IN4050 Mandatory Assignment 2, 2024: Supervised Learning

Name: Hanne Rønning Berg

Username: hannerbe@uio.no

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: Tuesday, October 29, 2024, 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 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 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. A significant part is 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 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 and Pandas, 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.

Beware

This is a revised assignment compared to earlier years. 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 [161…

```
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 2000 instances and five classes, with 400 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 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(2024)`.

In [162…

```
# Generating the dataset
from sklearn.datasets import make_blobs
X, t_multi = make_blobs(n_samples=[400, 400, 400, 400, 400], centers=[[0,1],[4,2
                        n_features=2, random_state=2024, cluster_std=[1.0, 2.0, 1.0, 0
```

In [163…

```
# Shuffling the dataset
indices = np.arange(X.shape[0])
rng = np.random.RandomState(2024)
rng.shuffle(indices)
indices[:10]
```

Out[163…
```
array([ 937, 1776,  868, 1282, 1396,  147,  601, 1193, 1789,  547])
```

In [164…
```
# Splitting into train, dev and test
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 with only two classes by merging the existing labels in (X,t), so that `0`, `1` and `2` become the new `0` and `3` and `4` become the new `1`. Let's call the new set (X, t2). This will be a binary set. We now have two datasets:

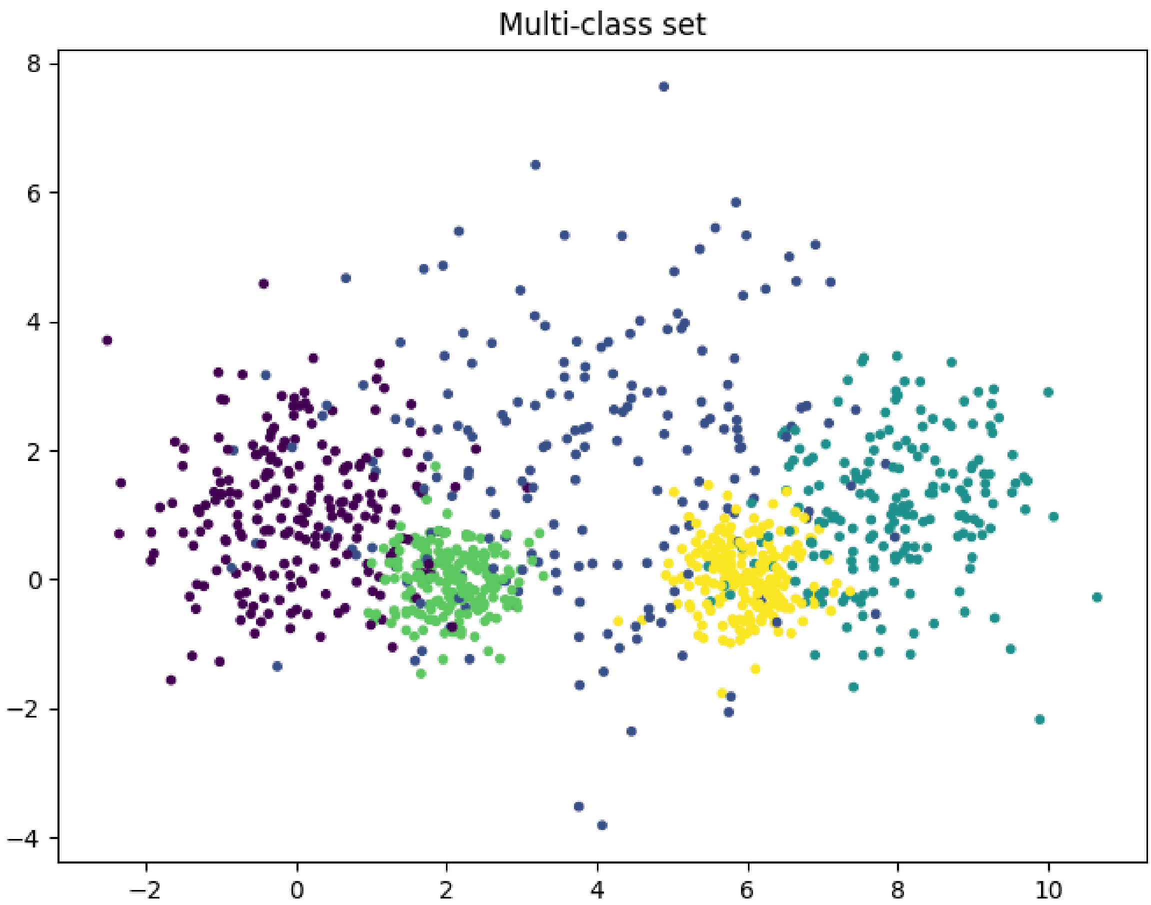
- Binary set: `(X, t2)`
- Multi-class set: `(X, t_multi)`

In [165…
```
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 [166…
```
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[166…
```
Text(0.5, 1.0, 'Multi-class set')
```

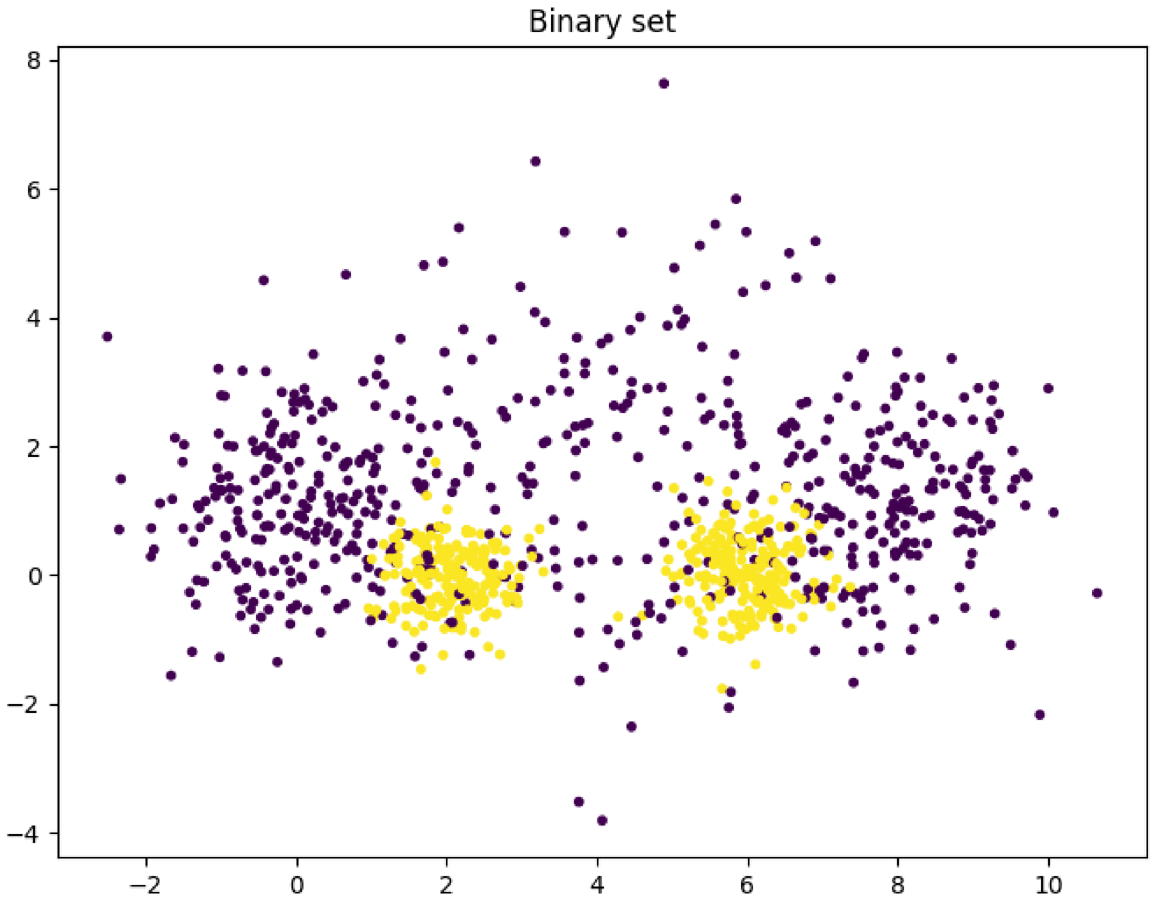


In [167…

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

Out[167…

```
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 7. We include it here with a little added flexibility.

In [168…

```
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 [169…

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

In [170…

```
class NumpyLinRegClass(NumpyClassifier):

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

    def fit(self, X_train, t_train, lr = 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 -= lr / N *  X_train.T @ (X_train @ weights - 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
```

We can train and test a first classifier.

In [171…

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

In [172…

```
cl = NumpyLinRegClass()
cl.fit(X_train, t2_train, epochs=3)
print("Accuracy on the validation set:", accuracy(cl.predict(X_val), t2_val))
```

```
Accuracy on the validation set: 0.58
```

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

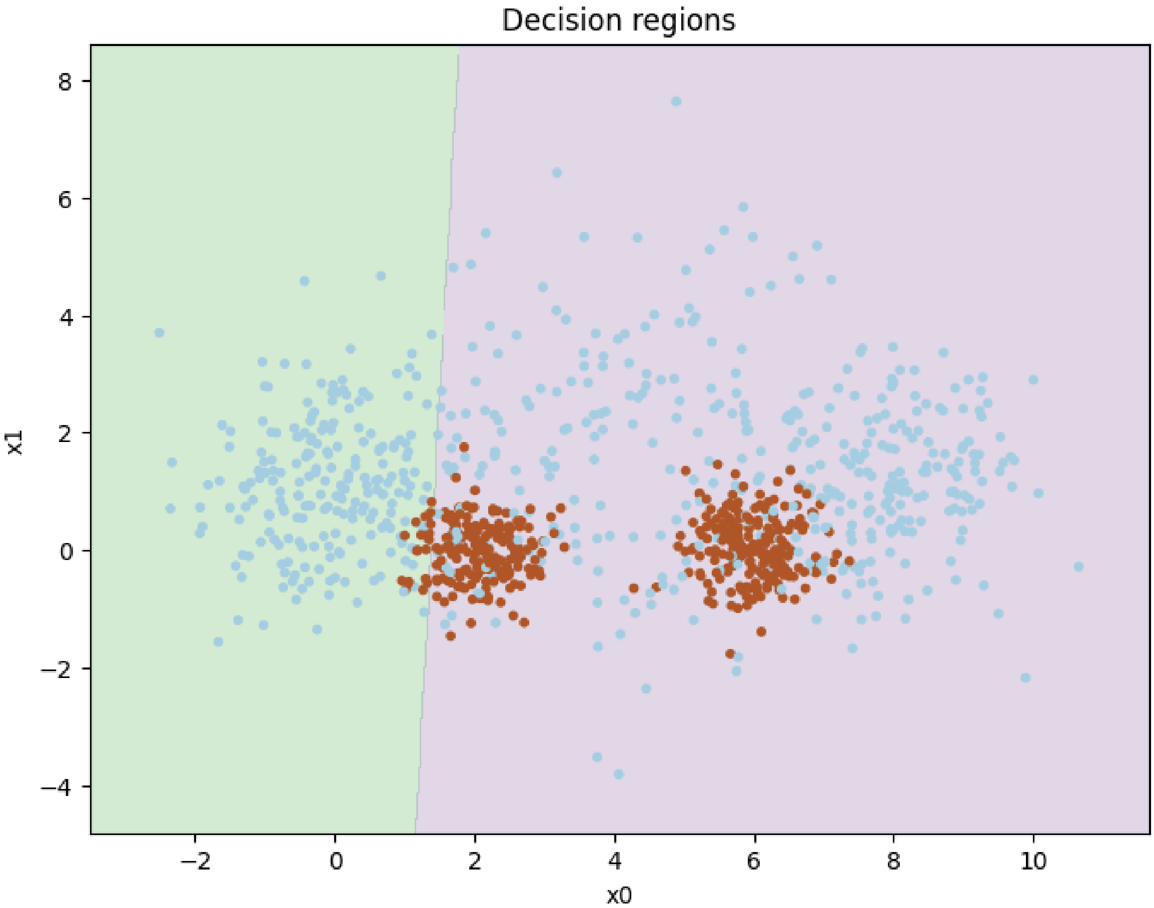
In [173…

```
def plot_decision_regions(X, t, clf=[], size=(8,6)):
    """Plot the data set (X,t) together with the decision boundary of the classi
    # 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_max, h))
```

```
    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")
```

In [174…

```
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.

In [175…

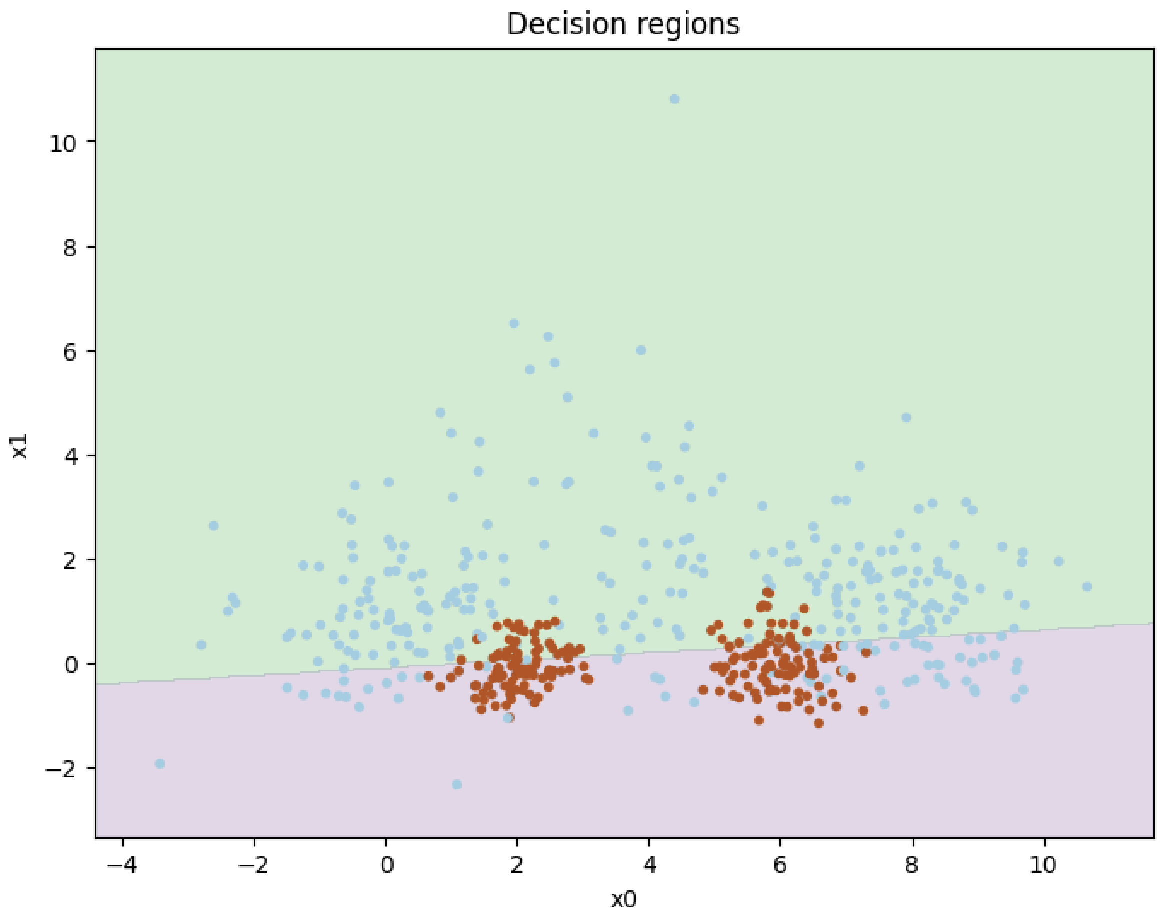
```
lrs = [10**-3, 10**-2, 10**-1, 1, 2]
n_epochs = [1000, 100, 20, 10, 5, 3]

best = 0
hyperparams = ()
for lr in lrs:
    for epochs in n_epochs:
        cl= NumpyLinRegClass()
        cl.fit(X_train, t2_train, lr = lr, epochs=epochs)
        accur = accuracy(cl.predict(X_val), t2_val)
        print(f"Accuracy: {accur}, for the validation set with learning rate {lr
        if accur > best:
            best = accur
            hyperparams = (lr, epochs)

print()
print("The best hyperparameters was:")
print(f"Learning rate: {hyperparams[0]} and {hyperparams[1]} epochs")
print(f"Which gave an accuracy of {best}")
```

```
Accuracy: 0.566, for the validation set with learning rate 0.001 and 1000 epochs.
Accuracy: 0.488, for the validation set with learning rate 0.001 and 100 epochs.
Accuracy: 0.604, for the validation set with learning rate 0.001 and 20 epochs.
Accuracy: 0.604, for the validation set with learning rate 0.001 and 10 epochs.
Accuracy: 0.604, for the validation set with learning rate 0.001 and 5 epochs.
Accuracy: 0.604, for the validation set with learning rate 0.001 and 3 epochs.
Accuracy: 0.75, for the validation set with learning rate 0.01 and 1000 epochs.
Accuracy: 0.566, for the validation set with learning rate 0.01 and 100 epochs.
Accuracy: 0.464, for the validation set with learning rate 0.01 and 20 epochs.
Accuracy: 0.47, for the validation set with learning rate 0.01 and 10 epochs.
Accuracy: 0.598, for the validation set with learning rate 0.01 and 5 epochs.
Accuracy: 0.604, for the validation set with learning rate 0.01 and 3 epochs.
Accuracy: 0.534, for the validation set with learning rate 0.1 and 1000 epochs.
Accuracy: 0.534, for the validation set with learning rate 0.1 and 100 epochs.
Accuracy: 0.534, for the validation set with learning rate 0.1 and 20 epochs.
Accuracy: 0.542, for the validation set with learning rate 0.1 and 10 epochs.
Accuracy: 0.53, for the validation set with learning rate 0.1 and 5 epochs.
Accuracy: 0.58, for the validation set with learning rate 0.1 and 3 epochs.
Accuracy: 0.604, for the validation set with learning rate 1 and 1000 epochs.
Accuracy: 0.534, for the validation set with learning rate 1 and 100 epochs.
Accuracy: 0.534, for the validation set with learning rate 1 and 20 epochs.
Accuracy: 0.534, for the validation set with learning rate 1 and 10 epochs.
Accuracy: 0.466, for the validation set with learning rate 1 and 5 epochs.
Accuracy: 0.466, for the validation set with learning rate 1 and 3 epochs.
Accuracy: 0.604, for the validation set with learning rate 2 and 1000 epochs.
Accuracy: 0.534, for the validation set with learning rate 2 and 100 epochs.
Accuracy: 0.534, for the validation set with learning rate 2 and 20 epochs.
Accuracy: 0.534, for the validation set with learning rate 2 and 10 epochs.
Accuracy: 0.466, for the validation set with learning rate 2 and 5 epochs.
Accuracy: 0.466, for the validation set with learning rate 2 and 3 epochs.

The best hyperparameters was:
Learning rate: 0.01 and 1000 epochs
Which gave an accuracy of 0.75
```

```
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\1812362137.py:22: RuntimeWarnin
g: overflow encountered in matmul
  weights -= lr / N *  X_train.T @ (X_train @ weights - t_train)
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\1812362137.py:22: RuntimeWarnin
g: invalid value encountered in matmul
  weights -= lr / N *  X_train.T @ (X_train @ weights - t_train)
```

In [176…

```
cl = NumpyLinRegClass()
cl.fit(X_train, t2_train, lr = 0.01, epochs=1000 )
plot_decision_regions(X_val, t2_val, cl)
```



Task: Scaling

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

- Implement a scaler, at least the standard scaler (normalizer), but you can also try other techniques
- 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-parameter settings and show how you found them.

In [177…

```
def normalize(X):
    X_mean = X.mean(axis=0)
    X_std = X.std(axis=0)
    return (X - X_mean) / X_std
```

```
X_train_norm = normalize(X_train)
X_val_norm = normalize(X_val)


def test_hyperparams(X, t, X_val, t_val):
    lrs = [10**-4, 0.0005, 10**-3, 0.005, 10**-2, 10**-1, 1, 2]
    n_epochs = [4000, 1000,500, 100, 20, 10, 5, 3]

    best = 0
    hyperparams = ()
    for lr in lrs:
        for epochs in n_epochs:
            cl= NumpyLinRegClass()
            cl.fit(X, t, lr = lr, epochs=epochs)
            accur = accuracy(cl.predict(X_val), t_val)
            #print(f"Accuracy: {accur}, for the validation set with learning rat
            if accur > best:
                best = accur
                hyperparams = (lr, epochs)

    return hyperparams, best

test_hyperparams(X_train_norm, t2_train, X_val_norm, t2_val)
```

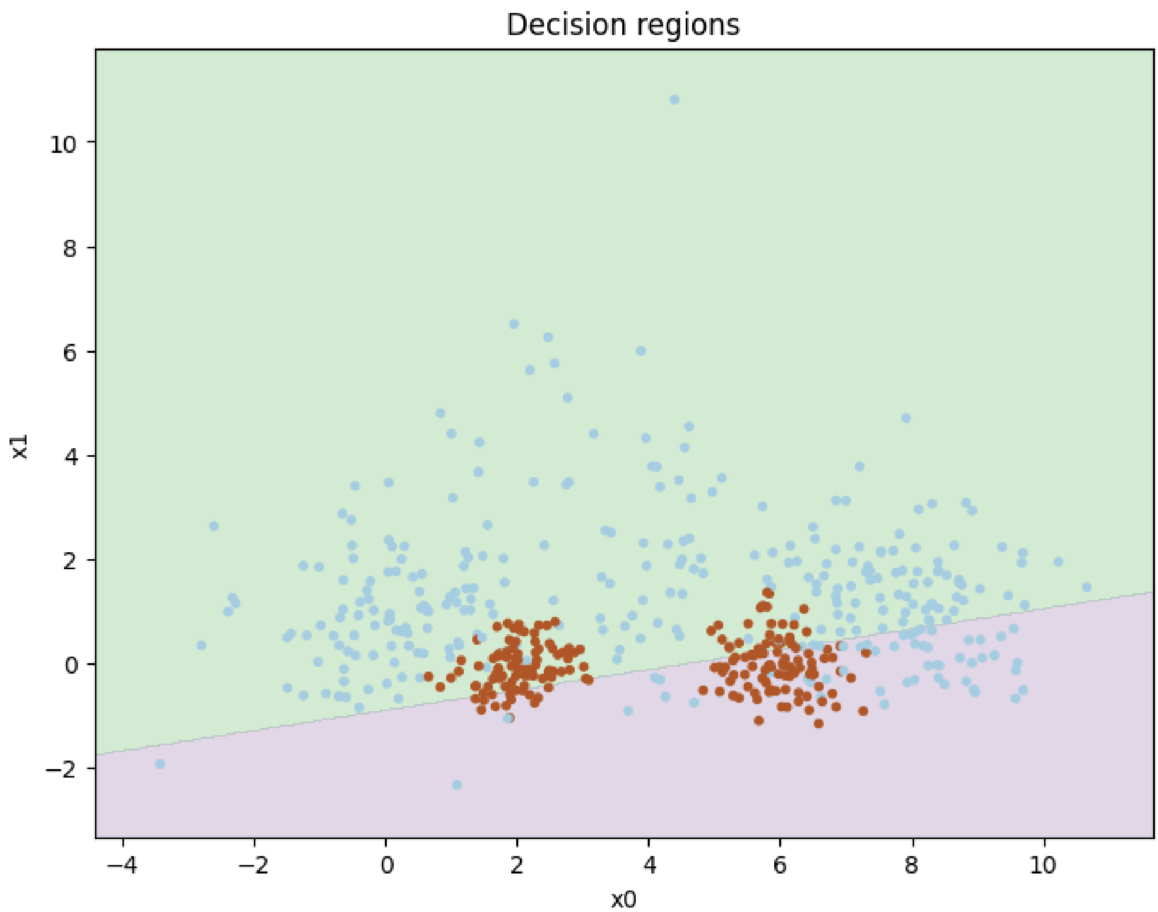
Out[177…

```
((0.001, 4000), 0.754)
```

From 0.75 to 0.754 in accuracy.

In [178…

```
cl = NumpyLinRegClass()
cl.fit(X_train, t2_train, lr = 0.001, epochs=4000 )
plot_decision_regions(X_val, t2_val, cl)
```



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 week07.

b) 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.

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`. 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 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 [179…

```python
def sigmoid(X):
    return 1 / (1 + np.exp(-X))

class NumpyLogRegClass(NumpyClassifier):

    def __init__(self, bias=-1):
        self.bias=bias
        self.losses = []
        self.accuracies = []
        self.val_losses = []
        self.val_accuracies = []
        self.epoch_trained = 0

    def fit(self, X_train, t_train, eta = 0.1, epochs=10, X_val = None, t_val =
        """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)

        has_valset = X_val is not None

        if has_valset:
            X_val = add_bias(X_val, self.bias)

        N, m = np.shape(X_train)

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

        for e in range(epochs):

            loss = -np.mean(t_train * np.log(self.forward(X_train)) + (1 - t_tra
            self.losses.append(loss)

            if has_valset:
                loss = -np.mean(t_val * np.log(self.forward(X_val)) + (1 - t_val
                self.val_losses.append(loss)

            weights -= eta/N * X_train.T @ (self.forward(X_train) - t_train) #up

            #accur = accuracy(self.predict(X_train), t_train)
            accur = accuracy(self.forward(X_train), t_train)
            self.accuracies.append(accur)
```

```
            if has_valset:
                self.val_accuracies.append(accuracy(self.forward(X_val), t_val))

            if len(self.losses) > n_epochs_no_update and self.losses[-n_epochs_n
                break

        self.epoch_trained = len(self.losses) #the number of losses stored is al

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

    def predict(self, X, threshold = .5):
        """X is a Kxm matrix for some K>=1
        predict the value for each point in X"""

        #this function is also used in fit(). Can't add bias again if there alre
        if self.bias and X.shape[1] != self.weights.shape[0]:
            X = add_bias(X, self.bias)
        return (self.forward(X) > threshold).astype('int')



    def predict_probability(self, X):
        z = add_bias(X, self.bias)
        return self.forward(z)

    def loss(self, X, t):
        return -np.mean(t * np.log(self.forward(X)) + (1 - t) * np.log(1 - self.
```

In [180…

```
def test_hyperparams(X,y, X_val, y_val):
    tols = [10**-3, 10**-2, 10**-1, 1]
    lrs = [10**-4, 0.0005, 10**-3, 0.005, 10**-2, 10**-1, 1, 2]

    best = 0
    hyperparams = ()

    for lr in lrs:
        for tol in tols:
            cl= NumpyLogRegClass()
            cl.fit(X, y, eta = lr, tol=tol, epochs=10_000) #epochs can be super
            accur = accuracy(cl.predict(X_val), y_val)
            print(f"Accuracy: {accur}, for the validation set with learning rate
            if accur > best:
                best = accur
                hyperparams = (lr, tol)

    print()
    print("The best hyperparameters was:")
    print(f"Learning rate: {hyperparams[0]} and {hyperparams[1]} tolerance")
    print(f"Which gave an accuracy of {best}")
```

In [181…

```
test_hyperparams(X_train, t2_train, X_val, t2_val)
```

```
Accuracy: 0.562, for the validation set with learning rate 0.0001 and tolerance:
0.001.
Accuracy: 0.562, for the validation set with learning rate 0.0001 and tolerance:
0.01.
Accuracy: 0.562, for the validation set with learning rate 0.0001 and tolerance:
0.1.
Accuracy: 0.562, for the validation set with learning rate 0.0001 and tolerance:
1.
Accuracy: 0.562, for the validation set with learning rate 0.0005 and tolerance:
0.001.
Accuracy: 0.562, for the validation set with learning rate 0.0005 and tolerance:
0.01.
Accuracy: 0.562, for the validation set with learning rate 0.0005 and tolerance:
0.1.
Accuracy: 0.562, for the validation set with learning rate 0.0005 and tolerance:
1.
Accuracy: 0.562, for the validation set with learning rate 0.001 and tolerance:
0.001.
Accuracy: 0.562, for the validation set with learning rate 0.001 and tolerance:
0.01.
Accuracy: 0.562, for the validation set with learning rate 0.001 and tolerance:
0.1.
Accuracy: 0.562, for the validation set with learning rate 0.001 and tolerance:
1.
Accuracy: 0.678, for the validation set with learning rate 0.005 and tolerance:
0.001.
Accuracy: 0.562, for the validation set with learning rate 0.005 and tolerance:
0.01.
Accuracy: 0.562, for the validation set with learning rate 0.005 and tolerance:
0.1.
Accuracy: 0.562, for the validation set with learning rate 0.005 and tolerance:
1.
Accuracy: 0.734, for the validation set with learning rate 0.01 and tolerance: 0.
001.
Accuracy: 0.562, for the validation set with learning rate 0.01 and tolerance: 0.
01.
Accuracy: 0.562, for the validation set with learning rate 0.01 and tolerance: 0.
1.
Accuracy: 0.562, for the validation set with learning rate 0.01 and tolerance: 1.
Accuracy: 0.752, for the validation set with learning rate 0.1 and tolerance: 0.0
01.
Accuracy: 0.74, for the validation set with learning rate 0.1 and tolerance: 0.0
1.
Accuracy: 0.614, for the validation set with learning rate 0.1 and tolerance: 0.
1.
Accuracy: 0.614, for the validation set with learning rate 0.1 and tolerance: 1.
Accuracy: 0.612, for the validation set with learning rate 1 and tolerance: 0.00
1.
Accuracy: 0.612, for the validation set with learning rate 1 and tolerance: 0.01.
Accuracy: 0.612, for the validation set with learning rate 1 and tolerance: 0.1.
Accuracy: 0.612, for the validation set with learning rate 1 and tolerance: 1.
Accuracy: 0.692, for the validation set with learning rate 2 and tolerance: 0.00
1.
Accuracy: 0.692, for the validation set with learning rate 2 and tolerance: 0.01.
Accuracy: 0.692, for the validation set with learning rate 2 and tolerance: 0.1.
Accuracy: 0.692, for the validation set with learning rate 2 and tolerance: 1.

The best hyperparameters was:
Learning rate: 0.1 and 0.001 tolerance
Which gave an accuracy of 0.752
```

In [182…
```
test_hyperparams(X_train_norm, t2_train, X_val_norm, t2_val)
```

```
Accuracy: 0.762, for the validation set with learning rate 0.0001 and tolerance:
0.001.
Accuracy: 0.762, for the validation set with learning rate 0.0001 and tolerance:
0.01.
Accuracy: 0.762, for the validation set with learning rate 0.0001 and tolerance:
0.1.
Accuracy: 0.762, for the validation set with learning rate 0.0001 and tolerance:
1.
Accuracy: 0.762, for the validation set with learning rate 0.0005 and tolerance:
0.001.
Accuracy: 0.762, for the validation set with learning rate 0.0005 and tolerance:
0.01.
Accuracy: 0.762, for the validation set with learning rate 0.0005 and tolerance:
0.1.
Accuracy: 0.762, for the validation set with learning rate 0.0005 and tolerance:
1.
Accuracy: 0.762, for the validation set with learning rate 0.001 and tolerance:
0.001.
Accuracy: 0.762, for the validation set with learning rate 0.001 and tolerance:
0.01.
Accuracy: 0.762, for the validation set with learning rate 0.001 and tolerance:
0.1.
Accuracy: 0.762, for the validation set with learning rate 0.001 and tolerance:
1.
Accuracy: 0.758, for the validation set with learning rate 0.005 and tolerance:
0.001.
Accuracy: 0.762, for the validation set with learning rate 0.005 and tolerance:
0.01.
Accuracy: 0.762, for the validation set with learning rate 0.005 and tolerance:
0.1.
Accuracy: 0.762, for the validation set with learning rate 0.005 and tolerance:
1.
Accuracy: 0.758, for the validation set with learning rate 0.01 and tolerance: 0.
001.
Accuracy: 0.762, for the validation set with learning rate 0.01 and tolerance: 0.
01.
Accuracy: 0.762, for the validation set with learning rate 0.01 and tolerance: 0.
1.
Accuracy: 0.762, for the validation set with learning rate 0.01 and tolerance: 1.
Accuracy: 0.76, for the validation set with learning rate 0.1 and tolerance: 0.00
1.
Accuracy: 0.758, for the validation set with learning rate 0.1 and tolerance: 0.0
1.
Accuracy: 0.758, for the validation set with learning rate 0.1 and tolerance: 0.
1.
Accuracy: 0.758, for the validation set with learning rate 0.1 and tolerance: 1.
Accuracy: 0.764, for the validation set with learning rate 1 and tolerance: 0.00
1.
Accuracy: 0.76, for the validation set with learning rate 1 and tolerance: 0.01.
Accuracy: 0.758, for the validation set with learning rate 1 and tolerance: 0.1.
Accuracy: 0.758, for the validation set with learning rate 1 and tolerance: 1.
Accuracy: 0.764, for the validation set with learning rate 2 and tolerance: 0.00
1.
Accuracy: 0.76, for the validation set with learning rate 2 and tolerance: 0.01.
Accuracy: 0.76, for the validation set with learning rate 2 and tolerance: 0.1.
Accuracy: 0.76, for the validation set with learning rate 2 and tolerance: 1.

The best hyperparameters was:
Learning rate: 1 and 0.001 tolerance
Which gave an accuracy of 0.764
```

With scaled data the best model was marginally better than the model with non-scaled data. However, the results from the scaled data are overall a lot more even and better.

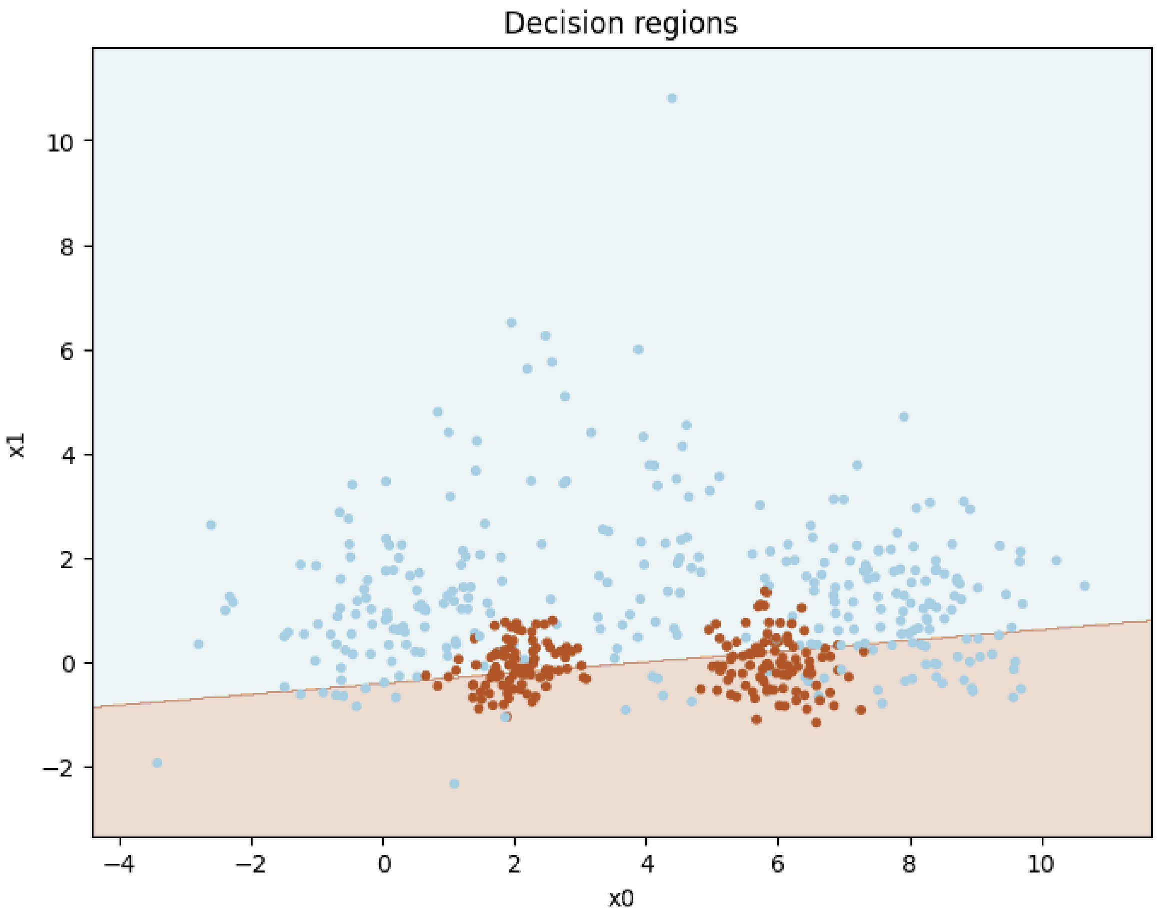
In [183…

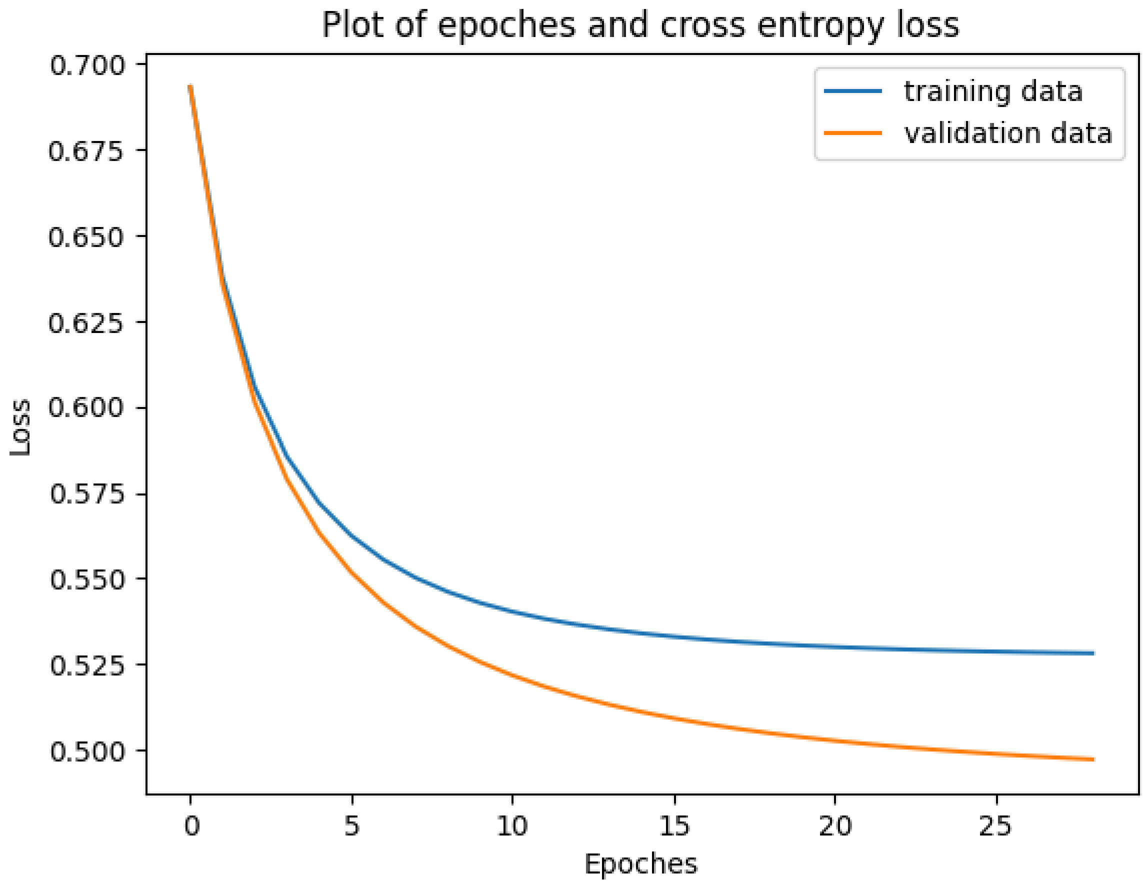
```
cl = NumpyLogRegClass()
cl.fit(X_train_norm, t2_train, epochs=10_000, X_val = X_val_norm, t_val = t2_val
print("Accuracy on the validation set:", accuracy(cl.predict(X_val_norm), t2_val
print(f"Number of epoches trained: {cl.epoch_trained}")

plot_decision_regions(X_val, t2_val, cl)


plt.figure()
plt.plot(cl.losses, label = "training data")
plt.plot(cl.val_losses, label = "validation data")
plt.xlabel('Epoches')
plt.ylabel('Loss')
plt.title('Plot of epoches and cross entropy loss')
plt.legend()

plt.figure()
plt.plot(cl.accuracies, label = "training data")
plt.plot(cl.val_accuracies, label = "validation data")
plt.xlabel('Epoches')
plt.ylabel('Accuracies')
plt.title('Plot of epoches and accuracy')
plt.legend()
```

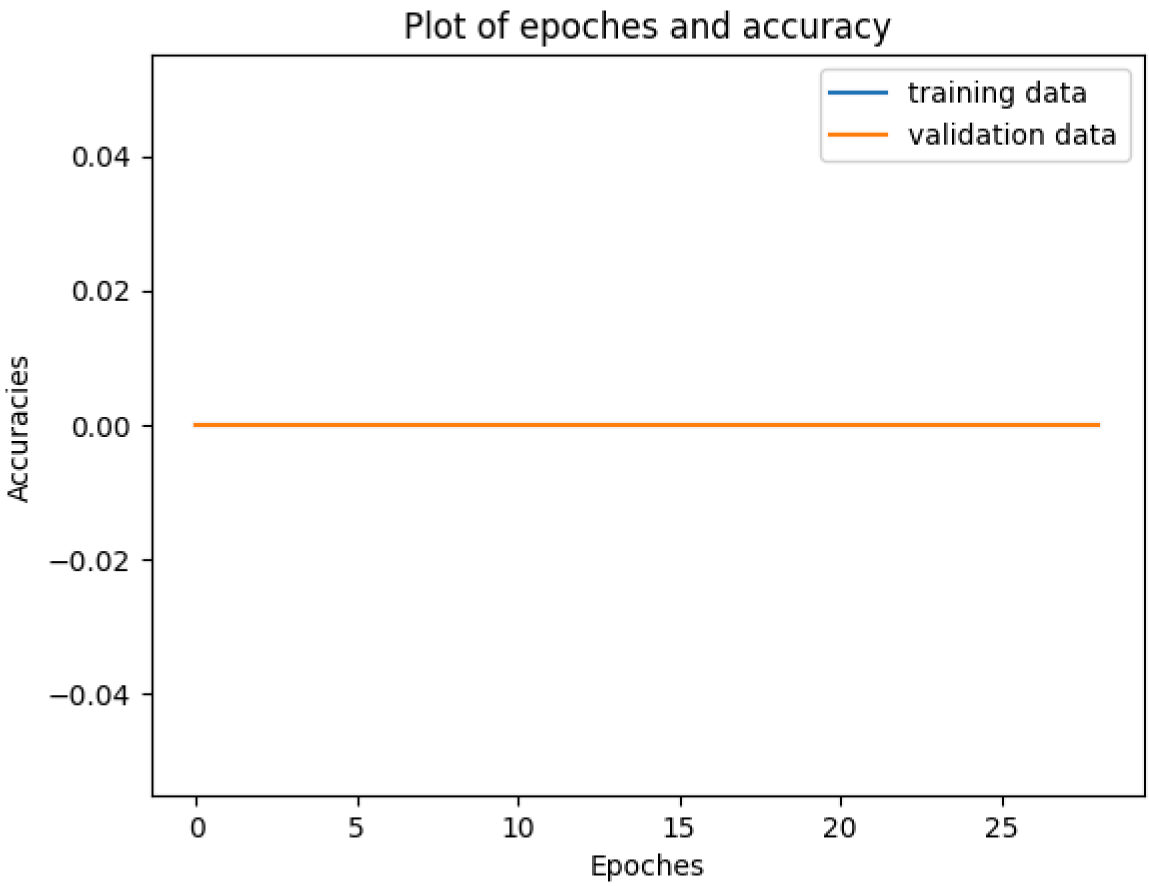
```
Accuracy on the validation set: 0.764
Number of epoches trained: 29
```

Out[183…

```
<matplotlib.legend.Legend at 0x225afc04580>
```







The loss is strictly decreasing, as is expected, but slows down at the end, so in the last epochs there are not much change.

The accuracy graph is wrong, it should be a graph thats increasing, but slows dont on the end. The accuracy is not neceserraly strictly increasing, as the classifier decreases the loss for each epoch, but should in general go upwards.

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 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.

In [184…

```
class NumpyOnevRest(NumpyLogRegClass):

    def __init__(self):
        self.models = []
        self.predictions = []

    def fit_multiclass(self, X_train, t_train, lr = 0.1, epochs=10, X_val = None
        classes = np.unique(t_train)
        for c in classes:
            tc_train = (t_train == c).astype(int)

            cl = NumpyLogRegClass()
            cl.fit(X_train, tc_train, epochs = epochs, eta = lr, tol = tol)

            self.models.append(cl)

    def predict(self, X):
        self.predictions = [model.predict_probability(X) for model in self.model
        return np.array([max(enumerate(preds), key=lambda x: x[1])[0] for preds
```

In [185…

```
def tune_hyperparams(X, y, X_val, y_val):
    #Setting epochs to 10_000, as tol will stop the looping anyways
    lrs = [10**-4, 10**-3, 10**-2, 10**-1, 1, 2]
    tols = [10**-4, 10**-3, 10**-2, 10**-1, 1, 2]
    best = 0
    hyperparams =()
    epochs_trained = 0
    for lr in lrs:
        for tol in tols:
```

```
            cl= NumpyOnevRest()
            cl.fit_multiclass(X, y, lr = lr, tol=tol, epochs=10_000)
            accur = accuracy(cl.predict(X_val), y_val)
            print(f"Accuracy: {accur}, for the validation set with learning rate
            if accur > best:
                best = accur
                hyperparams = (lr, tol)

    print()
    print("The best hyperparameters was:")
    print(f"Learning rate: {hyperparams[0]} and tolerance: {hyperparams}")
    print(f"Which gave an accuracy of {best}")
```

In [186…

```
#On the data
tune_hyperparams(X_train, t_multi_train, X_val, t_multi_val)
```

```
Accuracy: 0.314, for the validation set with learning rate 0.0001 and tolerance:
0.0001.
Accuracy: 0.314, for the validation set with learning rate 0.0001 and tolerance:
0.001.
Accuracy: 0.31, for the validation set with learning rate 0.0001 and tolerance:
0.01.
Accuracy: 0.31, for the validation set with learning rate 0.0001 and tolerance:
0.1.
Accuracy: 0.31, for the validation set with learning rate 0.0001 and tolerance:
1.
Accuracy: 0.31, for the validation set with learning rate 0.0001 and tolerance:
2.
Accuracy: 0.486, for the validation set with learning rate 0.001 and tolerance:
0.0001.
Accuracy: 0.314, for the validation set with learning rate 0.001 and tolerance:
0.001.
Accuracy: 0.314, for the validation set with learning rate 0.001 and tolerance:
0.01.
Accuracy: 0.31, for the validation set with learning rate 0.001 and tolerance: 0.
1.
Accuracy: 0.31, for the validation set with learning rate 0.001 and tolerance: 1.
Accuracy: 0.31, for the validation set with learning rate 0.001 and tolerance: 2.
Accuracy: 0.78, for the validation set with learning rate 0.01 and tolerance: 0.0
001.
Accuracy: 0.486, for the validation set with learning rate 0.01 and tolerance: 0.
001.
Accuracy: 0.314, for the validation set with learning rate 0.01 and tolerance: 0.
01.
Accuracy: 0.318, for the validation set with learning rate 0.01 and tolerance: 0.
1.
Accuracy: 0.314, for the validation set with learning rate 0.01 and tolerance: 1.
Accuracy: 0.314, for the validation set with learning rate 0.01 and tolerance: 2.
Accuracy: 0.846, for the validation set with learning rate 0.1 and tolerance: 0.0
001.
Accuracy: 0.782, for the validation set with learning rate 0.1 and tolerance: 0.0
01.
Accuracy: 0.496, for the validation set with learning rate 0.1 and tolerance: 0.0
1.
Accuracy: 0.35, for the validation set with learning rate 0.1 and tolerance: 0.1.
Accuracy: 0.344, for the validation set with learning rate 0.1 and tolerance: 1.
Accuracy: 0.344, for the validation set with learning rate 0.1 and tolerance: 2.
Accuracy: 0.414, for the validation set with learning rate 1 and tolerance: 0.000
1.
Accuracy: 0.41, for the validation set with learning rate 1 and tolerance: 0.001.
Accuracy: 0.392, for the validation set with learning rate 1 and tolerance: 0.01.
Accuracy: 0.386, for the validation set with learning rate 1 and tolerance: 0.1.
Accuracy: 0.326, for the validation set with learning rate 1 and tolerance: 1.
Accuracy: 0.326, for the validation set with learning rate 1 and tolerance: 2.
Accuracy: 0.394, for the validation set with learning rate 2 and tolerance: 0.000
1.
Accuracy: 0.386, for the validation set with learning rate 2 and tolerance: 0.00
1.
Accuracy: 0.362, for the validation set with learning rate 2 and tolerance: 0.01.
Accuracy: 0.364, for the validation set with learning rate 2 and tolerance: 0.1.
Accuracy: 0.37, for the validation set with learning rate 2 and tolerance: 1.
Accuracy: 0.37, for the validation set with learning rate 2 and tolerance: 2.

The best hyperparameters was:
Learning rate: 0.1 and tolerance: (0.1, 0.0001)
Which gave an accuracy of 0.846
```

In [187…

```
#On the scaled data
tune_hyperparams(X_train_norm, t_multi_train, X_val_norm, t_multi_val)
```

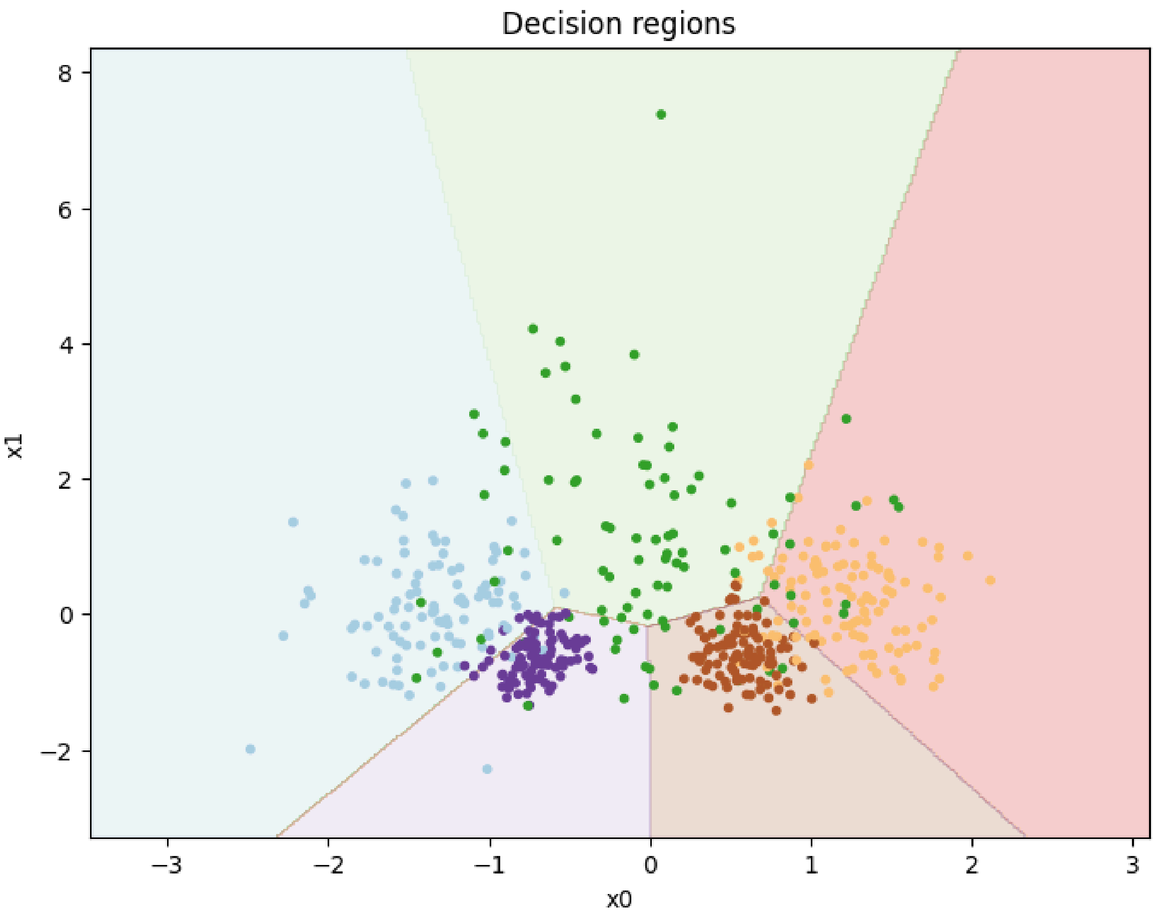
```
Accuracy: 0.708, for the validation set with learning rate 0.0001 and tolerance:
0.0001.
Accuracy: 0.708, for the validation set with learning rate 0.0001 and tolerance:
0.001.
Accuracy: 0.708, for the validation set with learning rate 0.0001 and tolerance:
0.01.
Accuracy: 0.708, for the validation set with learning rate 0.0001 and tolerance:
0.1.
Accuracy: 0.708, for the validation set with learning rate 0.0001 and tolerance:
1.
Accuracy: 0.708, for the validation set with learning rate 0.0001 and tolerance:
2.
Accuracy: 0.778, for the validation set with learning rate 0.001 and tolerance:
0.0001.
Accuracy: 0.708, for the validation set with learning rate 0.001 and tolerance:
0.001.
Accuracy: 0.708, for the validation set with learning rate 0.001 and tolerance:
0.01.
Accuracy: 0.708, for the validation set with learning rate 0.001 and tolerance:
0.1.
Accuracy: 0.708, for the validation set with learning rate 0.001 and tolerance:
1.
Accuracy: 0.708, for the validation set with learning rate 0.001 and tolerance:
2.
Accuracy: 0.808, for the validation set with learning rate 0.01 and tolerance: 0.
0001.
Accuracy: 0.778, for the validation set with learning rate 0.01 and tolerance: 0.
001.
Accuracy: 0.708, for the validation set with learning rate 0.01 and tolerance: 0.
01.
Accuracy: 0.708, for the validation set with learning rate 0.01 and tolerance: 0.
1.
Accuracy: 0.708, for the validation set with learning rate 0.01 and tolerance: 1.
Accuracy: 0.708, for the validation set with learning rate 0.01 and tolerance: 2.
Accuracy: 0.842, for the validation set with learning rate 0.1 and tolerance: 0.0
001.
Accuracy: 0.808, for the validation set with learning rate 0.1 and tolerance: 0.0
01.
Accuracy: 0.776, for the validation set with learning rate 0.1 and tolerance: 0.0
1.
Accuracy: 0.708, for the validation set with learning rate 0.1 and tolerance: 0.
1.
Accuracy: 0.708, for the validation set with learning rate 0.1 and tolerance: 1.
Accuracy: 0.708, for the validation set with learning rate 0.1 and tolerance: 2.
Accuracy: 0.84, for the validation set with learning rate 1 and tolerance: 0.000
1.
Accuracy: 0.84, for the validation set with learning rate 1 and tolerance: 0.001.
Accuracy: 0.81, for the validation set with learning rate 1 and tolerance: 0.01.
Accuracy: 0.772, for the validation set with learning rate 1 and tolerance: 0.1.
Accuracy: 0.726, for the validation set with learning rate 1 and tolerance: 1.
Accuracy: 0.726, for the validation set with learning rate 1 and tolerance: 2.
Accuracy: 0.842, for the validation set with learning rate 2 and tolerance: 0.000
1.
Accuracy: 0.84, for the validation set with learning rate 2 and tolerance: 0.001.
Accuracy: 0.822, for the validation set with learning rate 2 and tolerance: 0.01.
Accuracy: 0.788, for the validation set with learning rate 2 and tolerance: 0.1.
Accuracy: 0.77, for the validation set with learning rate 2 and tolerance: 1.
Accuracy: 0.77, for the validation set with learning rate 2 and tolerance: 2.

The best hyperparameters was:
```

```
Learning rate: 0.1 and tolerance: (0.1, 0.0001)
Which gave an accuracy of 0.842
```

In [188…

```
cl= NumpyOnevRest()
cl.fit_multiclass(X_train_norm, t_multi_train, lr = 0.1, tol=0.0001, epochs=10_0
accur = accuracy(cl.predict(X_val_norm), t_multi_val)
print("Accuracy on the validation set:", accuracy(cl.predict(X_val_norm), t_mult
plot_decision_regions(X_val_norm, t_multi_val, cl)
```

```
Accuracy on the validation set: 0.842
```



Multinomial logistic regression

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 one-vs-rest classifier. (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 [189…

```
class MultinomialLogisticReg(NumpyClassifier):

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

```
    def softmax(self, logits):
        exp_logits = np.exp(logits - np.max(logits, axis=1, keepdims=True))
        return exp_logits / np.sum(exp_logits, axis=1, keepdims=True)

    def cross_entropy_loss(self, predictions, labels):
        m = labels.shape[0]
        log_likelihood = -np.log(predictions[range(m), labels])
        return np.sum(log_likelihood) / m

    def fit(self, X_train, t_train, lr=0.1, epochs=10, X_val = None, t_val = Non

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

        (N, M) = X_train.shape
        num_classes = len(np.unique(t_train))

        self.weights = weights = np.zeros((M, num_classes))
        self.biases = biases = np.zeros((1, num_classes))

        for _ in range(epochs):

            logits = np.dot(X_train, weights) + biases
            predictions = self.softmax(logits)

            #loss = self.cross_entropy_loss(predictions, t_train)


            y_one_hot = np.eye(num_classes)[t_train]

            grad_W = (1/N) * np.dot(X_train.T, (predictions - y_one_hot))
            grad_b = (1/N) * np.sum(predictions - y_one_hot, axis=0, keepdims=Tr


            weights -= lr * grad_W
            biases -= lr * grad_b


    def predict(self, X):
        """Predict class labels for samples in X."""
        if self.bias:
            X = add_bias(X, self.bias)
        logits = np.dot(X, self.weights) + self.biases
        predictions = self.softmax(logits)
        return np.argmax(predictions, axis=1)
```

In [190…

```
lrs = [10**-4, 10**-3, 10**-2, 10**-1, 1, 2]
epochs = [1, 2, 10, 10**3, 10**4, 10**5]
best = 0
hyperparams = ()
epochs_trained = 0
for lr in lrs:
    for epoch in epochs:
        cl= MultinomialLogisticReg()
        cl.fit(X_train_norm, t_multi_train, lr = lr, epochs=epoch)
        accur = accuracy(cl.predict(X_val_norm), t_multi_val)
        #print(f"Accuracy: {accur}, for the validation set with learning rate {l
        if accur > best:
            best = accur
```

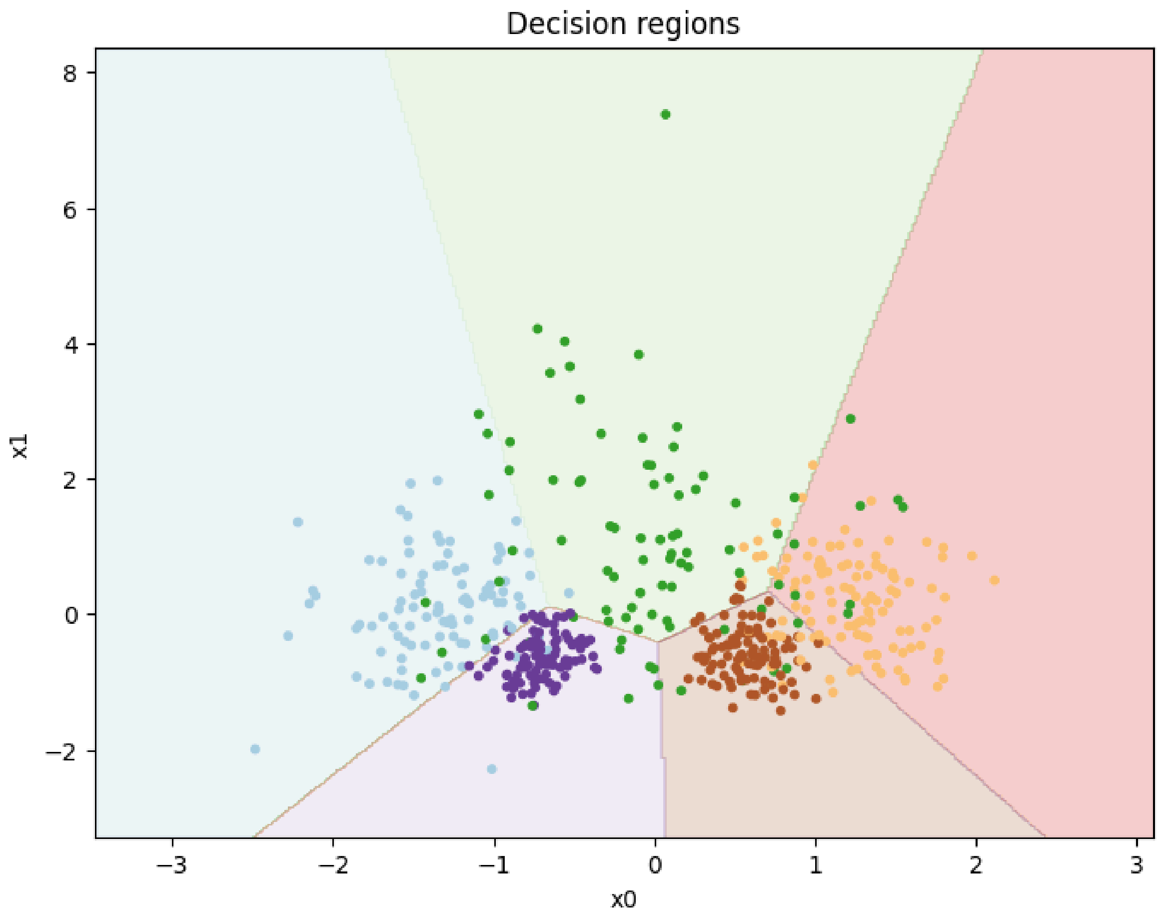
```
            hyperparams = (lr, epoch)
print()
print("The best hyperparameters was:")
print(f"Learning rate: {hyperparams[0]} and epochs: {hyperparams[1]}")
print(f"Which gave an accuracy of {best}")
```

```
The best hyperparameters was:
Learning rate: 0.001 and epochs: 100000
Which gave an accuracy of 0.854
```

In [191…

```
cl= MultinomialLogisticReg()
cl.fit(X_train_norm, t_multi_train, lr = 0.001 , epochs=100_000)
accur = accuracy(cl.predict(X_val_norm), t_multi_val)
print("Accuracy on the validation set:", accuracy(cl.predict(X_val_norm), t_mult
plot_decision_regions(X_val_norm, t_multi_val, cl)
```

```
Accuracy on the validation set: 0.854
```



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. 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 [192…

```
# 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 [193…

```
#Since MLP involves randomness, we set a seed to make this code and results repr
np.random.seed(42)
```

In [232…

```
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

        #storage:
        self.bias=bias



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

        outputs = self.activ(hidden_outs @ self.weights2)

        return hidden_outs, outputs

    def fit(self, X_train, t_train, lr=0.001, epochs = 100):
        """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 training data,
        where the values are 0 or 1.
        lr is the learning rate
        """
        self.lr = lr

        # Turn t_train into a column vector, a N*1 matrix:
```

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

        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)

        for e in range(epochs):
            # One epoch
            # The forward step:
            hidden_outs, outputs = self.forward(X_train_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:]))

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


    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_probability(self, X):
        Z = add_bias(X, self.bias)
        forw = self.forward(Z)[1]
        return forw[:, 0]
```

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.

In [235…

```
def test_hyperparams(X, t, X_val, t_val):
    lrs = [10**-4, 0.0005, 10**-3, 10**-2, 10**-1, 1, 2]
    n_epochs = [8000, 4000, 5000, 1000,500, 100, 20, 10,]
    dims = [2,6,10,15, 20]

    best = 0
    hyperparams = ()
    for lr in lrs:
```

```
        for epochs in n_epochs:
            for dim in dims:
                cl= MLPBinaryLinRegClass(dim_hidden=dim)
                cl.fit(X, t, lr = lr, epochs=epochs)
                accur = accuracy(cl.predict(X_val), t_val)
                #print(f"Accuracy: {accur}, for the validation set with learning
                if accur > best:
                    best = accur
                    hyperparams = (lr, epochs, dim)

    return hyperparams, best
```

In [236…

```
test_hyperparams(X_train_norm, t2_train, X_val_norm, t2_val)
```

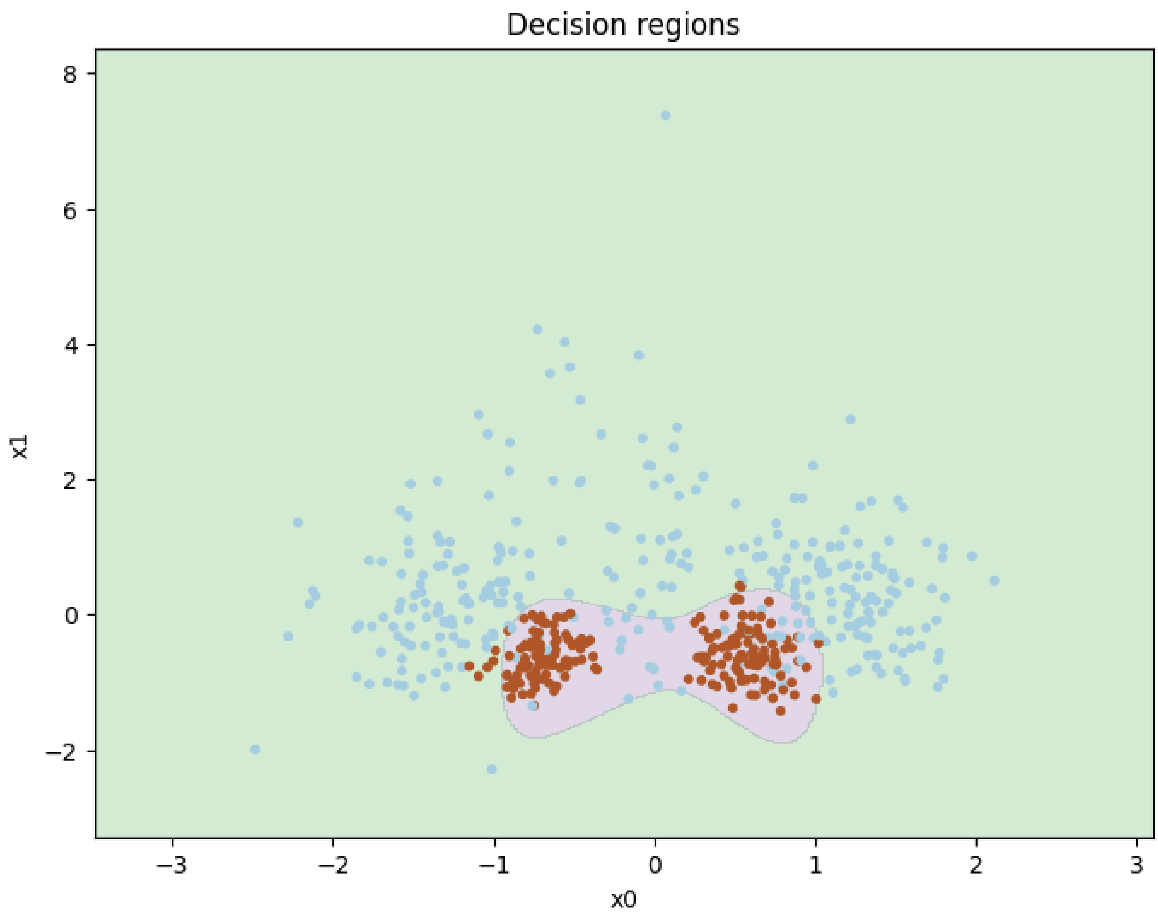
```
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\702837997.py:3: RuntimeWarning:
overflow encountered in exp
  return 1/(1+np.exp(-x))
```

Out[236… ((0.001, 8000, 20), 0.926)

In [237…

```
cl = MLPBinaryLinRegClass(dim_hidden=20)
cl.fit(X_train_norm, t2_train, lr = 0.001, epochs=8000)
print("Accuracy on the validation set:", accuracy(cl.predict(X_val_norm), t2_val

plot_decision_regions(X_val_norm, t2_val, cl)
```

```
Accuracy on the validation set: 0.906
```



The non-linear classifier is a lot better on this dataset.

The desicion region and accuracy is a bit worse that the hyperparameter-testing said, but that may be because of the randomness

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_probability()` 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 5. 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 10 times with the optimal hyper-parameters and report the mean and standard deviation of the accuracies over the 10 runs.

In [240…

```
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

        #storage:
        self.bias=bias
        self.losses = []
        self.accuracies = []
```

```
        self.val_losses = []
        self.val_accuracies = []
        self.epoch_trained = 0

    def cross_entropy_loss(y_pred, y_true):
        loss = -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pr
        return loss

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

        outputs = self.activ(hidden_outs @ self.weights2)

        return hidden_outs, outputs

    def fit(self, X_train, t_train, lr=0.001, epochs = 100, X_val = None, t_val
        """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 training data,
        where the values are 0 or 1.
        lr is the learning rate
        """
        self.lr = lr

        # Turn t_train into a column vector, a N*1 matrix:
        T_train = t_train.reshape(-1,1)

        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)

        for e in range(epochs):

            # The forward step:
            hidden_outs, outputs = self.forward(X_train_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:]))

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

```
            self.weights1 -= self.lr * X_train_bias.T @ hiddenact_deltas

            #store accuracies and losses
            accur = accuracy(self.predict(X_train), t_train)
            self.accuracies.append(accur)

            loss = -np.mean(t_train * np.log(self.predict_probability(X_train))
            self.losses.append(loss)

            if X_val is not None:
                val_accur = accuracy(self.predict(X_val), t_val)
                self.val_accuracies.append(val_accur)
                val_loss = -np.mean(t_val * np.log(self.predict_probability(X_va
                self.val_losses.append(val_loss)

            #break if exceeding tolerance limit
            if len(self.losses) > n_epochs_no_update and self.losses[-n_epochs_n
                break
        self.epoch_trained = len(self.accuracies)


    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_probability(self, X):
        Z = add_bias(X, self.bias)
        forw = self.forward(Z)[1]
        return forw[:, 0]
```

In [220…

```
def test_hyperparams(X, t, X_val, t_val):
    lrs = [10**-4, 0.0005, 10**-3, 0.005, 10**-2, 10**-1]
    tols = [10**-4, 0.0005, 10**-3, 0.005, 10**-2]
    dims = [2,6,10,15, 20]

    best = 0
    hyperparams = ()
    for lr in lrs:
        for tol in tols:
            for dim in dims:
                cl= MLPBinaryLinRegClass(dim_hidden=dim)
                cl.fit(X, t, lr = lr, epochs=10_000, tol=tol, X_val=X_val, t_val
                accur = accuracy(cl.predict(X_val), t_val)

                if accur > best:
                    best = accur
                    hyperparams = (lr, tol, dim)

    return hyperparams, best
```

In [221…

```
test_hyperparams(X_train, t2_train, X_val, t2_val)
```

```
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\702837997.py:3: RuntimeWarning:
overflow encountered in exp
  return 1/(1+np.exp(-x))
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\77964951.py:82: RuntimeWarning:
divide by zero encountered in log
  loss = -np.mean(t_train * np.log(self.predict_probability(X_train)) + (1 - t_tr
ain) * np.log(1 - self.predict_probability(X_train)))
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\77964951.py:82: RuntimeWarning:
invalid value encountered in multiply
  loss = -np.mean(t_train * np.log(self.predict_probability(X_train)) + (1 - t_tr
ain) * np.log(1 - self.predict_probability(X_train)))
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\77964951.py:88: RuntimeWarning:
divide by zero encountered in log
  val_loss = -np.mean(t_val * np.log(self.predict_probability(X_val)) + (1 - t_va
l) * np.log(1 - self.predict_probability(X_val)))
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\77964951.py:88: RuntimeWarning:
invalid value encountered in multiply
  val_loss = -np.mean(t_val * np.log(self.predict_probability(X_val)) + (1 - t_va
l) * np.log(1 - self.predict_probability(X_val)))
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\77964951.py:92: RuntimeWarning:
invalid value encountered in scalar subtract
  if len(self.losses) > n_epochs_no_update and self.losses[-n_epochs_no_update-1]
-self.losses[-1]<tol:
```

Out[221…

```
((0.0005, 0.0001, 20), 0.906)
```

In [222…

```
#with normalized data:

test_hyperparams(X_train_norm, t2_train, X_val_norm, t2_val)
```

```
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\77964951.py:82: RuntimeWarning:
divide by zero encountered in log
  loss = -np.mean(t_train * np.log(self.predict_probability(X_train)) + (1 - t_tr
ain) * np.log(1 - self.predict_probability(X_train)))
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\77964951.py:82: RuntimeWarning:
invalid value encountered in multiply
  loss = -np.mean(t_train * np.log(self.predict_probability(X_train)) + (1 - t_tr
ain) * np.log(1 - self.predict_probability(X_train)))
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\77964951.py:88: RuntimeWarning:
divide by zero encountered in log
  val_loss = -np.mean(t_val * np.log(self.predict_probability(X_val)) + (1 - t_va
l) * np.log(1 - self.predict_probability(X_val)))
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\77964951.py:88: RuntimeWarning:
invalid value encountered in multiply
  val_loss = -np.mean(t_val * np.log(self.predict_probability(X_val)) + (1 - t_va
l) * np.log(1 - self.predict_probability(X_val)))
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\702837997.py:3: RuntimeWarning:
overflow encountered in exp
  return 1/(1+np.exp(-x))
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\77964951.py:92: RuntimeWarning:
invalid value encountered in scalar subtract
  if len(self.losses) > n_epochs_no_update and self.losses[-n_epochs_no_update-1]
-self.losses[-1]<tol:
```

Out[222…

```
((0.001, 0.0001, 6), 0.906)
```

In [223…

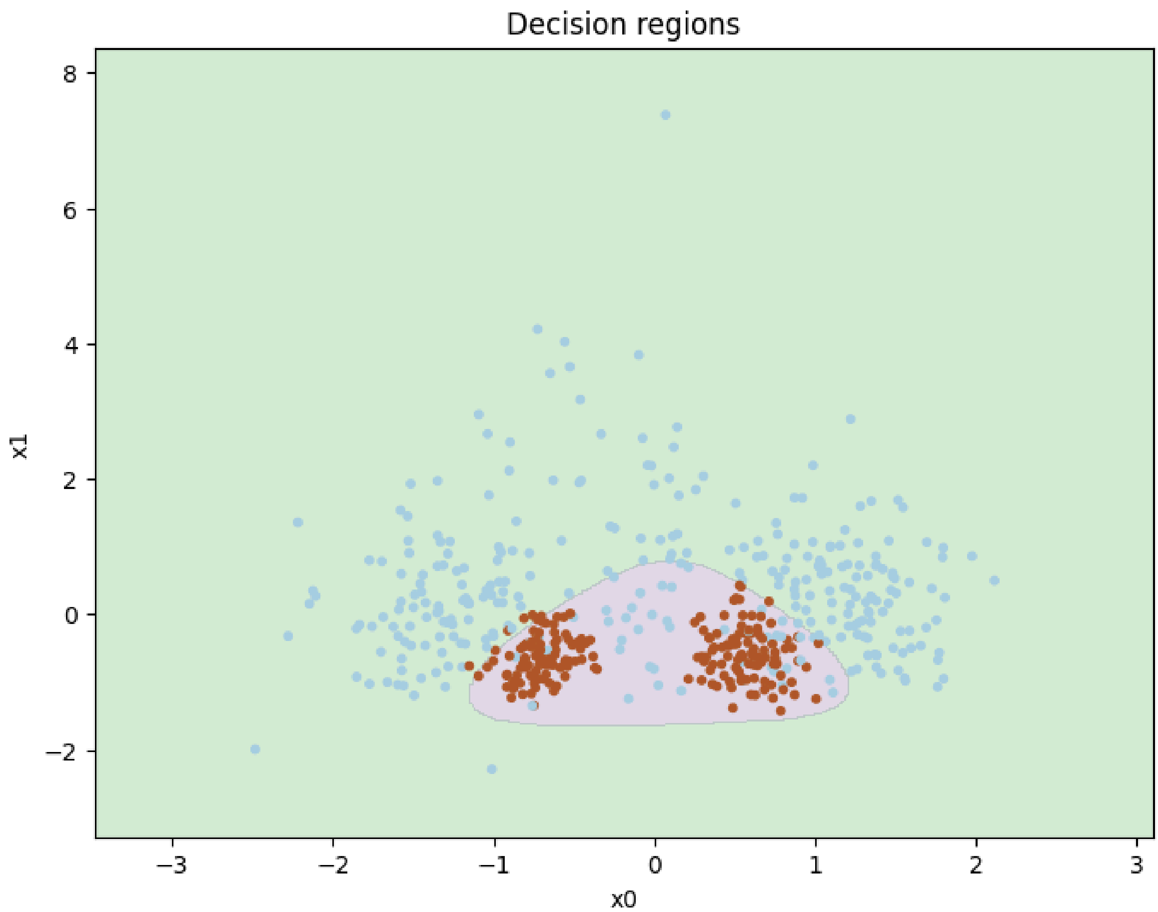
```
cl = MLPBinaryLinRegClass(dim_hidden=6)
cl.fit(X_train_norm, t2_train, lr = 0.001, tol = 0.0001, epochs=10_000, X_val=X_
print("Accuracy on the validation set:", accuracy(cl.predict(X_val_norm), t2_val
#print(f"Number of epoches trained: {cl.epoch_trained}")
```

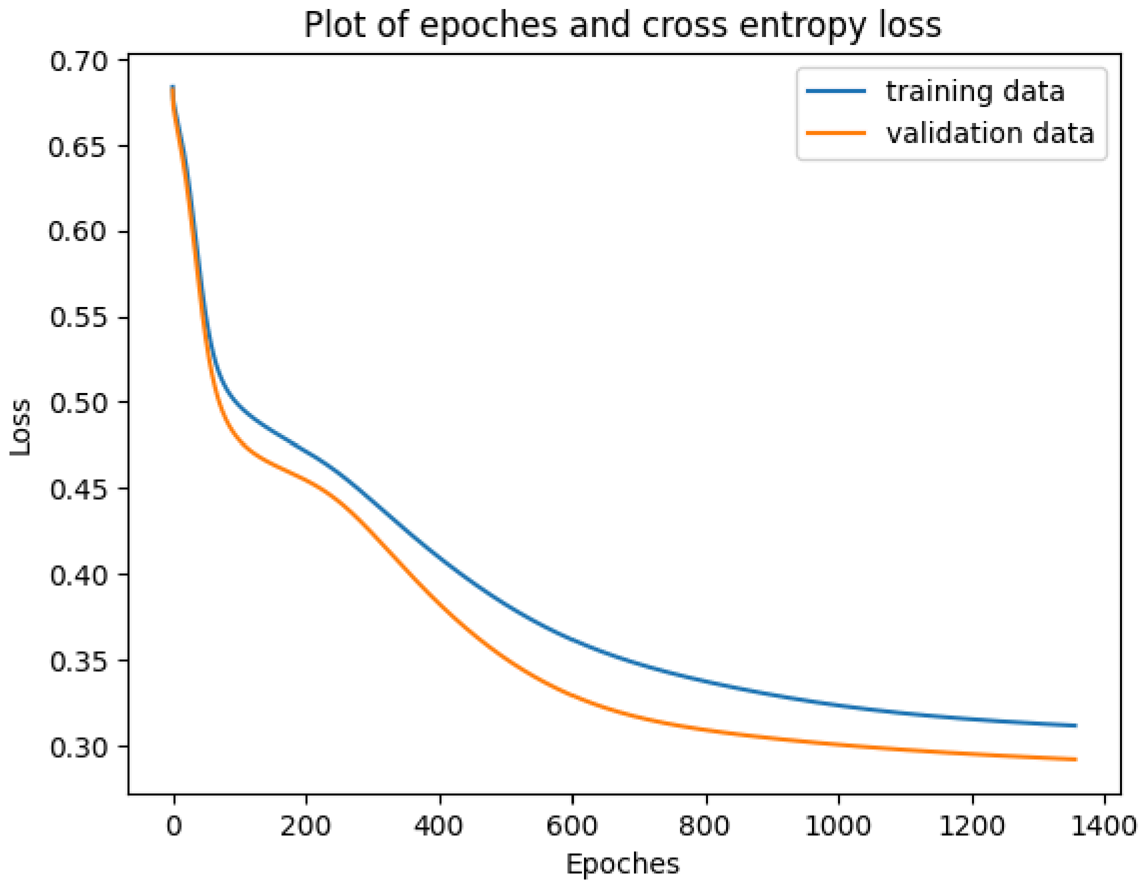
```
plot_decision_regions(X_val_norm, t2_val, cl)


plt.figure()
plt.plot(cl.losses, label = "training data")
plt.plot(cl.val_losses, label = "validation data")
plt.xlabel('Epoches')
plt.ylabel('Loss')
plt.title('Plot of epoches and cross entropy loss')
plt.legend()

plt.figure()
plt.plot(cl.accuracies, label = "training data")
plt.plot(cl.val_accuracies, label = "validation data")
plt.xlabel('Epoches')
plt.ylabel('Accuracies')
plt.title('Plot of epoches and accuracy')
plt.legend()
```

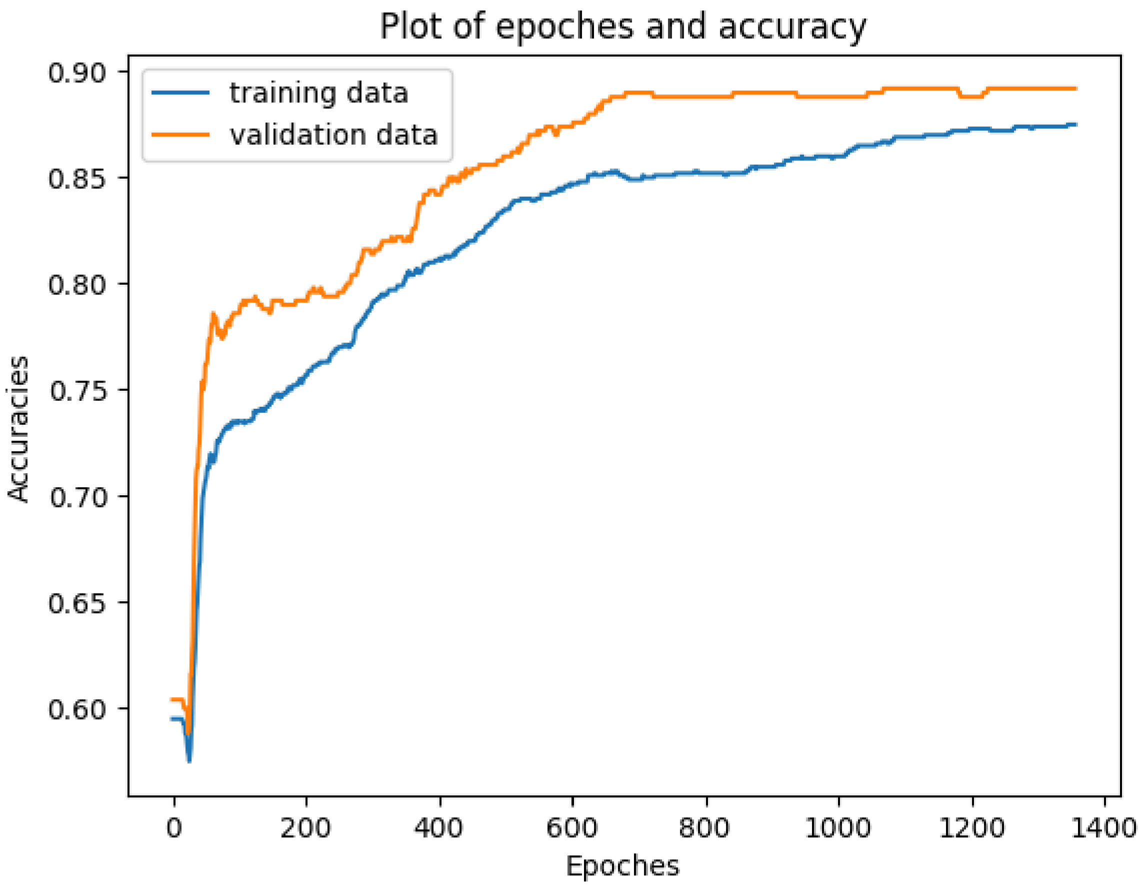
```
Accuracy on the validation set: 0.892
```

Out[223…

```
<matplotlib.legend.Legend at 0x225b63dd930>
```







The losses are decreasing. It is unusual that the validation data performs better, but in this case it does.

The accuracies are more "rugged", but that makes sense. Remember that the goal of the classifier is to minimize the loss function, so the loss function should be smooth downwards, while the accuracies may jump a little, but still get larger and larger each time

In [224…

```
accuracies = []
for _ in range(10):
    cl = MLPBinaryLinRegClass(dim_hidden=6)
    cl.fit(X_train_norm, t2_train, lr = 0.001, tol = 0.0001, epochs=10_000, X_va
    accuracies.append(accuracy(cl.predict(X_val_norm), t2_val))

print(f"Mean accuracy: {np.mean(accuracies)} \nStandard deviation: {np.std(accur

accuracies
```

```
Mean accuracy: 0.901
Standard deviation: 0.0037148351242013446
```

Out[224…

```
[0.898, 0.902, 0.906, 0.9, 0.904, 0.904, 0.892, 0.902, 0.902, 0.9]
```

The mean accuracy is 0.901, and all the results are similar (low standard deviation)

Multi-class neural network

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 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.

In [225…

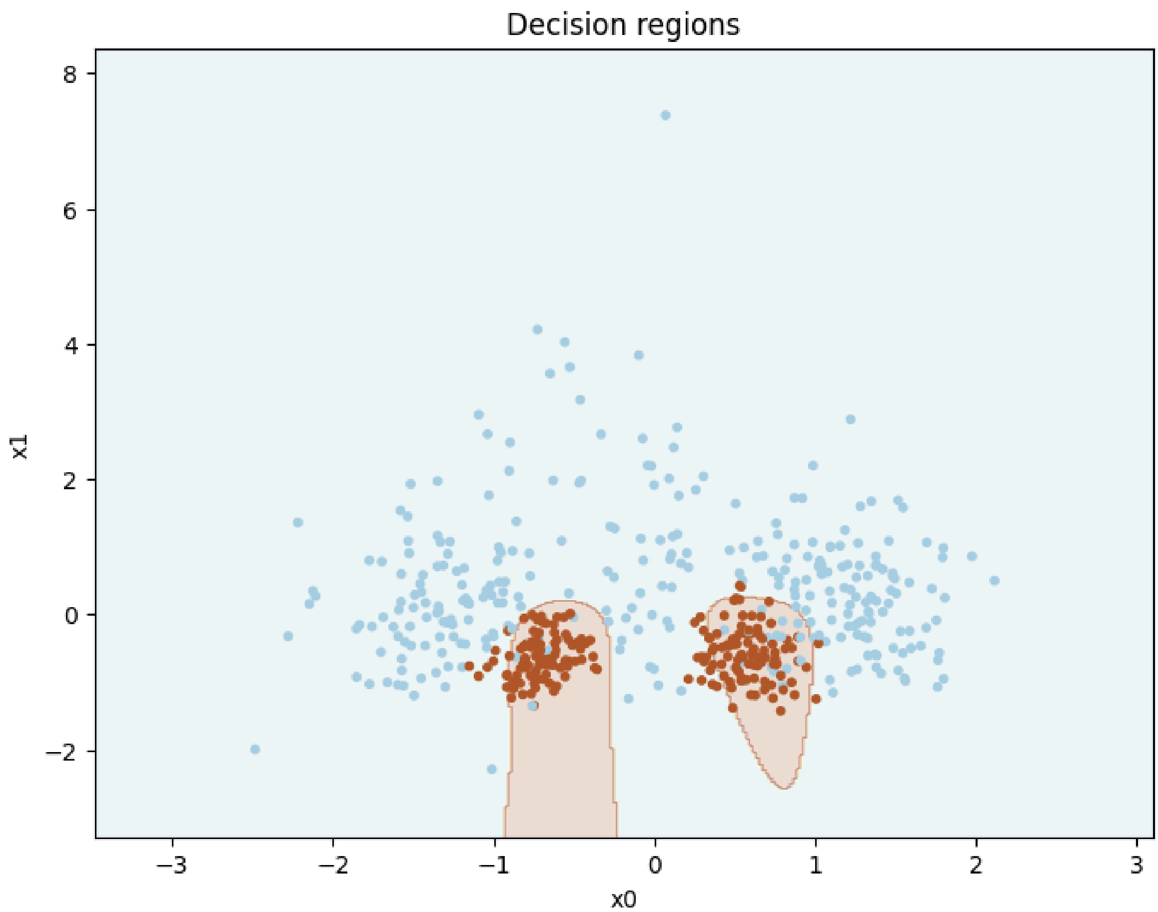
```
def softmax(x):
    exps = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exps / np.sum(exps, axis=1, keepdims=True)

class MultiNNClass(NumpyClassifier):
    """A multi-layer neural network with one hidden layer for multiclass classif

    def __init__(self, bias=-1, dim_hidden=6):
        self.bias = bias
        self.dim_hidden = dim_hidden
        self.activ = logistic
        self.activ_diff = logistic_diff

    def forward(self, X):
        hidden_outs = self.activ(X @ self.weights1)
        hidden_outs = add_bias(hidden_outs, self.bias)
        outputs = softmax(hidden_outs @ self.weights2)
        return hidden_outs, outputs
```

```
    def fit(self, X_train, t_train, lr=0.001, epochs=100):
        self.lr = lr
        T_train = self.one_hot_encode(t_train)
        dim_in = X_train.shape[1]
        dim_out = T_train.shape[1]

        self.weights1 = (np.random.rand(dim_in + 1, self.dim_hidden) * 2 - 1) /
        self.weights2 = (np.random.rand(self.dim_hidden + 1, dim_out) * 2 - 1) /
        X_train_bias = add_bias(X_train, self.bias)

        for e in range(epochs):
            hidden_outs, outputs = self.forward(X_train_bias)
            out_deltas = (outputs - T_train)
            hiddenout_diffs = out_deltas @ self.weights2.T
            hiddenact_deltas = hiddenout_diffs[:, 1:] * self.activ_diff(hidden_c
            self.weights2 -= self.lr * hidden_outs.T @ out_deltas
            self.weights1 -= self.lr * X_train_bias.T @ hiddenact_deltas

    def predict(self, X):
        Z = add_bias(X, self.bias)
        outputs = self.forward(Z)[1]
        return np.argmax(outputs, axis=1)

    def predict_probability(self, X):
        Z = add_bias(X, self.bias)
        outputs = self.forward(Z)[1]
        return outputs

    def one_hot_encode(self, t):
        n_classes = np.max(t) + 1
        return np.eye(n_classes)[t]
```

In [226…

```
#sanity check:
cl = MultiNNClass(dim_hidden=6)
cl.fit(X_train_norm, t2_train, lr = 0.005, epochs=10_000)
print("Accuracy on the validation set:", accuracy(cl.predict(X_val_norm), t2_val
#print(f"Number of epoches trained: {cl.epoch_trained}")

plot_decision_regions(X_val_norm, t2_val, cl)
```

```
Accuracy on the validation set: 0.902
```



Looks nice

In [227…

```
def test_hyperparams(X, t, X_val, t_val):
    lrs = [10**-4, 0.0005, 10**-3, 10**-2, 10**-1]
    epochs = [10, 100, 10**3, 10**4]
    dims = [2,6,10]

    best = 0
    hyperparams = ()
    for lr in lrs:
        for epoch in epochs:
            for dim in dims:
                cl= MultiNNClass(dim_hidden=dim)
                cl.fit(X, t, lr = lr, epochs=epoch)
                accur = accuracy(cl.predict(X_val), t_val)

                if accur > best:
                    best = accur
                    hyperparams = (lr, epoch, dim)

    return hyperparams, best
```

In [228…

```
test_hyperparams(X_train, t_multi_train, X_val, t_multi_val)
```

```
C:\Users\hanne\AppData\Local\Temp\ipykernel_20800\702837997.py:3: RuntimeWarning: overflow encountered in exp
  return 1/(1+np.exp(-x))
```

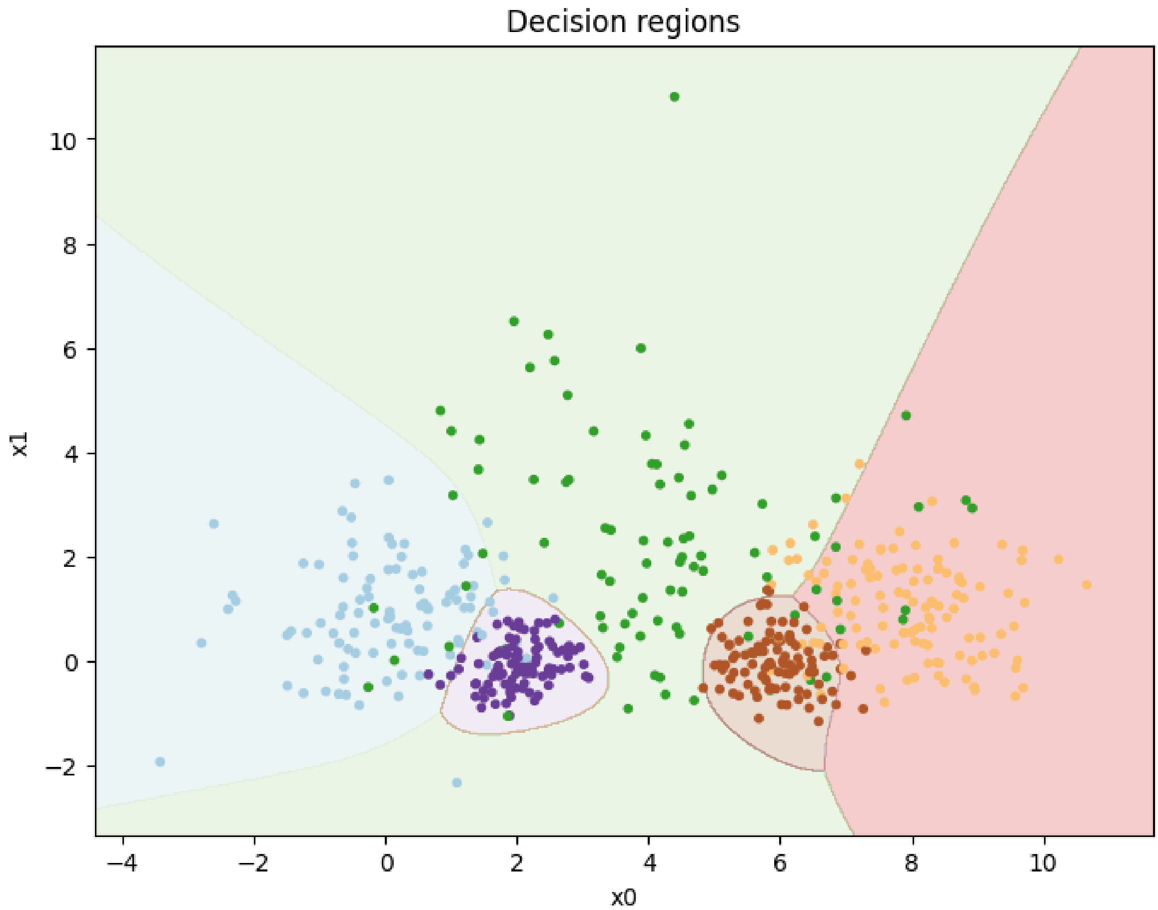
Out[228…

```
((0.001, 10000, 6), 0.888)
```

The best accuracy we got was 0.88, which mean 88,8% of the instances were classified accuratly

In [229…

```
cl = MultiNNClass(dim_hidden=6)
cl.fit(X_train, t_multi_train, lr = 0.001, epochs=10_000)
plot_decision_regions(X_val, t_multi_val, cl)
```



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 [265…

```
#used the best hyperparameters from earlier testing. Use normalized dataset on a

linear_cl = NumpyLinRegClass()
linear_cl.fit(X_train_norm, t2_train, lr = 0.001, epochs=4000)
linear_accuracies = [accuracy(linear_cl.predict(X_train_norm), t2_train), accura


logistic_cl = NumpyLogRegClass()
logistic_cl.fit(X_train_norm, t2_train, epochs=10_000, eta = 1, tol=10**-3)
logistic_accuracies = [accuracy(logistic_cl.predict(X_train_norm), t2_train), ac

multi_layer_cl = MLPBinaryLinRegClass(dim_hidden=6)
multi_layer_cl.fit(X_train_norm, t2_train, lr = 0.001, tol=0.0001, epochs=10_000
multi_layer_accuracies = [accuracy(multi_layer_cl.predict(X_train_norm), t2_trai
```

In [266…

```
import pandas as pd
scores = list(zip(linear_accuracies , logistic_accuracies, multi_layer_accuracie
header = ['', 'Linear', 'Logistic', 'Multilayer']
arr = np.array(scores)
ekstra = np.array(["Training", "Validation", "Test"])
arr_med_ekstra = np.column_stack((ekstra,arr))
print(pd.DataFrame(arr_med_ekstra, columns=header))
```

```
                Linear Logistic Multilayer
0     Training   0.716    0.719      0.884
1   Validation   0.754    0.764        0.9
2         Test   0.722    0.722       0.89
```

Multilayer performs the best, which is expected as the data was not linear. The test set results are good, which means that we have not overfitted our model

In [268…

```
#Precition and recall:
from sklearn.metrics import precision_score, recall_score

linear_preds = [linear_cl.predict(X_train_norm), linear_cl.predict(X_val_norm),
logistic_preds = [logistic_cl.predict(X_train_norm), logistic_cl.predict(X_val_n
multi_layer_preds = [multi_layer_cl.predict(X_train_norm), multi_layer_cl.predic


# Calculate Precision
precisions_linear = [precision_score(t2_train, linear_preds[0]), precision_score
precisions_logistic = [precision_score(t2_train, logistic_preds[0]), precision_s
precisions_multi_layer = [precision_score(t2_train, multi_layer_preds[0]), preci

# Calculate recall
recall_linear = [recall_score(t2_train, linear_preds[0]), recall_score(t2_val, l
recall_logistic = [recall_score(t2_train, logistic_preds[0]), recall_score(t2_va
recall_multi_layer = [recall_score(t2_train, multi_layer_preds[0]), recall_score


print("Precision scores")

scores = list(zip(precisions_linear , precisions_logistic, precisions_multi_laye
header = ['', 'Linear', 'Logistic', 'Multilayer']
arr = np.array(scores)
ekstra = np.array(["Training", "Validation", "Test"])
arr_med_ekstra = np.column_stack((ekstra,arr))
print(pd.DataFrame(arr_med_ekstra, columns=header))
```

```
print("\n")
print("Recall scores")

scores = list(zip(recall_linear , recall_logistic, recall_multi_layer))
header = ['', 'Linear', 'Logistic', 'Multilayer']
arr = np.array(scores)
ekstra = np.array(["Training", "Validation", "Test"])
arr_med_ekstra = np.column_stack((ekstra,arr))
print(pd.DataFrame(arr_med_ekstra, columns=header))
```

```
Precision scores
                        Linear            Logistic          Multilayer
0    Training  0.6630727762803235  0.6557788944723618  0.8016701461377871
1  Validation  0.6963350785340314   0.696078431372549  0.8217391304347826
2        Test  0.6593406593406593  0.6526315789473685  0.8034188034188035


Recall scores
                        Linear            Logistic          Multilayer
0    Training  0.6074074074074074  0.6444444444444445  0.9481481481481482
1  Validation  0.6717171717171717  0.7171717171717171  0.9545454545454546
2        Test  0.6091370558375635  0.6294416243654822  0.9543147208121827
```

Multi-class task (X, t_multi)

Compare the three multi-class classifiers, 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 the results.

In [248…

```
onevrest = NumpyOnevRest()
onevrest.fit_multiclass(X_train_norm, t_multi_train, lr = 0.1, tol=0.001)
onevrest_accuracies = [accuracy(onevrest.predict(X_train_norm), t_multi_train),

multinomial = MultinomialLogisticReg()
multinomial.fit(X_train_norm, t_multi_train, lr = 0.01, epochs=100_000)
multinomial_accuracies = [accuracy(multinomial.predict(X_train_norm), t_multi_tr

NNclass = MultiNNClass(dim_hidden=6)
NNclass.fit(X_train_norm, t_multi_train, lr = 0.001, epochs=10_000)
NNclass_accuracies = [accuracy(NNclass.predict(X_train_norm), t_multi_train), ac
```

In [270…

```
scores = list(zip(onevrest_accuracies , multinomial_accuracies, NNclass_accuraci
header = ['', 'One v Rest', 'Multinomial Logistic', 'Neural Network']
arr = np.array(scores)
ekstra = np.array(["Training", "Validation", "Test"])
arr_med_ekstra = np.column_stack((ekstra,arr))
print(pd.DataFrame(arr_med_ekstra, columns=header))
```

```
               One v Rest Multinomial Logistic Neural Network
0    Training      0.673                0.853          0.865
1  Validation       0.71                0.836          0.864
2        Test       0.65                0.834          0.838
```

All the results on the test set is worse than training and validation. There is not that much of a difference, so the models seems ok in terms of overfitting. The multiclass neural network classifier performs the best.

Good luck!