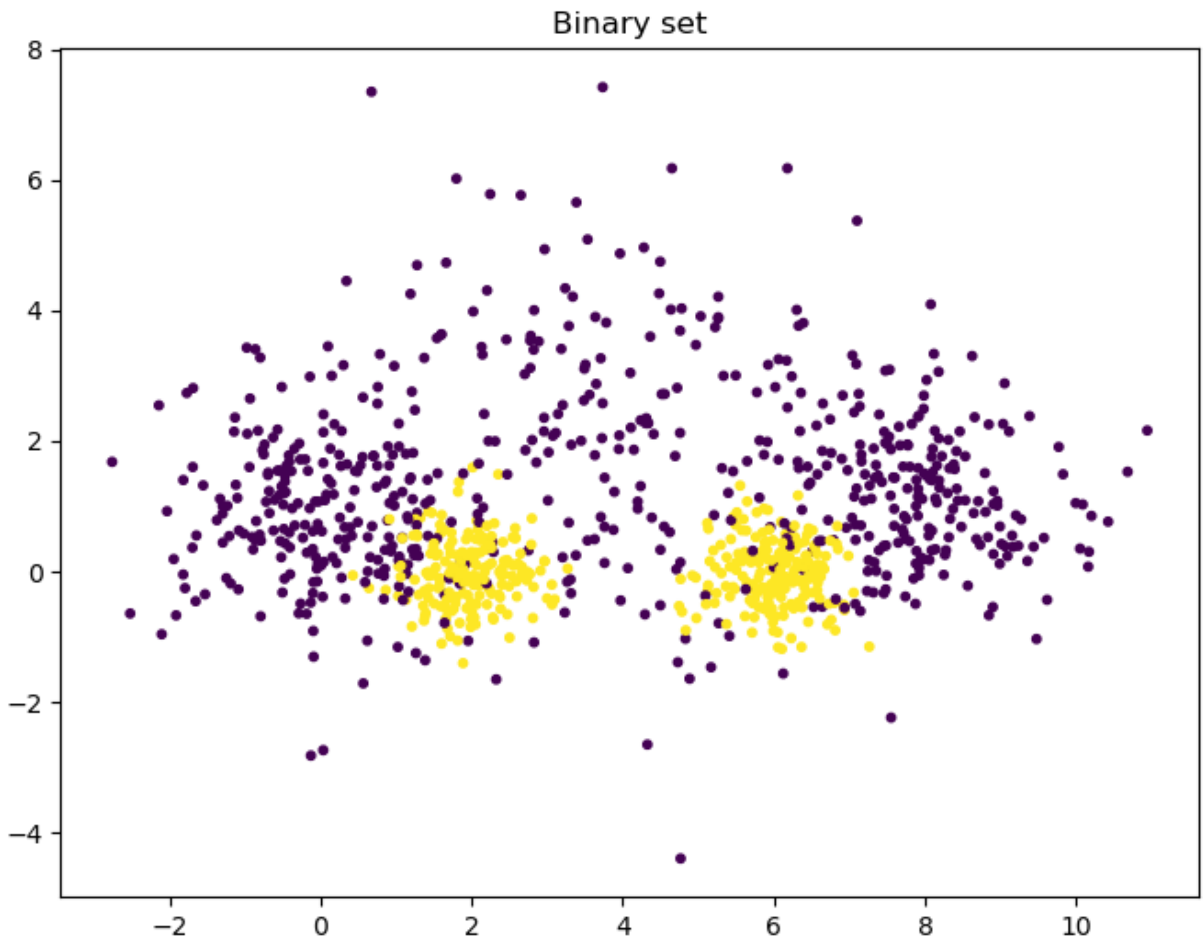
In [ ]:
```
plt.figure(figsize=(8,6)) # You may adjust the size
plt.scatter(X_train[:, 0], X_train[:, 1], c=t_multi_train, s=10.0)
plt.title('Multi-class set')
```

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



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

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



Part I: Linear 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. We include it here with a little added flexibility.

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

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

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

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

    def fit(self, X_train, t_train, eta = 0.07, epochs=200):
        """X_train is a Nxm matrix, N data points, m features
        t_train is avector of length N,
        the targets values for the training data"""

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

        (N, m) = X_train.shape

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

        for e in range(epochs):
            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.

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

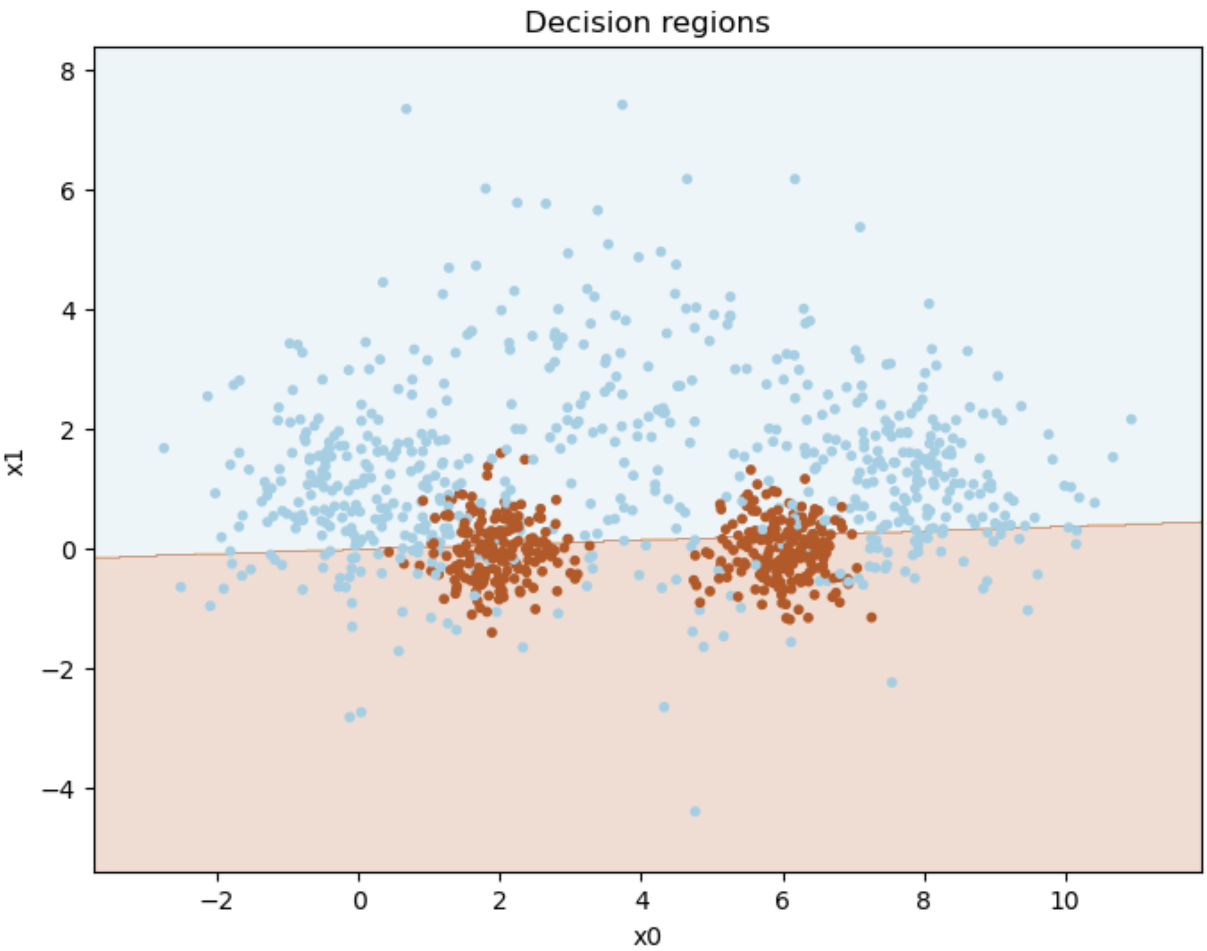
In [ ]:
```
cl = NumpyLinRegClass()
cl.fit(X_train, t2_train)
accuracy(cl.predict(X_val), t2_val)
```

Out[ ]:
```
0.704
```

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 [ ]:
```
def plot_decision_regions(X, t, clf=[], size=(8,6)):
    """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 make 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 = 'Paired')

    plt.scatter(X[:,0], X[:,1], c=t, s=10.0, 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)
```



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, *eta* and *epochs*, to get the best accuracy on the validation set.

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

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

Task: Loss

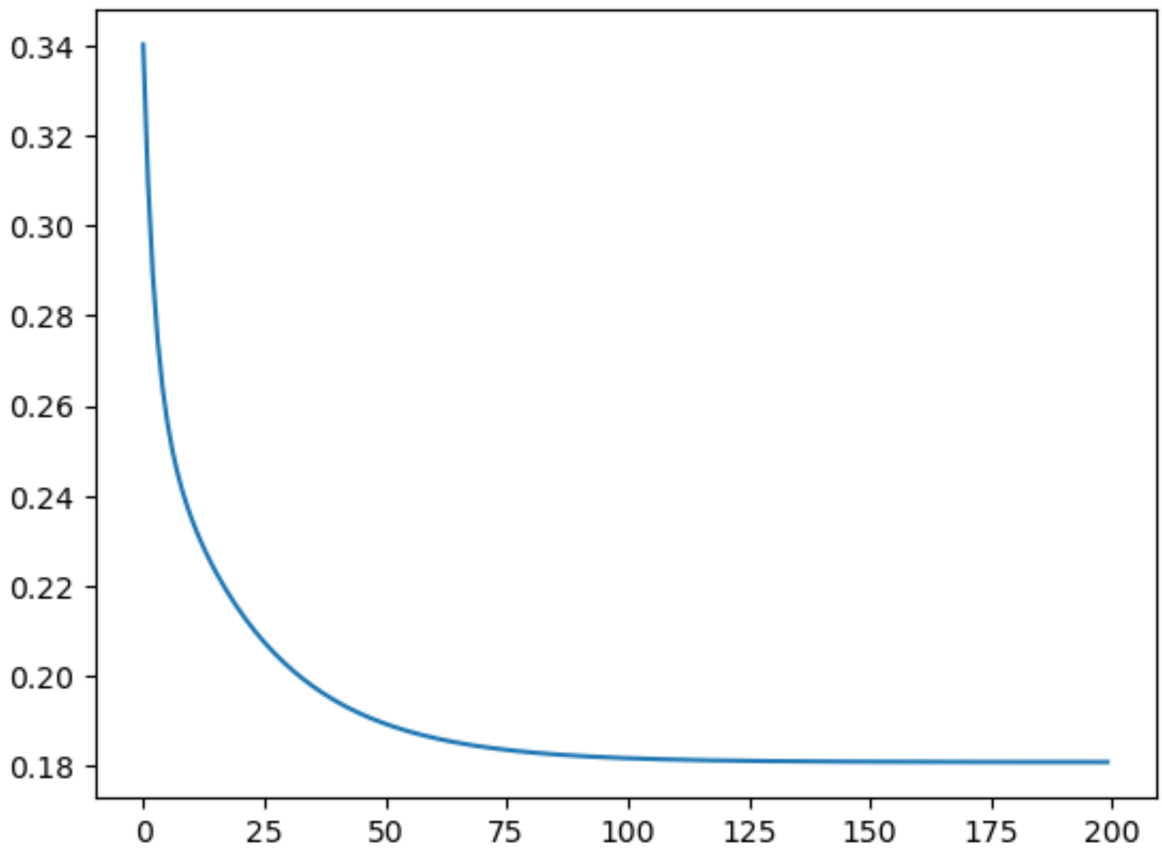
The linear regression classifier is trained with mean squared error loss. So far, we have not calculated the loss explicitly in the code. Extend the code to calculate the loss on the training set for each epoch and to store the losses such that the losses can be inspected after training.

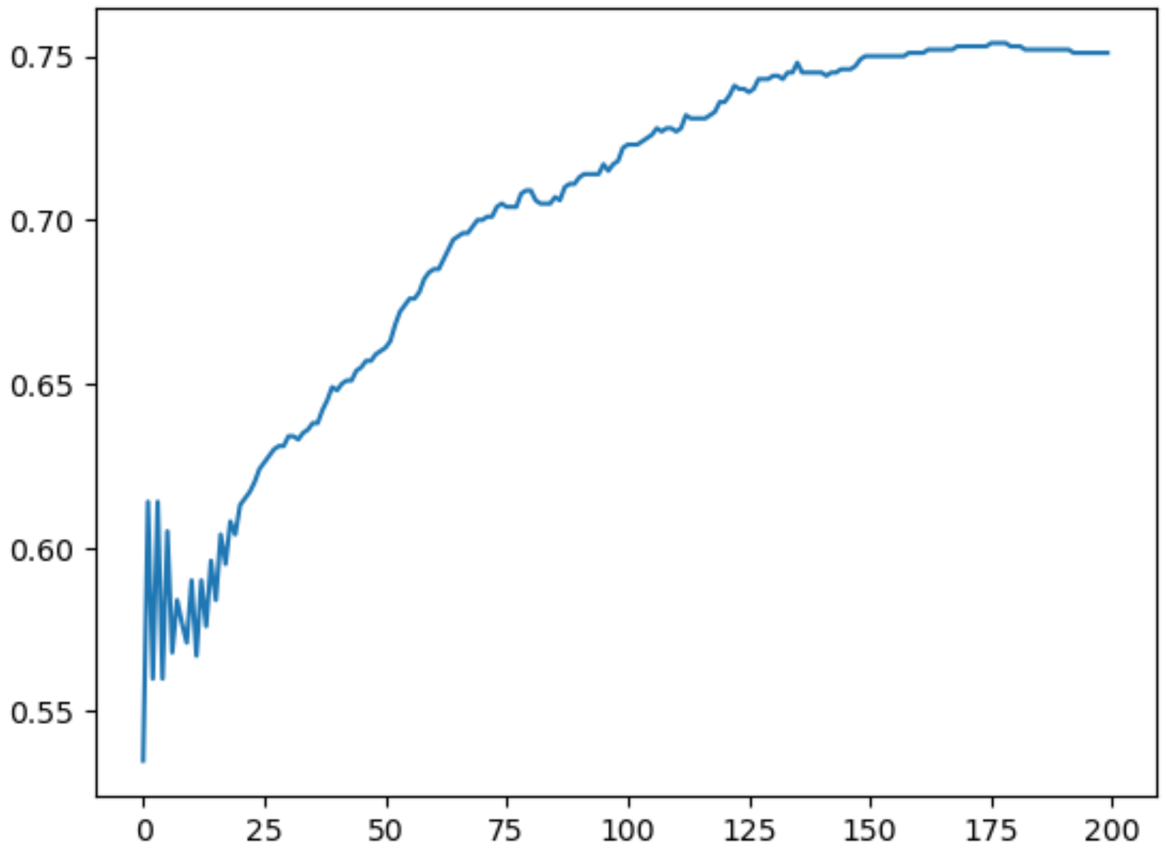
Also extend the classifier to calculate the accuracy on the training data after each epoch.

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

Comment on what you see: Are the function monotone? Is this as expected?

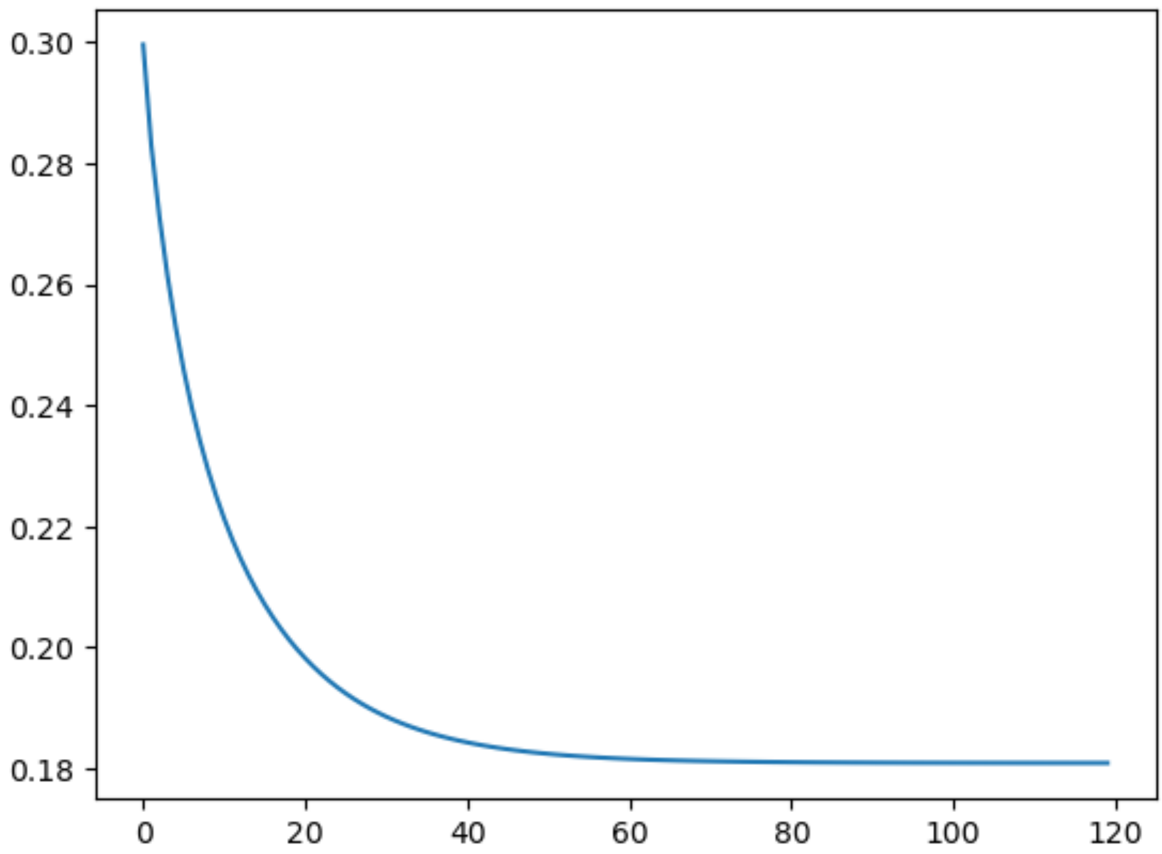
In [ ]:
```
class NumpyLinRegClass(NumpyClassifier):
    lossList = []
    accList = []

    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 targets values for the training data"""

        if self.bias:
            X_train2 = add_bias(X_train, self.bias)

        (N, m) = X_train2.shape

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

        self.lossList = []
        self.accList = []

        for e in range(epochs):
            weights -= eta / N *  X_train2.T @ (X_train2 @ weights - t_tr
            self.lossList.append(self.mse(t_train, X_train2 @ weights))
            self.accList.append(accuracy(self.predict(X_train), t_train))

    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

    def plot_data(self):
        plt.plot(self.lossList)
        plt.show()
        plt.plot(self.accList)
        plt.show()

    def mse(self, y, y_pred):
        return np.mean((y - y_pred)**2)

run = NumpyLinRegClass()
x_train1 = np.copy(X_train)

run.fit(x_train1, t2_train, eta=0.07, epochs=200)
run.plot_data()
```

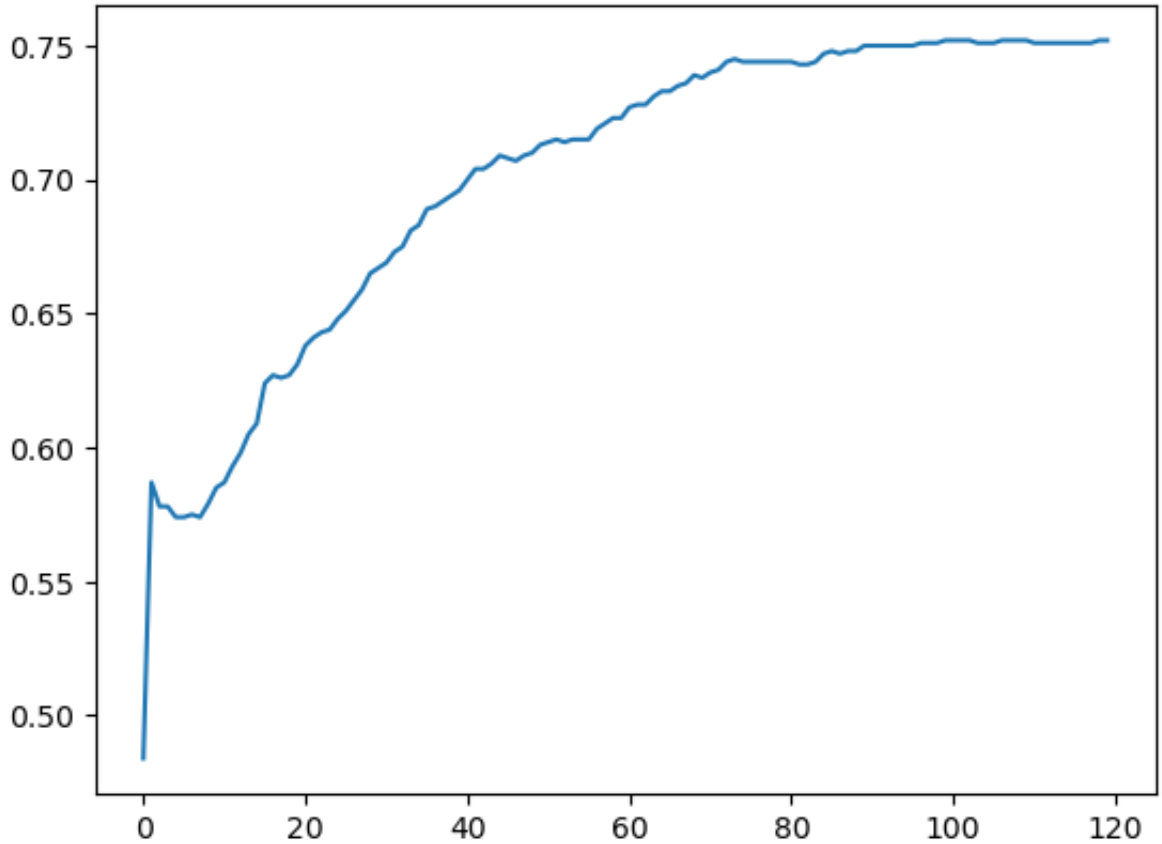




Task: Scaling

In [ ]:
```
def scale(X):
    return X-np.mean(X,axis=0)/np.std(X,axis=0)

run = NumpyLinRegClass()
x_train2 = np.copy(X_train)
x_val = np.copy(X_val)
scalar_data = scale(x_train2)
scalar_val = scale(x_val)

run.fit(scalar_data, t2_train, eta=0.072, epochs=120)
plt.plot(run.lossList)
pred = run.predict(scalar_val)
plt.show()
plt.plot(run.accList)
plt.show()
```





we have seen in the lectures that scaling the data may improve training speed.

- Implement a scaler, either standard scaler (normalizer) or max-min scaler
- Scale the data
- Train the model on the scaled data
- Experiment with hyper-parameter settings and see whether you can speed up the training.
- Report final hyper-meter settings and show how you found them.
- Plot the loss curve and the accuracy curve for the classifier trained on scaled data with the best settings you found.

Logistic regression

a) You should now implement a logistic regression classifier similarly to the classifier based on linear regression. You may use code from the solution to weekly exercise set week07.

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

c) As with the classifier based on linear regression, we want to calculate loss and accuracy after each epoch. The prefered loss for logistic regression is binary cross-entropy. You could have used mean squared error. The most important is that your implementation of the loss corresponds to your implementation of the gradient descent.

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` and `n_epochs_no_update` and stop training when the loss has not improved with more than `tol` after running `n_epochs_no_update` epochs. A possible default value for `n_epochs_no_update` is 5. Also, add an attribute to the classifier which tells us after fitting how many epochs were ran.

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

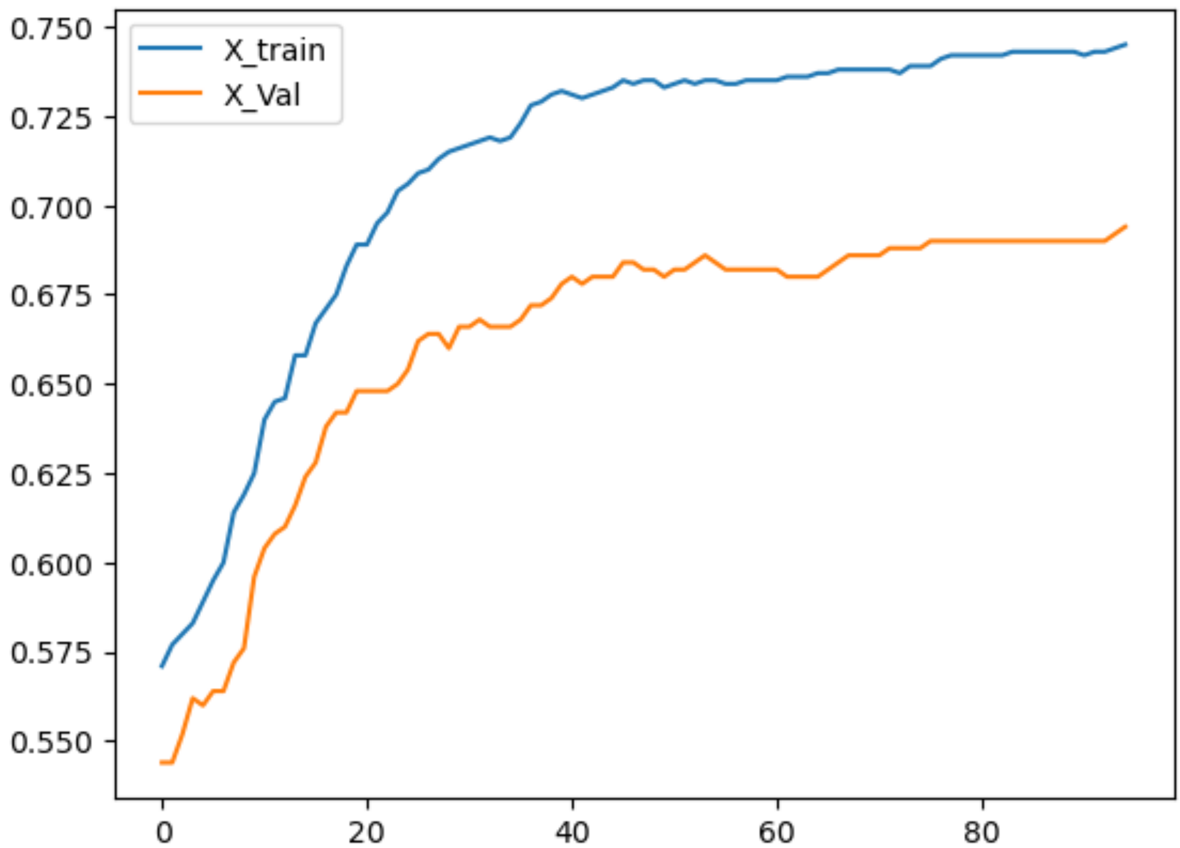
g) After a succesful training, plot both training loss and validation loss as functions of the number of epochs in one figure, and both accuracies as functions of the number of epochs in another figure. Comment on what you see.

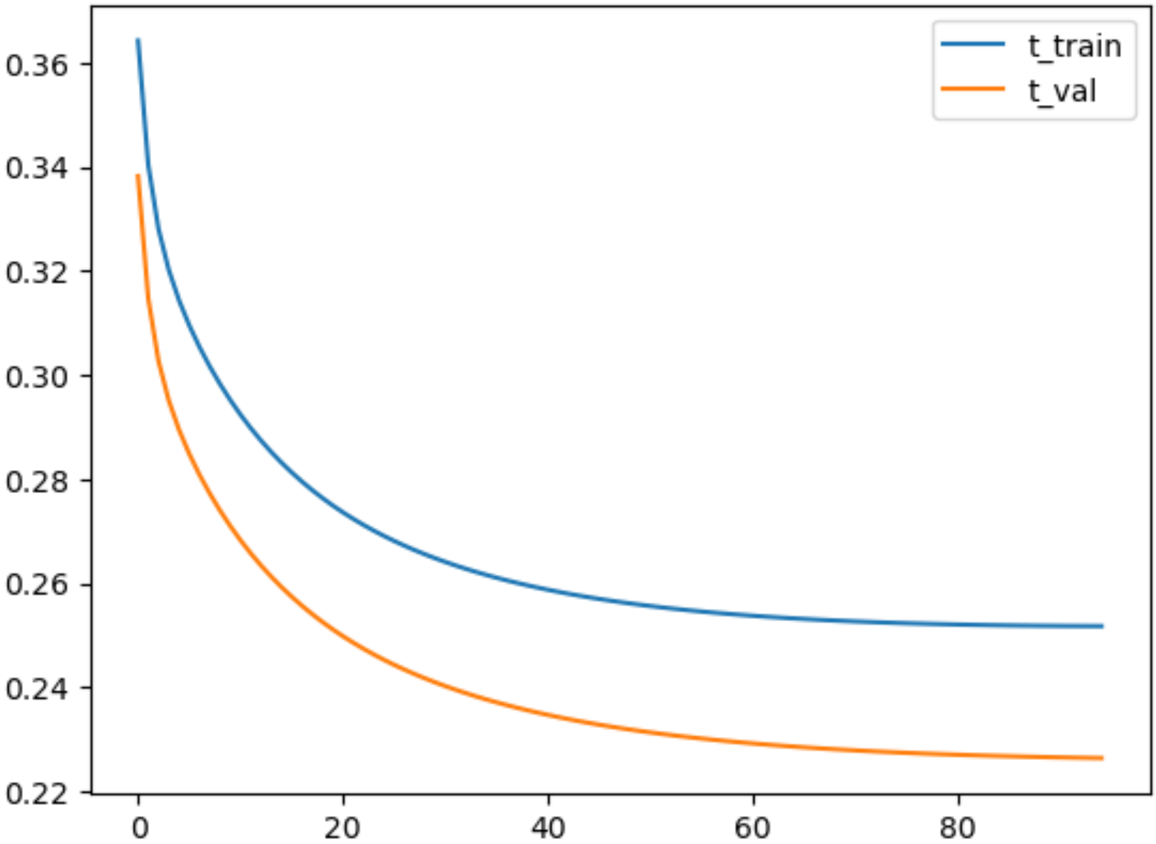
In [ ]:
```
import math
class NumpyLogReg(NumpyClassifier):
    lossList = []
    accList = []
    lossListVal = []
    accListVal = []

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

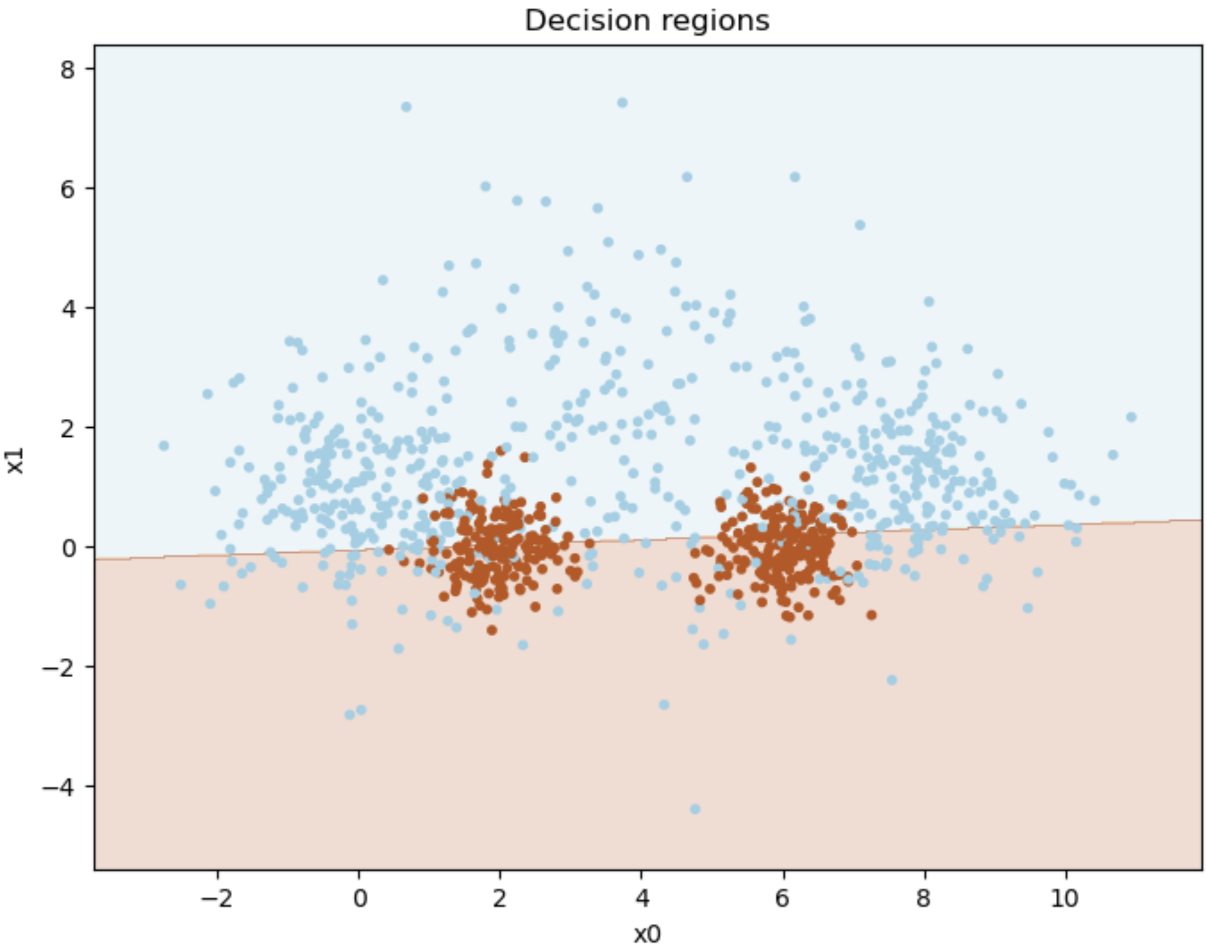
```
    def fit(self, X_train, t_train, x_val, t_val, tol, n_epochs_no_update

        self.lossList = []
        self.accList = []
        self.lossListVal = []
        self.accListVal = []

        (N, m) = X_train.shape
        if self.bias:
            X_train2 = add_bias(X_train,-1)
            x_val1 = add_bias(x_val,-1)

        self.weights = weights = np.zeros(m+1)
        prevLoss = math.inf
        num_epoch = 0
        no_new = 0
        bool = True
        while(bool):
            weights -= eta / N *  X_train2.T @ (self.forward(X_train2) -
            loss = self.crossEntropy(t_train,X_train2)
            self.lossList.append(loss)
            self.accList.append(accuracy(self.predict(X_train), t_train))
            #Here we run for the validation set
            self.lossListVal.append(self.crossEntropy(t_val,x_val1))
            self.accListVal.append(accuracy(self.predict(x_val), t_val))
            num_epoch += 1
            if (prevLoss - loss) <= tol:
                no_new += 1
                if no_new == n_epochs_no_update:
                    bool = False
            else:
                prevLoss = loss
                no_new = 0
        #print("We ran it for" , num_epoch , "epochs")


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

    def predict(self, x, threshold=0.5):
        z = add_bias(x,-1)
        return (self.forward(z) > threshold).astype('int')

    def predict_probability(self,x):
        z = add_bias(x,-1)
        return np.round(self.forward(z)*100)

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

    def crossEntropy(self,t_train,x_train):
        y = self.forward(x_train)
        t = t_train
        p = ((y**t)*(1-y))**(1-t)
        L_temp = -np.log(p)
        L = np.mean(L_temp)
        return L
```

```
run = NumpyLogReg()
x_train3 = np.copy(X_train)
t_train2 = np.copy(t2_train)
x_val2 = np.copy(X_val)
t3_val = np.copy(t2_val)
#x_train3 = scale(x_train3)
#x_val2 = scale(x_val2)
multrun = run.fit(x_train3,t_train2, x_val2, t3_val, 0.0001, 5, 0.1, 200)

#pred = run.predict(x_val1)
#pred(run.forward(x_train3))
#pred_prob = run.predict_probability(x_val1)
#print(pred_prob)
plt.plot(run.accList, label = "X_train")
plt.plot(run.accListVal, label = "X_Val")
plt.legend()
plt.show()
plt.plot(run.lossList, label = "t_train")
plt.plot(run.lossListVal, label = "t_val")
plt.legend()
plt.show()
plot_decision_regions(x_train3, t_train2, run)
```

```
We ran it for 95 epochs
```




t_train
t_val
0.36
0.34
0.32
0.30
0.28
0.26
0.24
0.22
0
20
40
60
80


Decision regions
x1
x0
8
6
4
2
0
−2
−4
−2
0
2
4
6
8
10

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).

"One-vs-rest" with logistic regression

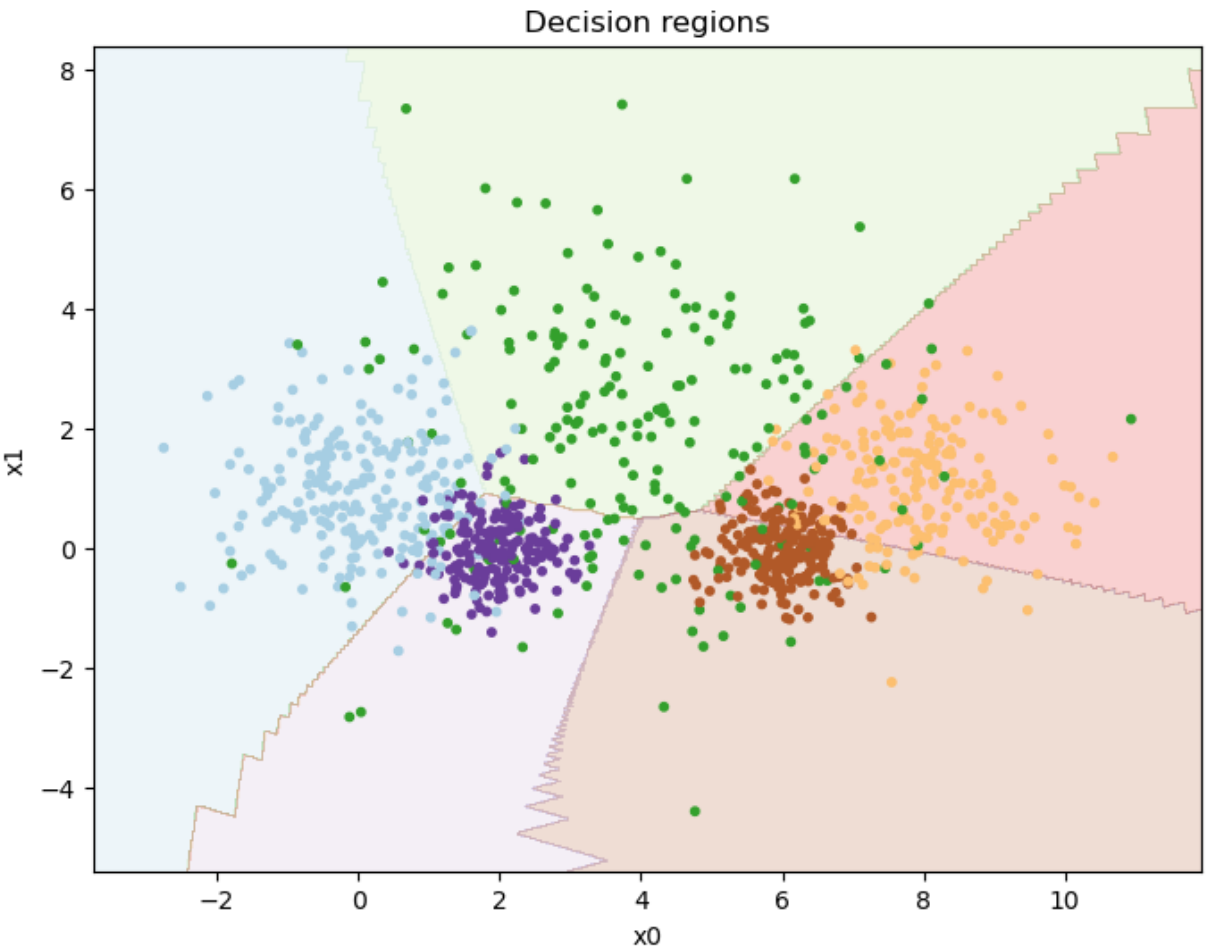
We saw in the lecture how a logistic regression classifier can be turned into a multi-class classifier using the one-vs-rest approach. We train one logistic regression classifier for each class. To predict the class of an item, we run all the binary classifiers and collect the probability score from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

Build such a classifier. Train the resulting classifier on (X_train, t_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.

In [ ]:
```
class MultiClassClassifier(NumpyClassifier):

    lossList = []
    accList = []
    predList = []
    tot_list = []


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

    def fit(self, X_train, t_train, eta = 0.1, epochs=10):

        self.lossList = []
        self.accList = []
        self.tot_list = []

        (N, m) = X_train.shape
        if self.bias:
            X_train2 = add_bias(X_train,-1)

        self.antClass = np.amax(t_train)
        for i in range(self.antClass+1):
            temp = np.copy(t_train)
            for j in range(t_train.size):
                if temp[j] == i:
                    temp[j] = 1
                else:
                    temp[j] = 0
```

```
            self.weights = weights = np.zeros(m+1)
            for e in range(epochs):
                y = self.forward(X_train2,weights)
                weights -= eta / N *  X_train2.T @ (y - temp)
            self.tot_list.append(weights)

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

    def predict(self, x, threshold=0.5):
        weight = self.tot_list
        pred_list = []
        for i in range(self.antClass + 1):
            y = self.predict_probability(x,weight[i])
            pred_list.append(y)
        out = self.decide(pred_list)
        return out

    def predict_probability(self,x,weights):
        z = add_bias(x,-1)
        return np.round(self.forward(z,weights)*100)

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

    def crossEntropy(self,t_train,x_train,weights):
        y = self.forward(x_train,weights)
        t = t_train
        p = ((y**t)*(1-y))**(1-t)
        L_temp = -np.log(p)
        L = np.mean(L_temp)
        return L

    def decide(self,prob_list):
        out_list = []
        for i in range(prob_list[0].size):
            high_prob = 0
            for j in range(5):
                if prob_list[j][i] > high_prob:
                    high_prob = prob_list[j][i]
                    index = j
            out_list.append(index)
        return np.array(out_list)


tmt = np.copy(t_multi_train)
Xt = np.copy(X_train)
x_v = np.copy(x_val)

run = MultiClassClassifier()
lists = run.fit(np.copy(X_train),np.copy(t_multi_train),0.1,1000)
plot_decision_regions(Xt,tmt,run)
```



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.

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.

Part II Multi-layer neural networks

A first non-linear classifier

The following code it a simple implementation of a multi-layer perceptron. It is quite restricted. There is only one hidden layer. 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.

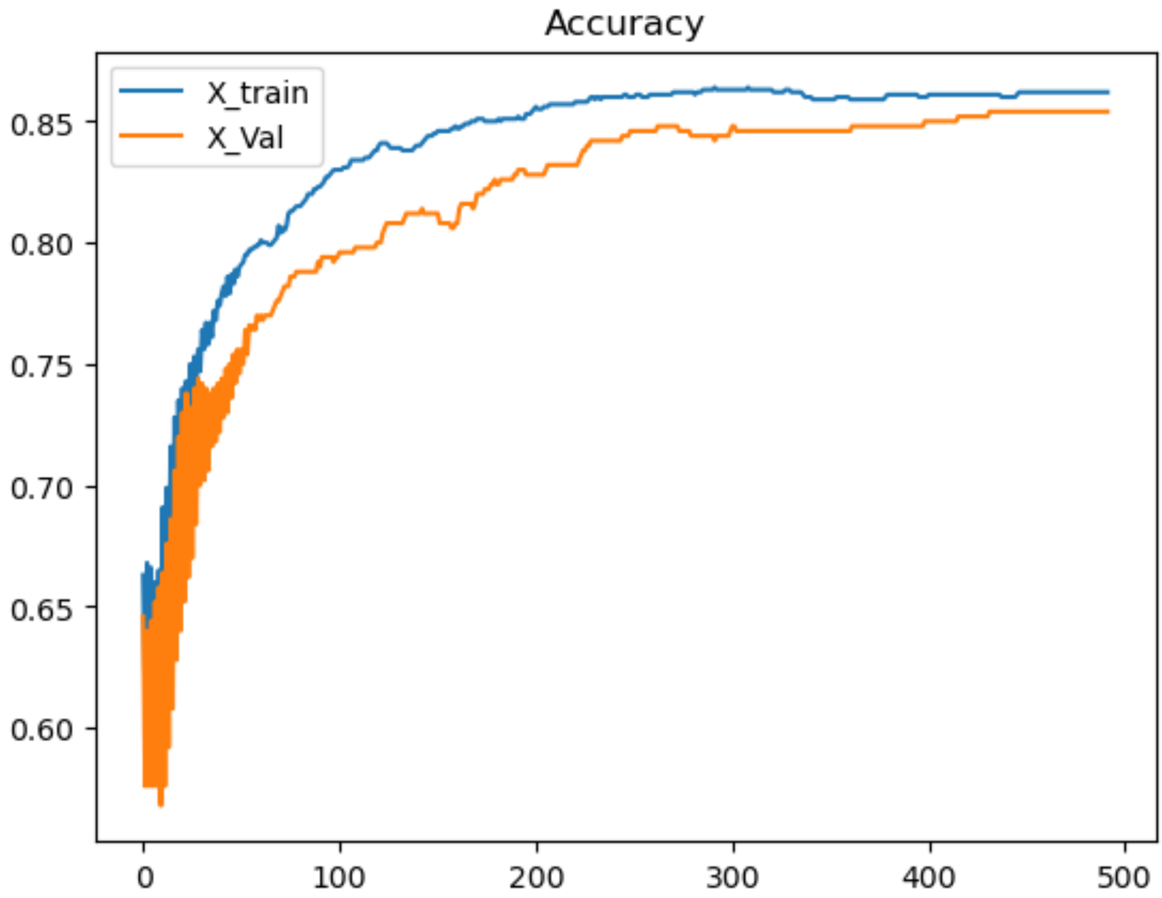
It can be used to make a non-linear classifier for the set (X, t2). Experiment with settings for learning rate and epochs and see how good results you can get. Report results for variouse settings. Be prepared to train for a looooong time. Plot the training set together with the decision regions as in part I.

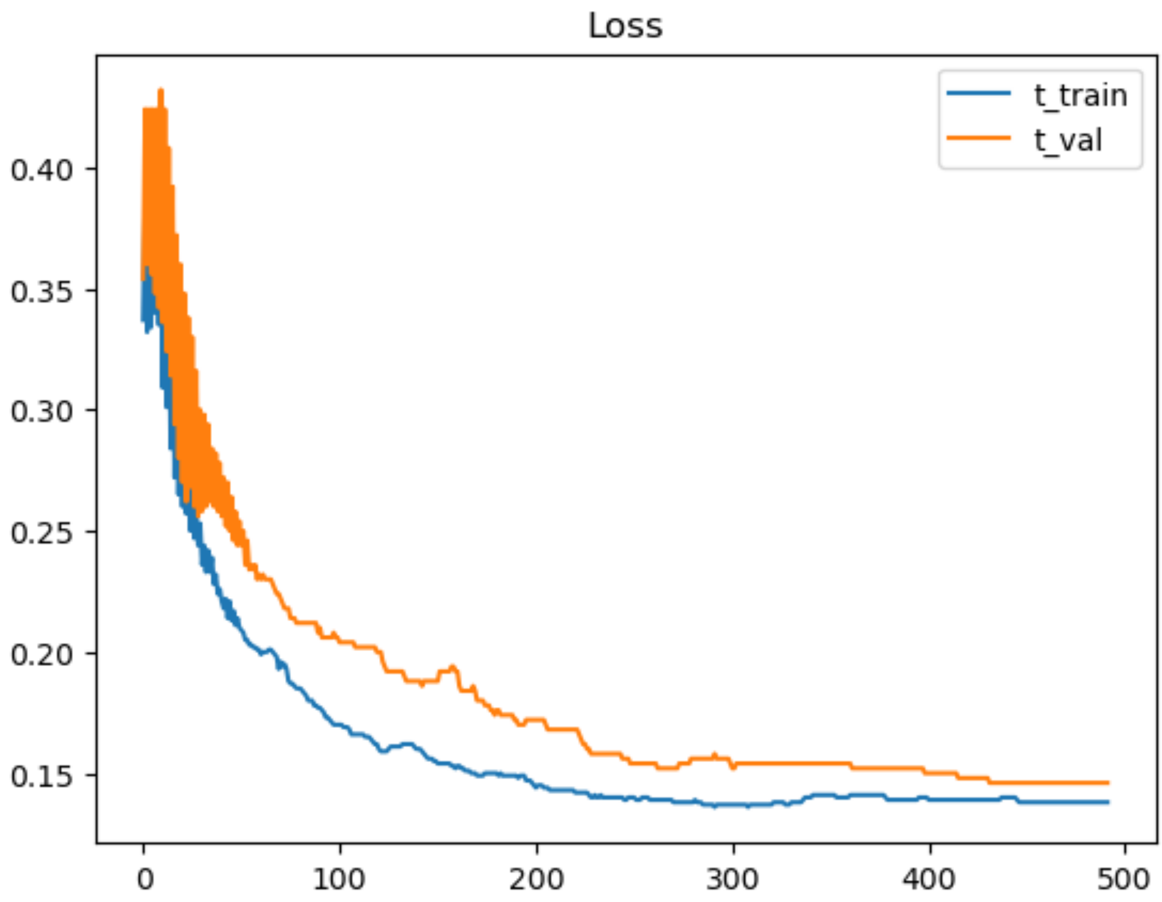
In [ ]:
```
class MLPBinaryLinRegClass(NumpyClassifier):
    """A multi-layer neural network with one hidden layer"""
    lossList = []
    accList = []
    accListVal = []
    lossListVal = []

    def __init__(self, bias=-1, dim_hidden = 56):
        """Intialize the hyperparameters"""
        self.bias = bias
        self.dim_hidden = dim_hidden

        def logistic(x):
            return 1/(1+np.exp(-x))
        self.activ = logistic

        def logistic_diff(y):
            return y * (1 - y)
        self.activ_diff = logistic_diff

    def fit(self, X_train, t_train, x_val, t_val, tol, n_epochs_no_update
        """Intialize 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.
        """
        self.lossList = []
        self.accList = []
        self.accListVal = []
        self.lossListVal = []

        self.eta = eta

        if self.bias:
            X_train2 = add_bias(X_train,-1)
            x_val2 = add_bias(x_val,-1)
```

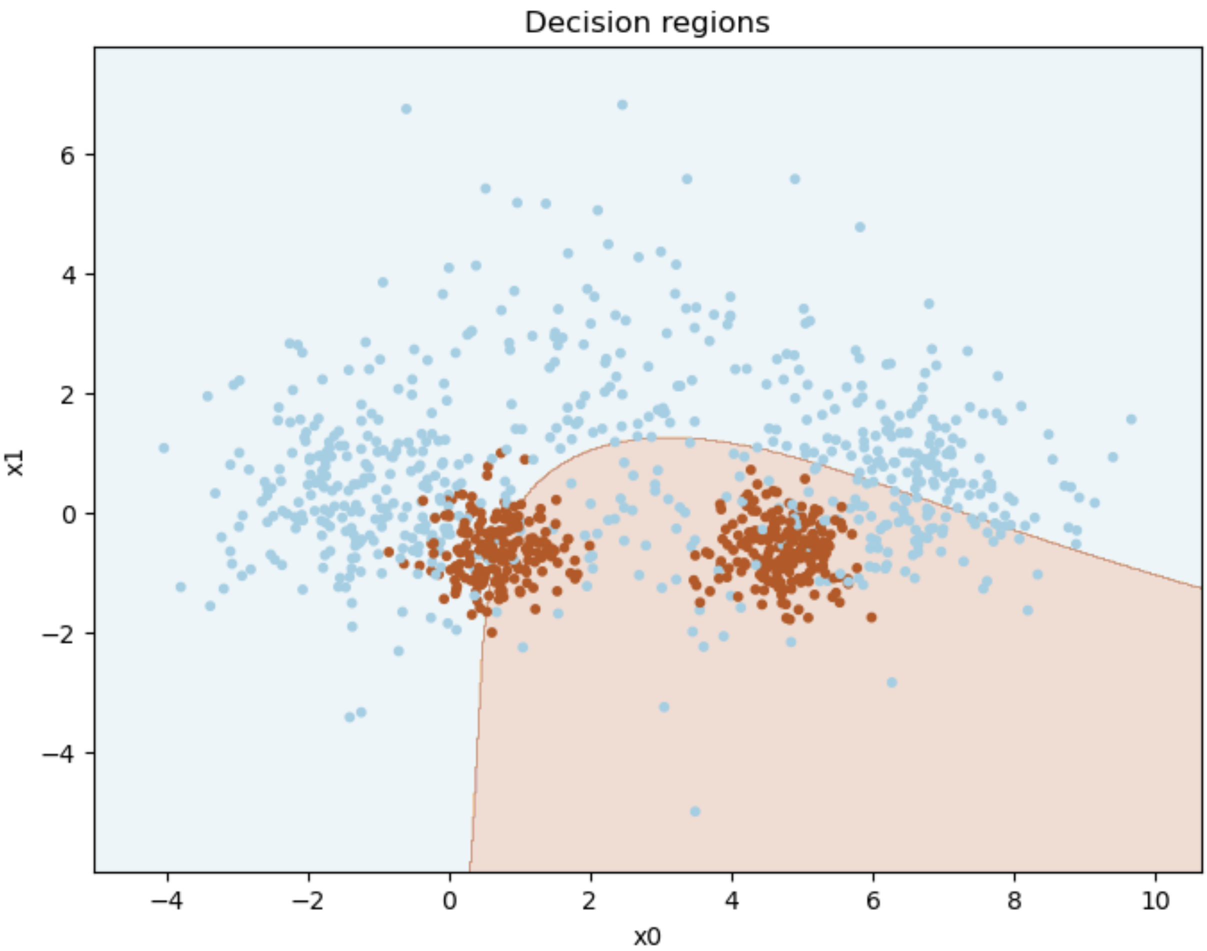
```
        T_train = t_train.reshape(-1,1)

        dim_in = X_train.shape[1]
        dim_out = T_train.shape[1]

        # Itilaize 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)

        prevLoss = math.inf
        num_epoch = 0
        no_new = 0
        bool = True
        while(bool):
        #for i in range(epochs):
            # One epoch
            hidden_outs, outputs = self.forward(X_train_bias)
            # The forward step
            out_deltas = (outputs - T_train)
            # The delta term on the output node
            hiddenout_diffs = out_deltas @ self.weights2.T
            # The delta terms at the output of the jidden layer
            hiddenact_deltas = (hiddenout_diffs[:, 1:] *
                                self.activ_diff(hidden_outs[:, 1:]))
            # The deltas at the input to the hidden layer
            self.weights2 -= self.eta * hidden_outs.T @ out_deltas
            self.weights1 -= self.eta * X_train_bias.T @ hiddenact_deltas
            # Update the weights

            loss = self.mse(t_train,self.predict(X_train))
            self.lossList.append(loss)
            self.accList.append(accuracy(self.predict(X_train), t_train))
            loss2 = self.mse(t_val,self.predict(x_val))
            self.lossListVal.append(loss2)
            self.accListVal.append(accuracy(self.predict(x_val), t_val))
            num_epoch += 1
            if (prevLoss - loss) <= tol:
                no_new += 1
                if no_new == n_epochs_no_update:
                    bool = False
            else:
                prevLoss = loss
                no_new = 0
        #print("We ran it for" , num_epoch , "epochs")
        return self.lossList, self.accList, self.lossListVal, self.accLis

    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"""
        hidden_activations = self.activ(X @ self.weights1)
        hidden_outs = add_bias(hidden_activations, self.bias)
```

```
        outputs = hidden_outs @ self.weights2
        return hidden_outs, outputs

    def crossEntropy(self,t_train,x_train):
        x,y = self.forward(x_train)
        t = t_train
        p = ((y**t)*(1-y))**(1-t)
        L_temp = -np.log(p)
        L = np.mean(L_temp)
        return L

    def mse(self, y, y_pred):
        return np.mean((y - y_pred)**2)

    def predict_probability(self,x):
        """Predict the class for the mebers of X"""
        Z = add_bias(X, self.bias)
        forw = self.forward(Z)[1]
        score= forw[:, 0]
        return score

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

In [ ]:
```
bestAccuracyList = 0
bestRun = 0
bestAccuracy = math.inf
for i in range(11):
    run = MLPBinaryLinRegClass()
    x_train4 = np.copy(X_train)
    t_train3 = np.copy(t2_train)
    x_val3 = np.copy(X_val)
    t3_val1 = np.copy(t2_val)
    lossList, accList, lossListVal, accListVal = run.fit(x_train4,t_train
    if bestAccuracy > accList[-1]:
        bestRun = run

pred = run.predict(x_val3)
pred_prob = run.predict_probability(x_val3)

plt.plot(bestRun.accList, label = "X_train")
plt.plot(bestRun.accListVal, label = "X_Val")
plt.title("Accuracy")
plt.legend()
plt.show()
plt.plot(bestRun.lossList, label = "t_train")
plt.plot(bestRun.lossListVal, label = "t_val")
plt.title("Loss")
plt.legend()
plt.show()
plot_decision_regions(x_train3, t_train2, bestRun)
```


Accuracy
X_train
X_Val
0.85
0.80
0.75
0.70
0.65
0.60
0
100
200
300
400
500


Loss
t_train
t_val
0.40
0.35
0.30
0.25
0.20
0.15
0
100
200
300
400
500


Decision regions
x1
x0
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10

Improving the 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 method predict, which predicts a class for the data, include a method predict_probability which predict the probability of the data belonging to the positive class. The training should be based on this value as with logistic regression.

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

c) 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.

d) 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` and `n_epochs_no_update` and stop training when the loss has not improved with more than tol after `n_epochs_no_update`. A possible default value for `n_epochs_no_update` is 5. Also, add an attribute to the classifier which tells us after fitting how many epochs were ran.

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

f) After a succesful training with a 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 accuracies as functions of the number of epochs in another figure. Comment on what you see.

g) The algorithm contains an element of non-determinism. Hence, train the classifier 10 times with the optimal hyper-parameters and report the mean and standard deviation of the accuracies over the 10 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 network for non-linear multi-class classfication and apply it to set (X, t_multi).

Modify the netork to become a multi-class classifier. As a 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..

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 calculate accuracy 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 results 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 that the case here?

Also report precision and recall for class 1.

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

The following part is only mandatory for in4050-students. In3050-students are also welcome to do it. It requires the earlier IN4050-only tasks.

Compare the three multi-class classfiers, the one-vs-rest 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 what you see

In [ ]:
```
run1 = NumpyLinRegClass()
run2 = NumpyLogReg()
run3 = MLPBinaryLinRegClass()

run1.fit(np.copy(X_train),np.copy(t_train2), eta = 0.0704, epochs=200)
run2.fit(np.copy(X_train),np.copy(t_train2), np.copy(X_val), np.copy(t2_v
run3.fit(np.copy(X_train),np.copy(t_train2), np.copy(X_val), np.copy(t2_v

r1Train = accuracy(run1.predict(np.copy(X_train)),np.copy(t2_train))
r1Val = accuracy(run1.predict(np.copy(X_val)),np.copy(t2_val))
r1Test = accuracy(run1.predict(np.copy(X_test)),np.copy(t2_test))

r2Train = accuracy(run2.predict(np.copy(X_train)),np.copy(t2_train))
r2Val = accuracy(run2.predict(np.copy(X_val)),np.copy(t2_val))
r2Test = accuracy(run2.predict(np.copy(X_test)),np.copy(t2_test))

r3Train = accuracy(run3.predict(np.copy(X_train)),np.copy(t2_train))
r3Val = accuracy(run3.predict(np.copy(X_val)),np.copy(t2_val))
r3Test = accuracy(run3.predict(np.copy(X_test)),np.copy(t2_test))

print("    | train  | val   | test  |")
print(f"mlp | {r1Train}  | {r1Val}| {r1Test} |")
print(f"lin | {r2Train}  | {r2Val} | {r2Test} |")
print(f"lin | {r3Train}  | {r3Val}| {r3Test} |")
```

```
We ran it for 103 epochs
    | train  | val  | test  |
mlp | 0.751  | 0.704| 0.724 |
lin | 0.747  | 0.69 | 0.718 |
lin | 0.863  | 0.852| 0.862 |
```