in3050_in4050_2022_assignment_2

March 28, 2022

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

0.1.1 Rules

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

0.1.2 Delivery

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

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

0.1.3 What to deliver?

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

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

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

Your report/notebook should contain your name and username.

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

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

0.1.4 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.

0.1.5 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.

0.1.6 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!

0.1.7 Initialization

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

1 Part 1: Linear classifiers

1.1 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)`.

[2]:
```
from sklearn.datasets import make_blobs
X, t = make_blobs(n_samples=[400,400,400, 400, 400],␣
 ↪centers=[[0,1],[4,1],[8,1],[2,0],[6,0]],
                  n_features=2, random_state=2019, cluster_std=1.0)
# t is a the length of the array 2000 but composed of numbers inbetween 0 to 4,␣
 ↪aka the n-samples
```

[3]:
```
indices = np.arange(X.shape[0]) # goes from 0 to 1999
rng = np.random.RandomState(2022)
rng.shuffle(indices) #just shuffles the order of the X.shape
indices[:10]
```

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

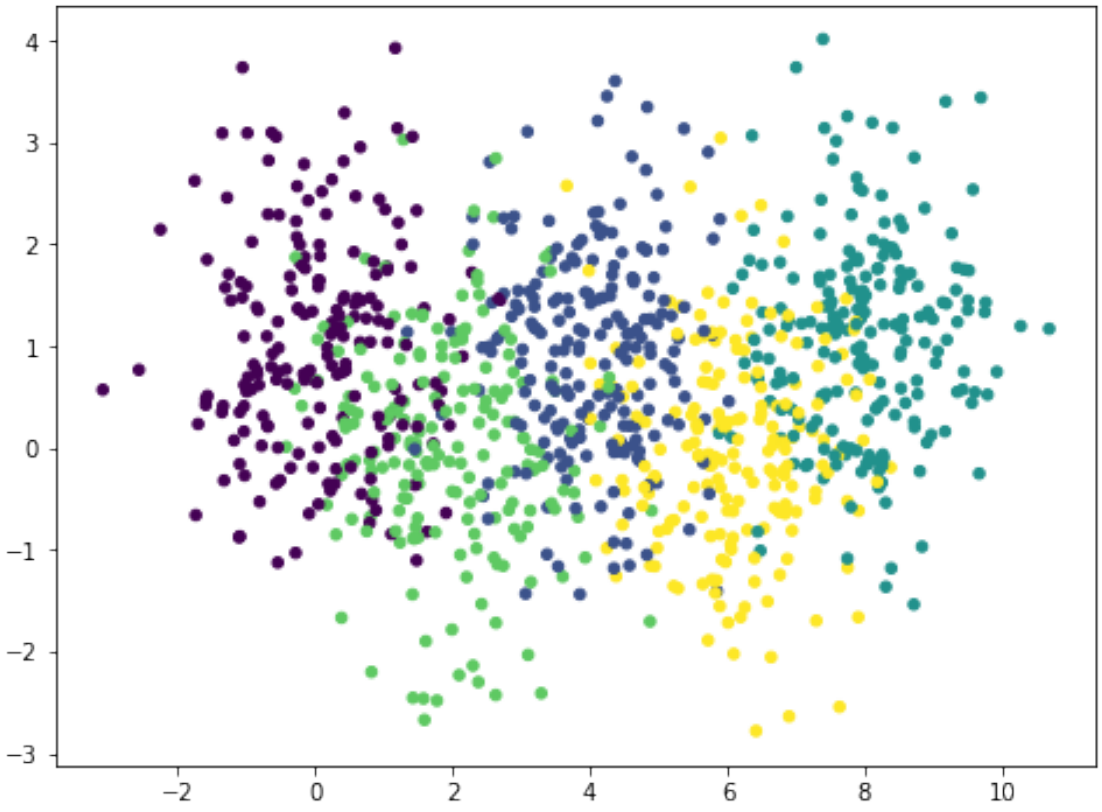
[4]:
```
X_train = X[indices[:1000],:]
X_val = X[indices[1000:1500],:]
X_test = X[indices[1500:],:]
t_train = t[indices[:1000]]
t_val = t[indices[1000:1500]]
t_test = t[indices[1500:]]
```

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

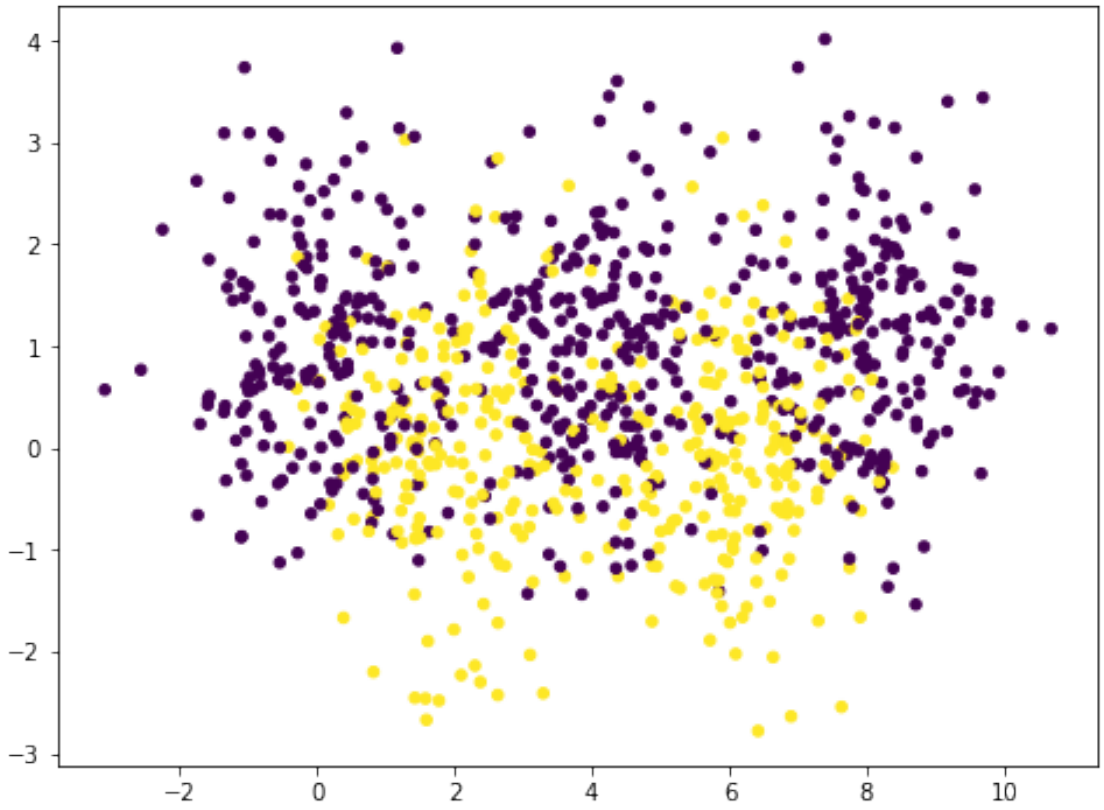
[5]:
```
t2_train = t_train >= 3 #composed of 0 and 1 since the number length is 4 This␣
 ↪is continued in the rest
t2_train = t2_train.astype('int')
t2_val = (t_val >= 3).astype('int')
t2_test = (t_test >= 3).astype('int')
# This gives us the binary set of all the t values
```

We can plot the two traing sets.

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



[7]:
```
plt.figure(figsize=(8,6))
plt.scatter(X_train[:, 0], X_train[:, 1], c=t2_train, s=20.0)
plt.show()
```



1.2 Binary classifiers

1.2.1 Linear regression

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

[8]:
```
def add_bias(X):
    # Put bias in position 0
    sh = X.shape
    if len(sh) == 1:
        #X is a vector
        return np.concatenate([np.array([1]), X]) #concatenate adds two array␣
↪to a tuple, this can be arranged with axis func
    else:
        # X is a matrix
        m = sh[0]
        bias = np.ones((m,1)) # Makes a m*1 matrix of 1-s
        return np.concatenate([bias, X], axis  = 1)
```

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

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

[101]:
```
class NumpyLinRegClass(NumpyClassifier):

    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 are the targets values for training data"""

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

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

        #Here is the loss calculation
        loss = []
        for e in range(epochs):
            loss_ = np.sum((X_train @ weights - t_train)**2)/k
            loss.append(loss_)
            weights -= eta / k *  X_train.T @ (X_train @ weights - t_train)

        return np.array(loss)

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

We can train and test a first classifier.

[102]:
```
cl = NumpyLinRegClass()
cl.fit(X_train, t2_train)
cl.accuracy(X_val, t2_val)
```

[102]: 0.49

[103]:
```
accuracy = 0
eta = 0.001
for e in [1,5,10,20,50,100,500,1000,1500,2000,3000,4000,5000,10000]:
    ls = NumpyLinRegClass()
    ls.fit(X_train, t2_train, epochs = e, eta = eta)
```

```
    best_accuracy = ls.accuracy(X_val, t2_val)
    if best_accuracy > accuracy:
        accuracy = best_accuracy
        best_eta = eta
        best_e = e
    print(best_accuracy)
    eta += 0.0005

print(f"Epochs: {e} Accuracy: {ls.accuracy(X_val, t2_val)}, Best epoch:␣
 ↪{best_e}, Best eta: {best_eta}")
```

```
0.572
0.572
0.572
0.572
0.514
0.51
0.588
0.648
0.658
0.662
0.668
0.668
0.668
0.668
Epochs: 10000 Accuracy: 0.668, Best epoch: 3000, Best eta: 0.006000000000000002
```

As seen, the result is very unimpressive. The best learning rate is 0.01, with an accuracy of 0.668 after 2000 epochs.

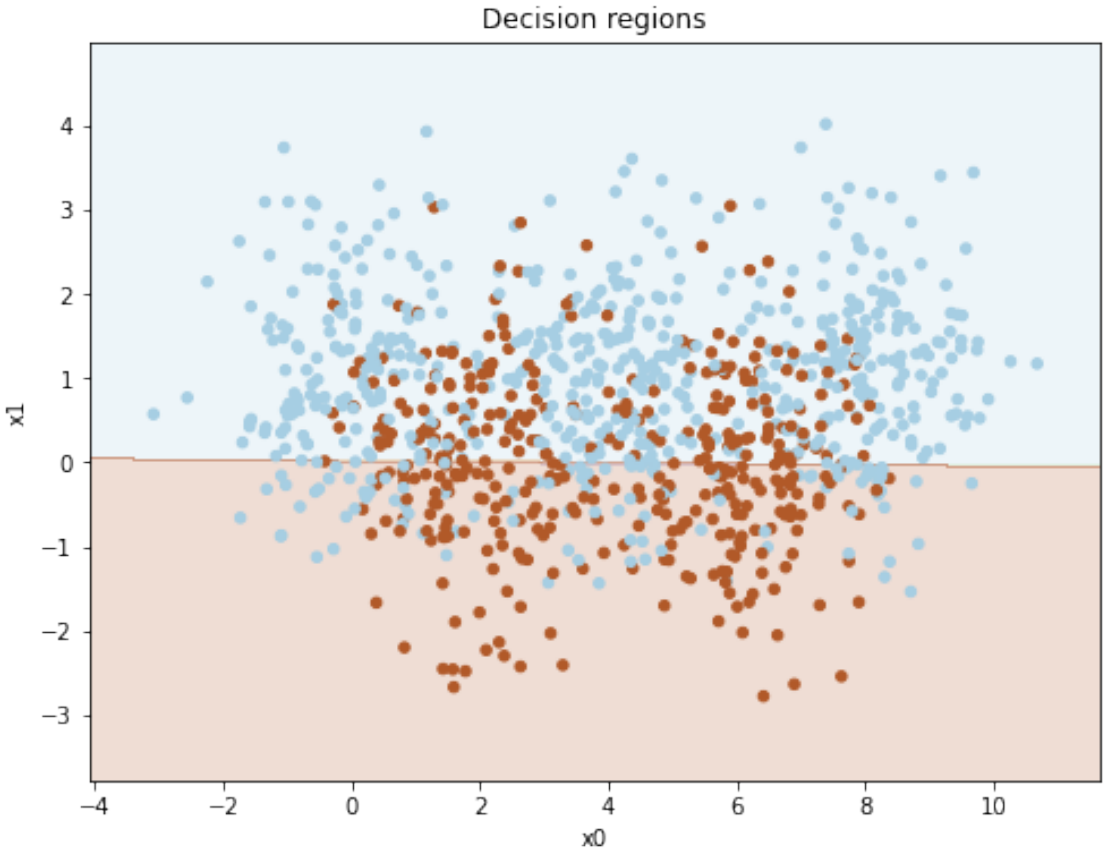
The result is far from impressive. Experiment with various settings for the hyper-parameters, eta and epochs. Report how the accuracy vary with the hyper-parameter settings. When you are satisfied with the result, you may plot the decision boundaries, as below.

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

[104]:
```
def plot_decision_regions(X, t, clf=[], size=(8,6)):
    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    h = 0.02  # step size in the mesh
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

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

    # Put the result into a color plot
```

```
    Z = Z.reshape(xx.shape)

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

    plt.scatter(X[:,0], X[:,1], c=t, s=20.0, cmap='Paired')

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

#    plt.show()
```

[105]:
```
plot_decision_regions(X_train, t2_train, ls)
```

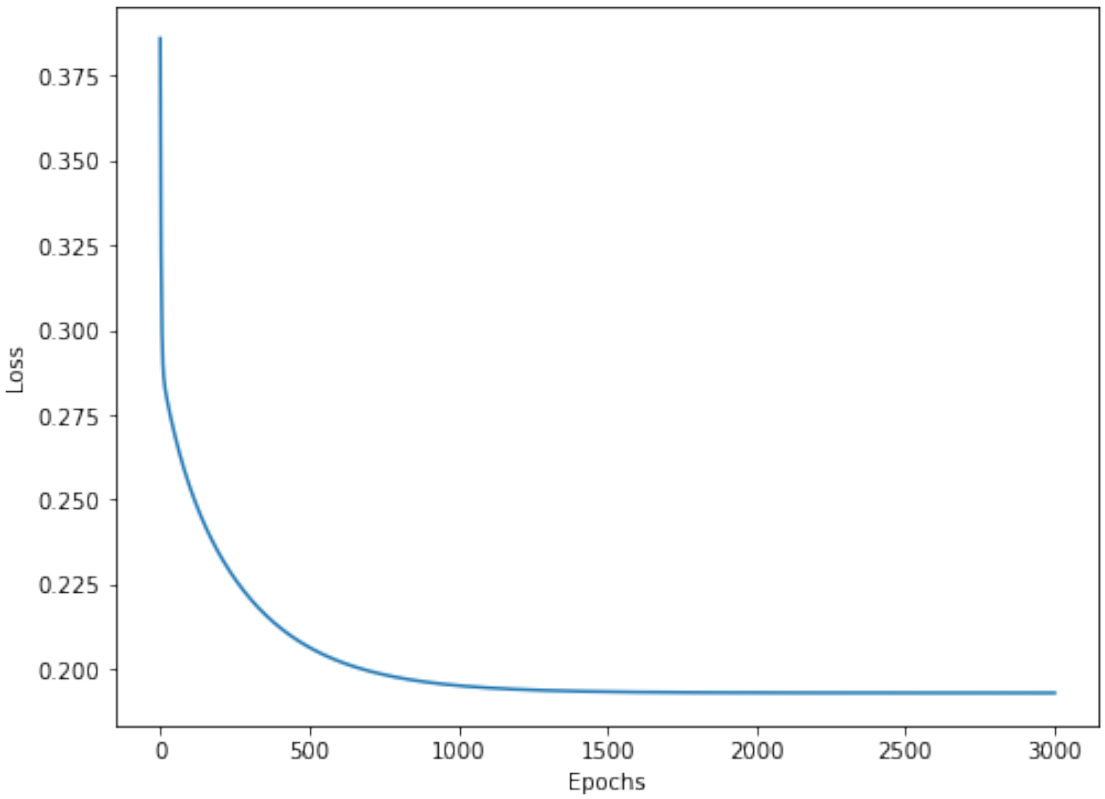


1.2.2 Loss

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

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

[106]:
```
cl = NumpyLinRegClass()
loss = cl.fit(X_train, t2_train, best_eta, best_e)
plt.figure(figsize=(8,6))
plt.plot(np.linspace(0, best_e, len(loss)), loss)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.show()
```



1.2.3 Control training

The training runs for a number of epochs. We cannot know beforehand for how many epochs it is reasonable to run the training. One possibility is to run the training until the learning does not improve much. Extend the fit-method with a keyword argument, `loss_diff`, and stop training

when the loss has not improved with more than loss_diff. Also add an attribute to the classifier which tells us after fitting how many epochs were ran.

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

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

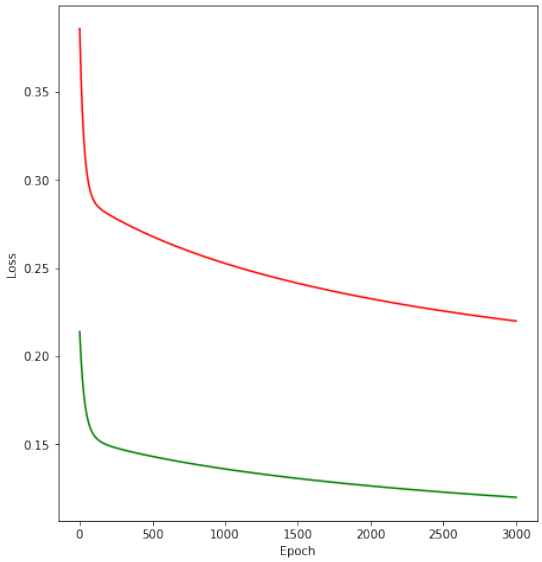
After a succesful training, plot both training loss snd vslidation loss as functions of the number of epochs in one figure, and both accuracies as functions of the number of epochs in another figure. Comment on what you see.

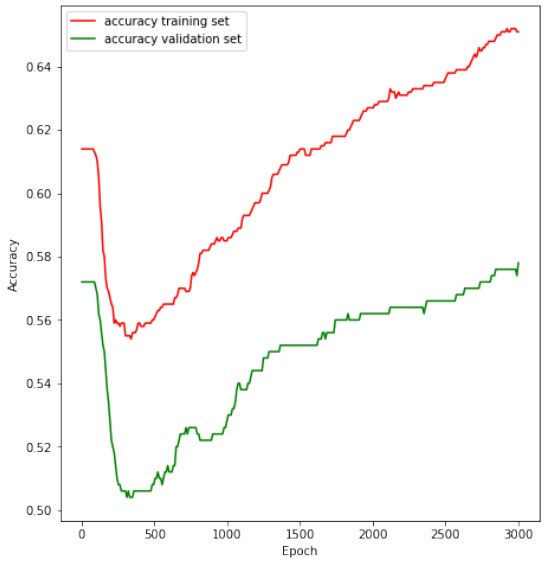
[107]:
```
class NumpyLinRegClass(NumpyClassifier):

    def fit(self, X_train, t_train, eta = 0.1, epochs=10, loss_diff = 0, X_val␣
↪= None, t_val = None):
        """X_train is a Nxm matrix, N data points, m features
        t_train are the targets values for training data"""

        (k, m) = X_train.shape

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

        if X_val.any() == None:
            X_val, t_val = X_train, t_train

        X_train = add_bias(X_train)
        X_val = add_bias(X_val)


        #Here is the loss calculation and the accuracy calculation for both␣
↪training set and validation set
        loss = []
        loss_val = []
        accuracy = []
        accuracy_val = []
        loss_ = 0
        for e in range(epochs):
            old_loss = loss_  #This is the previous loss value

            loss_ = np.sum((X_train @ weights - t_train)**2)/k
            loss_val_ = np.sum((X_val @ weights - t_val)**2)/k

            loss.append(loss_)
            loss_val.append(loss_val_)

            accuracy_ = self.accuracy(X_train[:,1:], t_train)
```

```
            accuracy_val_ = self.accuracy(X_val[:,1:], t_val)

            accuracy.append(accuracy_)
            accuracy_val.append(accuracy_val_)

            weights -= eta / k *  X_train.T @ (X_train @ weights - t_train)

            #Here is where we check if the loss difference per cycle is bigger␣
↪than loss_diff
            difference = abs(loss_ - old_loss)
            if difference < loss_diff:
                print(f"We stopped after {e} epochs, with an accuracy of␣
↪{accuracy_} in the training set and an {accuracy_val_} accuracy in the␣
↪validation set")
                break

        return np.array(loss), np.array(loss_val), np.array(accuracy), np.
↪array(accuracy_val)

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

[108]:
```
cl = NumpyLinRegClass()
loss, loss_val, accuracy, accuracy_val = cl.fit(X_train, t2_train, best_eta,␣
↪best_e, 1e-4, X_val, t2_val)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16,8))
ax1.plot(np.linspace(0, best_e, len(loss)), loss, color="red", label = "loss␣
↪training set")
ax1.plot(np.linspace(0, best_e, len(loss)), loss_val, color="green", label =␣
↪"loss validation set")
ax1.set_xlabel("Epoch")
ax1.set_ylabel("Loss")
ax2.plot(np.linspace(0, best_e, len(loss)), accuracy, color="red", label =␣
↪"accuracy training set")
ax2.set_xlabel("Epoch")
ax2.set_ylabel("Accuracy")
ax2.plot(np.linspace(0, best_e, len(loss)), accuracy_val, color="green", label␣
↪= "accuracy validation set")
plt.legend()
plt.show()
```

```
We stopped after 310 epochs, with an accuracy of 0.651 in the training set and
an 0.578 accuracy in the validation set
```





I am not completely sure on what is this huge drop in the accuracy but now i guess its a overfitting case. Overall the accuracy went up and the loss went lower

1.2.4 Logistic regression

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

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

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

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

Starting point: Code from weekly 7

[125]:
```
def logistic(x):
    return 1/(1+np.exp(-x))
```

[126]:
```
class NumpyLogReg(NumpyClassifier):

    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 are the targets values for training data"""

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

```
        X_train = add_bias(X_train)

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

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

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

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

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

[127]:
```
accuracy = 0
eta = 0.001
for e in [1,5,10,20,50,100,200,400,800,1000,1200,1500,1800,2000,2500]:
    ls = NumpyLogReg()
    ls.fit(X_train, t2_train, epochs = e, eta = eta)
    best_accuracy = ls.accuracy(X_val, t2_val)
    if best_accuracy > accuracy:
        accuracy = best_accuracy
        best_eta_log = eta
        best_e_log = e
    print(best_accuracy)
    eta += 0.0001

print(f"Epochs: {e} Accuracy: {ls.accuracy(X_val, t2_val)} Best epoch:␣
 ↪{best_e}, Best eta: {best_eta}")
```

```
0.53
0.53
0.528
0.53
0.528
0.536
0.548
0.568
0.628
0.65
0.672
```

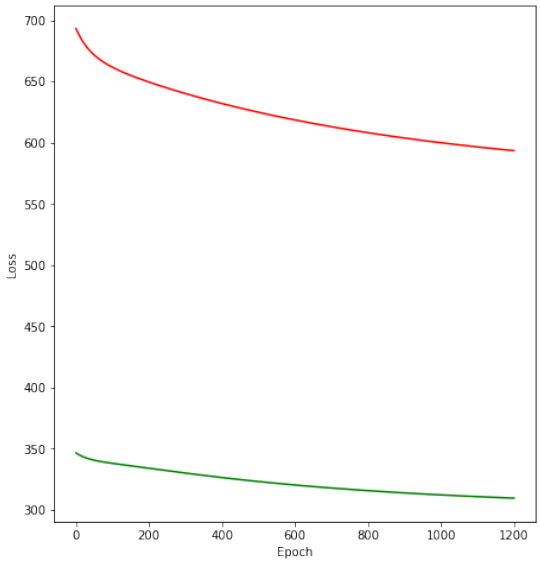
```
0.67
0.67
0.664
0.668
Epochs: 2500 Accuracy: 0.668 Best epoch: 1200, Best eta: 0.0020000000000000005
```

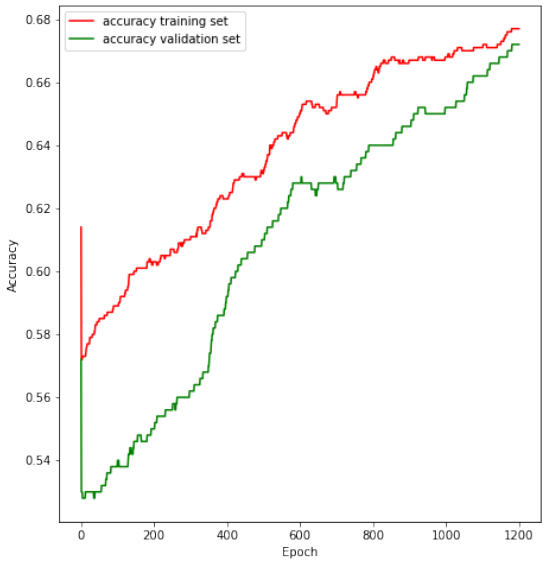
[301]:
```python
class NumpyLogReg(NumpyClassifier):

    def fit(self, X_train, t_train, eta = 0.01, epochs = 10, loss_diff = 0,␣
↪X_val = None, t_val = None):
        """X_train is a Nxm matrix, N data points, m features
        t_train are the targets values for training data"""

        if X_val is None:
            t_val = t_train
            X_val = X_train

        (k, m) = X_train.shape

        X_train = add_bias(X_train)
        X_val = add_bias(X_val)

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

        #Here is the loss calculation and the accuracy calculation for both␣
↪training set and validation set
        loss = []
        loss_val = []
        accuracy = []
        accuracy_val = []

        loss_ = 0
        for e in range(epochs):
            y = self.forward(X_train)
            y_val = self.forward(X_val)

            old_loss = loss_
            loss_ = sum(-np.log(y**t_train*(1-y)**(1-t_train))) # Calculating␣
↪the loss for each epoch
            loss_val_ = sum(-np.log(y_val**t_val*(1-y_val)**(1-t_val)))

            loss.append(loss_)
            loss_val.append(loss_val_)

            accuracy_ = self.accuracy(X_train[:,1:], t_train)
            accuracy_val_ = self.accuracy(X_val[:,1:], t_val)
```

```
            accuracy.append(accuracy_)
            accuracy_val.append(accuracy_val_)

            weights -= eta / k *  X_train.T @ (self.forward(X_train) - t_train)

            #Here is where we check if the loss difference per cycle is bigger␣
↪than loss_diff
            difference = abs(loss_ - old_loss)
            if difference < loss_diff:
                print(f"We stopped after {e} epochs, with an accuracy of␣
↪{accuracy_} in the training set and an {accuracy_val_} accuracy in the␣
↪validation set")
                break

        return loss, loss_val, accuracy, accuracy_val

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

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

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

[302]:
```
cl = NumpyLogReg()
loss, loss_val, accuracy, accuracy_val = cl.fit(X_train, t2_train, best_eta,␣
↪best_e, 1e-4, X_val, t2_val)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16,8))
ax1.plot(np.linspace(0, best_e, len(loss)), loss, color="red", label = "loss␣
↪training set")
ax1.plot(np.linspace(0, best_e, len(loss)), loss_val, color="green", label =␣
↪"loss validation set")
ax1.set_xlabel("Epoch")
ax1.set_ylabel("Loss")
ax2.plot(np.linspace(0, best_e, len(loss)), accuracy, color="red", label =␣
↪"accuracy training set")
ax2.set_xlabel("Epoch")
ax2.set_ylabel("Accuracy")
ax2.plot(np.linspace(0, best_e, len(loss)), accuracy_val, color="green", label␣
↪= "accuracy validation set")
plt.legend()
```

```
plt.show()
```





Overall the difference is quite little but it did get more accurate. Now im pretty much certain i have an overfitting case here. The loss is also much better in this case for the validation set

1.3 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).

1.3.1 "One-vs-rest" with logistic regression

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

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

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

[298]:
```
def assign_t(t, t_class):
    """Converts our t dataset to one-hot-encoded vectors"""
    for i in range(len(t)):
        if t[i] == t_class:
            t[i] = 1
        else:
```

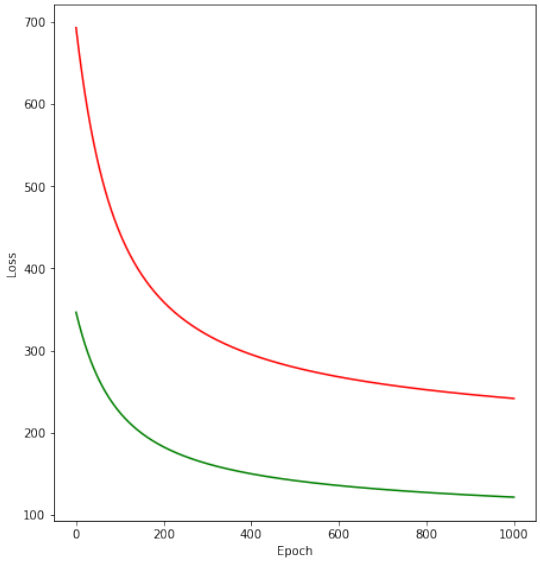
```
            t[i] = 0
    return t
```

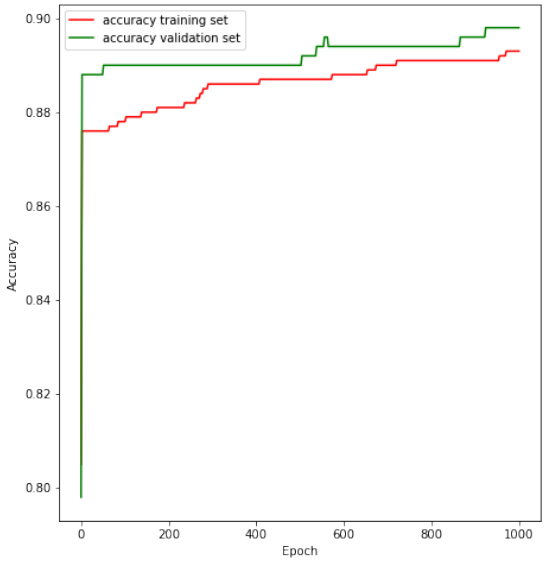
[299]:
```
def assign_items(t_train, t_val, X_train, X_val):
    """We test each classifier, to find the best fitted one for each element"""
    n = np.max(t_train) + 1
    accur_ls = []

    for i in range(n):
        cl = NumpyLogReg()
        t_train_c = assign_t(t_train.copy(), i)
        t_val_c = assign_t(t_val.copy(), i)

        eta = 0.001
        best_eta_log = 0
        best_e_log = 0
        best_e = []
        best_eta = []
        accuracy = 0
        eta = 0.001
        for e in␣
↪[1,5,10,20,50,70,100,150,200,250,300,350,400,800,1000,1200,1500,1800,2000,2500]:
↪
            cl.fit(X_train, t2_train, epochs = e, eta = eta)
            best_accuracy = cl.accuracy(X_val, t2_val)
            if best_accuracy > accuracy:
                accuracy = best_accuracy
                best_eta_log_ = eta
                best_e_log_ = e
            eta += 0.0001

        best_e.append(best_e_log_)
        best_eta.append(best_eta_log_)
        print(f'Best eta and amount of epochs for cl {i} is {best_eta_log_} and␣
↪{best_e_log_}')

        loss, loss_val, accuracy, accuracy_val = cl.fit(X_train, t_train_c,␣
↪best_eta_log_, best_e_log_, 0, X_val, t_val_c)
        accur = cl.accuracy(X_val, t_val_c)
        accur_ls.append(accur)
        print(f'For class cl {i}, the accuracy is {accur}')
        print('')
    print(f'The highest accuracy is given for cl {accur_ls.
↪index(max(accur_ls))}, with accuracy {max(accur_ls)}')
    return best_eta, best_e, accur_ls.index(max(accur_ls))
eta_best_ls, epochs_best_ls, a = assign_items(t_train, t_val, X_train, X_val)
```

```
Best eta and amount of epochs for cl 0 is 0.0024 and 1000
For class cl 0, the accuracy is 0.908

Best eta and amount of epochs for cl 1 is 0.0024 and 1000
For class cl 1, the accuracy is 0.794

Best eta and amount of epochs for cl 2 is 0.0024 and 1000
For class cl 2, the accuracy is 0.808

Best eta and amount of epochs for cl 3 is 0.0024 and 1000
For class cl 3, the accuracy is 0.754

Best eta and amount of epochs for cl 4 is 0.0024 and 1000
For class cl 4, the accuracy is 0.766

The highest accuracy is given for cl 0, with accuracy 0.908
```

[303]:
```
cl = NumpyLogReg()
t_train_c = assign_t(t_train.copy(), a)
t_val_c = assign_t(t_val.copy(), a)

loss, loss_val, accuracy, accuracy_val = cl.fit(X_train, t_train_c,␣
 ↪eta_best_ls[a], epochs_best_ls[a], 1e-1, X_val, t_val_c)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16,8))
ax1.plot(np.linspace(0, epochs_best_ls[a], len(loss)), loss, color="red", label␣
 ↪= "loss training set")
ax1.plot(np.linspace(0, epochs_best_ls[a], len(loss)), loss_val, color="green",␣
 ↪label = "loss validation set")
ax1.set_xlabel("Epoch")
ax1.set_ylabel("Loss")
ax2.plot(np.linspace(0, epochs_best_ls[a], len(loss)), accuracy, color="red",␣
 ↪label = "accuracy training set")
ax2.set_xlabel("Epoch")
ax2.set_ylabel("Accuracy")
ax2.plot(np.linspace(0, epochs_best_ls[a], len(loss)), accuracy_val,␣
 ↪color="green", label = "accuracy validation set")
plt.legend()
plt.show()
```

```
We stopped after 448 epochs, with an accuracy of 0.893 in the training set and
an 0.898 accuracy in the validation set
```





Here our accuracy spiked by quite a bit as we have reached about 90% accuracy and there is a noticable drop in the loss compared to our last tests.

2 Part II

2.1 Multi-layer neural networks

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

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

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

2.2 Round 1: One epoch of training

2.2.1 Scaling

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

[304]:
```
def normalization(X):
    return (X - np.min(X))/(np.max(X)-np.min(X))
```

```
Xn_train = normalization(X_train)
Xn_val = normalization(X_val)
```

2.2.2 Initialization

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

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

[196]:
```
eta = 0.0024 #Learning rate
dim_hidden = 3
```

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

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

[305]:
```
# convert t_train
def convert(t_train):
    n = np.max(t_train) + 1
    t_norm = []
    for i in range(n):
        t_norm.append(assign_t(t_train.copy(), i))
    t_norm = np.array(t_norm)
    return t_norm

t_norm = convert(t_train)
dim_in =  0  # Calculate the correct value from the input data
dim_in = Xn_train.shape[-1]
dim_out = 0  # Calculate the correct value from the input data
dim_out = np.shape(t_norm)
dim_out = dim_out[0]
print(dim_out)
```

```
5
```

We need two sets of weights: weights1 between the input and the hidden layer, and weights2, between the hidden layer and the output. Make sure that you take the bias terms into consideration and get the correct dimensions. The weight matrices should be initialized to small random numbers, not to zeros. It is important that they are initialized randomly, both to ensure that different neurons start with different initial values and to generate different results when you rerun the classifier. In

this introductory part, we have chosen to fix the random state to make it easier for you to control your calculations. But this should not be part of your final classifier.

[265]:
```
# Your code

rng = np.random.RandomState(2022)
W = (rng.rand(dim_in + 1, dim_hidden) * 2 - 1)/np.sqrt(dim_in) # Weights from␣
↪input to hidden
V = (rng.rand(dim_hidden + 1, dim_out) * 2 - 1)/np.sqrt(dim_hidden) # Weights␣
↪from hidden to output

print(W.shape)
print(W)
```

```
(3, 3)
[[-0.6938717  -0.00133246 -0.54675803]
 [-0.63643285  0.26220593 -0.01840165]
 [ 0.56237224  0.20852872  0.56139063]]
```

[219]:
```
rng = np.random.RandomState(2022)
weights1 = (rng.rand(dim_in + 1, dim_hidden) * 2 - 1)/np.sqrt(dim_in)
weights2 = (rng.rand(dim_hidden+1, dim_out) * 2 - 1)/np.sqrt(dim_hidden)
```

[220]:
```
weights1
```

[220]:
```
array([[-0.6938717 , -0.00133246, -0.54675803],
       [-0.63643285,  0.26220593, -0.01840165],
       [ 0.56237224,  0.20852872,  0.56139063]])
```

2.2.3 Forwards phase

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

[259]:
```
def generator(X_train, weights):
    X_train = add_bias(X_train) # We use the add_bias function to add on bias␣
↪in the first layer
    return X_train @ weights
```

[270]:
```
Wn_train = generator(Xn_train, W)
output = generator(logistic(Wn_train), V)
print(output)
```

```
[[-0.26320822  0.54537425  0.35995275  0.24810443  0.7166017 ]
 [-0.27381372  0.55196346  0.36341499  0.23805342  0.70474144]
```

```
[-0.25241309  0.54034835  0.35680049  0.25540545  0.72382392]
…
[-0.2751909   0.54350703  0.36086036  0.25345103  0.73198161]
[-0.26892766  0.54490251  0.3606034   0.2498627   0.72247966]
[-0.2496302   0.54332273  0.35732685  0.24964798  0.7125669 ]]
```

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

2.2.4 Backwards phase

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

[278]:
```
k,m = np.shape(output)

#Here is my overflow problem i have tweaked it a bit but the problem seems to␣
 ↪persist
delta0 = np.subtract(output, t_norm.T) * output * np.subtract(1, output)
```

Calculate the delta terms in the hidden layer.

[280]:
```
Bob = add_bias(logistic(Wn_train))

#Here is my overflow problem i have tweaked it a bit but the problem seems to␣
 ↪persist
delta1 = Bob * np.subtract(1, Bob) * (delta0 @ V.T)
```

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

[282]:
```
(N, m) = delta1.shape
delta1_n = np.zeros((N, m - 1))
for i in range(m-1):
    delta1_n[:,i] = delta1[:,i+1]

Xn_train_2 = add_bias(Xn_train) # Add bias to normalized X_train

V = np.subtract(V, (eta/k * np.dot(Bob.T, delta0))) # updating weights between␣
 ↪hidden and output

W = np.subtract(W, (eta/k * np.dot(Xn_train_2.T, delta1_n))) # updating weights␣
 ↪between input and hidden

print('New weights from input to hidden:')
print(W)
```

```
New weights from input to hidden:
[[-0.69388561 -0.00131466 -0.54669659]
 [-0.63644139  0.26220961 -0.01837323]
 [ 0.56236799  0.20853409  0.56140886]]
```

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

[29]:
```
print("New weights:")
print(weights1)
```

```
New weights:
[[-0.6938717  -0.00133246 -0.54675803]
 [-0.63643285  0.26220593 -0.01840165]
 [ 0.56237224  0.20852872  0.56139063]]
```

2.3 Step 2: A Multi-layer neural network classifier

2.3.1 Make the classifier

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

[334]:
```
class MNNClassifier(NumpyClassifier):
    """A multi-layer neural network with one hidden layer"""

    def __init__(self, eta = 0.001, dim_hidden = 6):
        """Initialize the hyperparameters"""
        self.eta = eta
        self.dim_hidden = dim_hidden

        # Should you put additional code here?

    def fit(self, X_train, t_train, X_val = None, t_val = None, epochs = 100):
        """
        The code here is a bit wonky because of the delta calculations, i lose␣
↪a fair bit of
        informations since im hiting a stack overflow, besides that the code␣
↪runs fine.
        """
```

```
        # Initilaization
        if X_val is None:
            self.X_val = X_train
            self.t_val = t_train

        else:
            self.t_val = t_val
            self.X_val = X_val

        self.X_train = X_train
        self.t_train = t_train

        dim_in = np.shape(self.X_train) # Calculate the correct value from the␣
↪input data
        dim_in = dim_in[1]
        dim_out = np.shape(self.t_train) # Calculate the correct value from the␣
↪output data
        dim_out = dim_out[0]

        self.W = (np.random.rand(dim_in + 1, dim_hidden) * 2 - 1)/np.
↪sqrt(dim_in) # Weights from input to hidden
        self.V = (np.random.rand(dim_hidden + 1, dim_out) * 2 - 1)/np.
↪sqrt(dim_hidden) # Weights from hidden to output

        for e in range(epochs):
            self.A, self.out = self.forward(self.X_train)


            #Here is where we stumble to the stack overflow
            delta0 = np.subtract(self.out, self.t_train.T) * self.out * np.
↪subtract(1, self.out)
            B = add_bias(self.A)
            delta1 = B * np.subtract(1, B) * (delta0 @ self.V.T)

            (n, m) = delta1.shape
            delta1_n = np.zeros((n, m - 1))
            for i in range(m-1):
                delta1_n[:,i] = delta1[:,i+1]

            self.X_train_b = add_bias(self.X_train) # Add bias to normalized␣
↪X_train

            self.V = np.subtract(self.V, (self.eta/n * np.dot(B.T, delta0))) #␣
↪updating weights between hidden and output
            self.W = np.subtract(self.W, (self.eta/n * np.dot(self.X_train_b.T,␣
↪delta1_n)))
```

```
    def forward(self, X):
        """
        This is just a copy of the code line when we started this task
        """
        z = generator(X, self.W)
        A = logistic(z) # We use logistic from earlier, the sigmond activation␣
↪function
        out = generator(A, self.V)
        return A, out

    def predict(self, x):
        score = self.forward(x)[1]
        return score.argmax(axis=1)
```

2.3.2 Multi-class

Train the network on (X_train, t_train) (after scaling), and test on (X_val, t_val). Tune the hyperparameters to get the best result: - number of epochs - learning rate - number of hidden nodes.

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

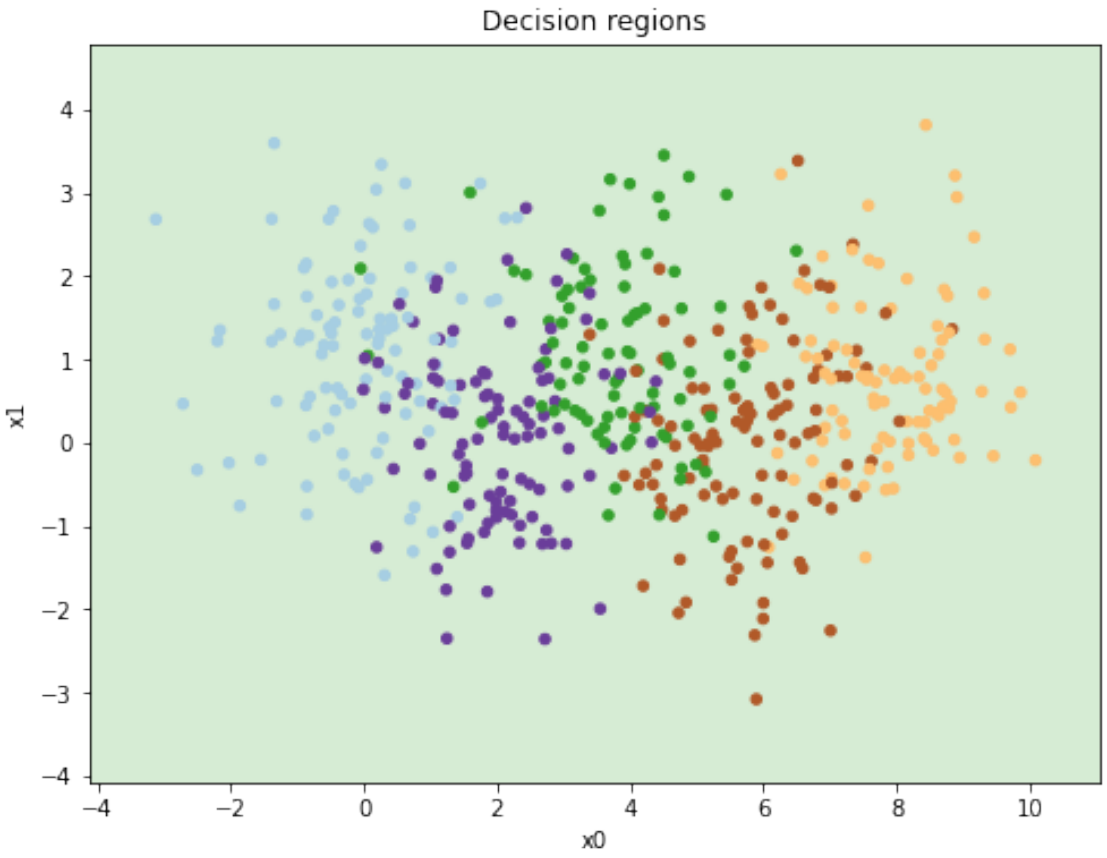
Discuss shortly how the results and decsion boundaries compare to the "one-vs-rest" classifier.

[335]:
```
eta = 0.001
dim_hidden = 3
acc = 0

epochs_best = 0
dim_hidden_best = 0
eta_best = 0

t_train_c = convert(t_train)
t_val_c = convert(t_val)

for i in range(10):
    epochs = 100
    for j in range(5):
        cl = MNNClassifier(eta, dim_hidden)
        cl.fit(Xn_train, t_train_c, Xn_val, t_val_c, epochs)
        ac = cl.accuracy(X_val, t_val)
        epochs += 200
        if ac > acc:
```

```
            acc = ac
            eta_best = eta
            dim_hidden_best = dim_hidden
            epochs_best = epochs
    eta += 0.003
    dim_hidden += 2

print(f'for non-binary classes: The optimal accuracy with weights from the␣
 ↪above runthroughs = {acc}, is aquired with eta = {eta_best}, dim_hidden =␣
 ↪{dim_hidden_best}, epochs = {epochs_best}')

cls = []
acc_ls = []
for i in range(10):
    cl = MNNClassifier(eta_best, dim_hidden_best)
    cls.append(cl)
    cl.fit(Xn_train, t_train_c, Xn_val, t_val_c, epochs_best)
    acc_ls.append(cl.accuracy(X_val, t_val))
acc_ls = np.array(acc_ls)

print(f'Mean accuracy of 10 runs is {acc_ls.mean()} and the standard deviation␣
 ↪is {np.std(acc_ls)}')

i, = np.where(np.isclose(acc_ls, max(acc_ls)))
plot_decision_regions(X_val, t_val, cls[i[0]])
```

```
/tmp/ipykernel_575/2665622831.py:42: RuntimeWarning: overflow encountered in
matmul
  delta1 = B * np.subtract(1, B) * (delta0 @ self.V.T)
/tmp/ipykernel_575/2665622831.py:42: RuntimeWarning: invalid value encountered
in multiply
  delta1 = B * np.subtract(1, B) * (delta0 @ self.V.T)
/tmp/ipykernel_575/2665622831.py:40: RuntimeWarning: overflow encountered in
multiply
  delta0 = np.subtract(self.out, self.t_train.T) * self.out * np.subtract(1,
self.out)
```

```
for non-binary classes: The optimal accuracy with weights from the above
runthroughs = 0.36, is aquired with eta = 0.001, dim_hidden = 3, epochs = 700
Mean accuracy of 10 runs is 0.202 and the standard deviation is 0.0
```



This as well as the code below though to a lesser degree seem to have a stack overflow case with both deltas, asides from that the generation works fine but the accuracy is too low for me be sure of that.

2.3.3 Binary class

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

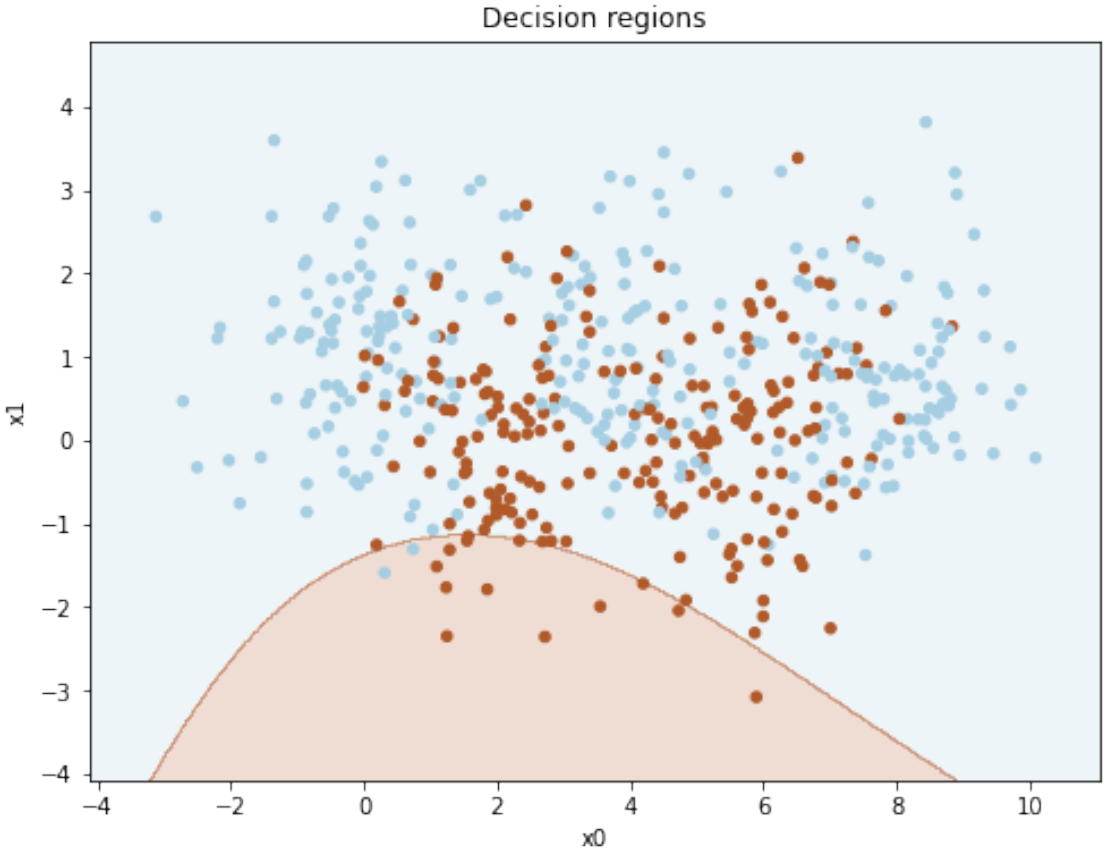
[338]:
```
eta = 0.01
dim_hidden = 3
acc = 0

epochs_best = 0
dim_hidden_best = 0
eta_best = 0

t2_train_c = convert(t2_train)
```

```
t2_val_c = convert(t2_val)

for i in range(10):
    epochs = 100
    for j in range(10):
        cl = MNNClassifier(eta, dim_hidden)
        cl.fit(Xn_train, t2_train_c, Xn_val, t2_val_c, epochs)
        ac = cl.accuracy(X_val, t2_val)
        epochs += 100
        if ac > acc:
            acc = ac
            eta_best = eta
            dim_hidden_best = dim_hidden
            epochs_best = epochs
    eta += 0.003
    dim_hidden += 2

print(f'for binary classes: The optimal accuracy = {acc}, is aquired with eta =␣
 ↪{eta_best}, dim_hidden = {dim_hidden_best}, epochs = {epochs_best}')

cls = []
acc_ls = []
for i in range(10):
    cl = MNNClassifier(eta_best, dim_hidden_best)
    cls.append(cl)
    cl.fit(Xn_train, t2_train_c, Xn_val, t2_val_c, epochs_best)
    acc_ls.append(cl.accuracy(X_val, t2_val))
acc_ls = np.array(acc_ls)

print(f'Mean accuracy of 10 runs is {acc_ls.mean()} and the standard deviation␣
 ↪is {np.std(acc_ls)}')

i, = np.where(np.isclose(acc_ls, max(acc_ls)))
plot_decision_regions(X_val, t2_val, cls[i[0]])
```

```
/tmp/ipykernel_575/2665622831.py:40: RuntimeWarning: overflow encountered in
multiply
  delta0 = np.subtract(self.out, self.t_train.T) * self.out * np.subtract(1,
self.out)
/tmp/ipykernel_575/2665622831.py:42: RuntimeWarning: invalid value encountered
in multiply
  delta1 = B * np.subtract(1, B) * (delta0 @ self.V.T)
/tmp/ipykernel_575/2665622831.py:42: RuntimeWarning: overflow encountered in
matmul
  delta1 = B * np.subtract(1, B) * (delta0 @ self.V.T)
```

```
for binary classes: The optimal accuracy = 0.638, is aquired with eta = 0.037,
dim_hidden = 21, epochs = 1100
```

```
Mean accuracy of 10 runs is 0.5741999999999999 and the standard deviation is
0.006600000000000006
```



3 Part III: Final testing

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

3.1 Binary task (X, t2)

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

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

Also report precision and recall for class 1.

[340]:
```
#NumpyLinRegClass
cl = NumpyLinRegClass()
loss, loss_val, accuracy, accuracy_val = cl.fit(X_train, t2_train, 0.003, 2500,␣
 ↪1e-4, X_test, t2_test)
epochs = np.linspace(0, 700, len(loss))
print('')
print('NumpyLinRegClass test set accuracy:', max(accuracy_val))
print('NumpyLinRegClass training set accuracy:', max(accuracy))

cl = NumpyLinRegClass()
loss, loss_val, accuracy, accuracy_val = cl.fit(X_train, t2_train, 0.003, 2500,␣
 ↪1e-4, X_val, t2_val)
epochs = np.linspace(0, 700, len(loss))
print('')
print('NumpyLinRegClass validation set accuracy:', max(accuracy_val))
print('')

# NumpyLogReg
cl = NumpyLogReg()
loss, loss_val, accuracy, accuracy_val = cl.fit(X_train, t2_train,␣
 ↪best_eta_log, best_e_log, 1e-1, X_test, t2_test)
epoch = np.linspace(0, best_e_log, len(loss))
print('')
print('NumpyLogReg test set accuracy:', max(accuracy_val))
print('NumpyLogReg training set accuracy:', max(accuracy))

cl = NumpyLogReg()
loss, loss_val, accuracy, accuracy_val = cl.fit(X_train, t2_train,␣
 ↪best_eta_log, best_e_log, 1e-1, X_val, t2_val)
epoch = np.linspace(0, best_e_log, len(loss))
print('')
print('NumpyLogReg validation set accuracy:', max(accuracy_val))
print('')
```

```
We stopped after 291 epochs, with an accuracy of 0.609 in the training set and
an 0.578 accuracy in the validation set

NumpyLinRegClass test set accuracy: 0.6
NumpyLinRegClass training set accuracy: 0.614
We stopped after 291 epochs, with an accuracy of 0.609 in the training set and
an 0.552 accuracy in the validation set

NumpyLinRegClass validation set accuracy: 0.572

We stopped after 213 epochs, with an accuracy of 0.603 in the training set and
an 0.592 accuracy in the validation set
```

```
NumpyLogReg test set accuracy: 0.6
NumpyLogReg training set accuracy: 0.614
We stopped after 213 epochs, with an accuracy of 0.603 in the training set and
an 0.554 accuracy in the validation set

NumpyLogReg validation set accuracy: 0.572
```

Overall the accuracy between the two test is fairly similar and have minimal differences as a whole

3.2 Multi-class task (X, t)

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

Comment on the results.

[315]:
```
"""Just reruning the onevsrest code the answers stay almost the same as␣
↪before"""
best_eta, best_e, a = assign_items(t_train, t_val, X_train, X_val) # training␣
↪and validation
```

```
Best eta and amount of epochs for cl 0 is 0.0024 and 1000
For class cl 0, the accuracy is 0.908

Best eta and amount of epochs for cl 1 is 0.0024 and 1000
For class cl 1, the accuracy is 0.794

Best eta and amount of epochs for cl 2 is 0.0024 and 1000
For class cl 2, the accuracy is 0.808

Best eta and amount of epochs for cl 3 is 0.0024 and 1000
For class cl 3, the accuracy is 0.754

Best eta and amount of epochs for cl 4 is 0.0024 and 1000
For class cl 4, the accuracy is 0.766

The highest accuracy is given for cl 0, with accuracy 0.908
```

[341]:
```
"""Same as above just reruning the code again"""
cl = NumpyLogReg()
t_train_c = assign_t(t_train.copy(), a)
t_val_c = assign_t(t_val.copy(), a)
loss, loss_val, accuracy, accuracy_val = cl.fit(X_train, t_train_c,␣
↪eta_best_ls[a], epochs_best_ls[a], 1e-1, X_val, t_val_c)
print('')
```

```
print(f'Accurcy of training: {np.max(accuracy)} Accuracy of validation: {np.
 ↪max(accuracy_val)}')
```

```
We stopped after 448 epochs, with an accuracy of 0.893 in the training set and
an 0.898 accuracy in the validation set
```

```
Accurcy of training: 0.893 Accuracy of validation: 0.898
```

The diference between the OneVSRest and MNN is quite obvious. With the OneVSRest i am about 90% accuray whilst with the MNN test im about 60% accurate. This is a mixture of the stack overflow case that we have but it could also mean that OneVSRest method is far more surperior than the MNN method.