IN4050 Mandatory Assignment 2, 2024: Supervised Learning

Rules

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

Delivery

Deadline: 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 [540…
```
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 [541…
```
# 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],[8,1],[2,0],[6,0]],
                  n_features=2, random_state=2024, cluster_std=[1.0, 2.0, 1.0, 0.5, 0.5])
```

In [542…
```
# Shuffling the dataset
indices = np.arange(X.shape[0])
rng = np.random.RandomState(2024)
rng.shuffle(indices)
indices[:10]
```

Out[542…
```
array([ 937, 1776,  868, 1282, 1396,  147,  601, 1193, 1789,  547])
```

In [543…
```
# 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:

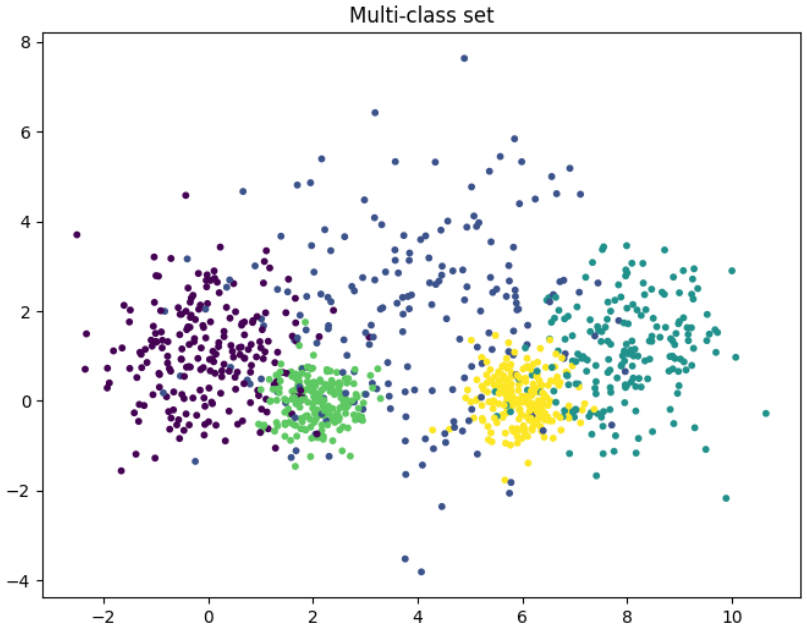
- Binary set: `(X, t2)`
- Multi-class set: `(X, t_multi)`

In [544…
```
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.

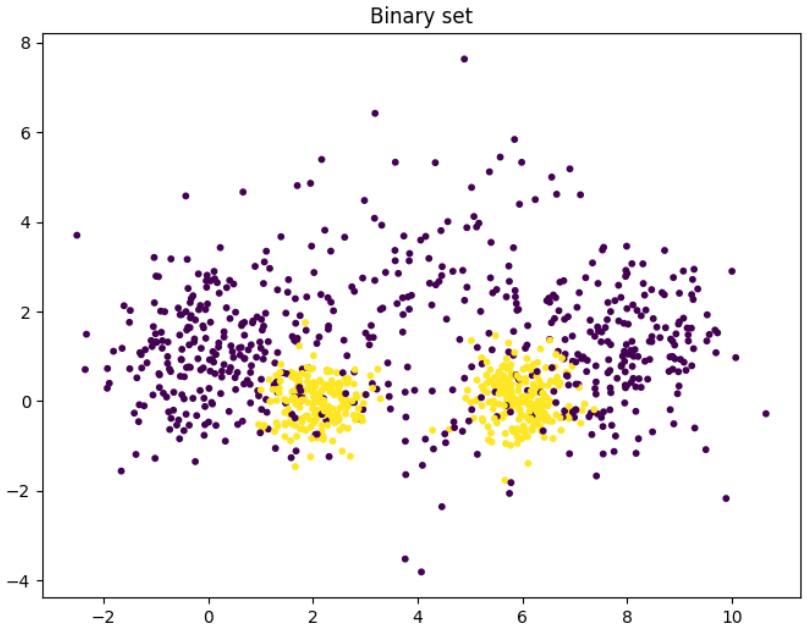
In [545…
```
plt.figure(figsize=(8,6)) # You may adjust the size
plt.scatter(X_train[:, 0], X_train[:, 1], c=t_multi_train, s=10.0)
plt.title("Multi-class set")
```

Out[545…
```
Text(0.5, 1.0, 'Multi-class set')
```



In [546…
```
plt.figure(figsize=(8,6))
plt.scatter(X_train[:, 0], X_train[:, 1], c=t2_train, s=10.0)
plt.title("Binary set")
```

Out[546…
```
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 [547…
```python
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 [548…
```python
class NumpyClassifier():
    """Common methods to all Numpy classifiers --- if any"""
```

In [549…
```python
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 a vector 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 _ 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

    def precision(self, y, t):
        """Calculate the precision of the predictions.
        Method implemented for part 3."""

        # True Positive and False Positive
        TP = np.sum((y == 1) & (t == 1))
        FP = np.sum((y == 1) & (t == 0))

        # Avoid division by zero
        if (TP + FP) == 0:
            return 0.0

        # Calculate and return precision
        return (TP / (TP + FP))

    def recall(self, y, t):
        """Calculate the recall of the predictions.
        Method implemented for part 3."""

        # True Positive (TP) and False Negative (FN) counts
        TP = np.sum((y == 1) & (t == 1))
```

```
        FN = np.sum((y == 0) & (t == 1))

        # Avoid division by zero
        if (TP + FN) == 0:
            return 0.0

        # Calculate recall
        recall = TP / (TP + FN)
        return recall
```

We can train and test a first classifier.

In [550…
```
def accuracy(predicted, gold):
    return np.mean(predicted == gold)
```

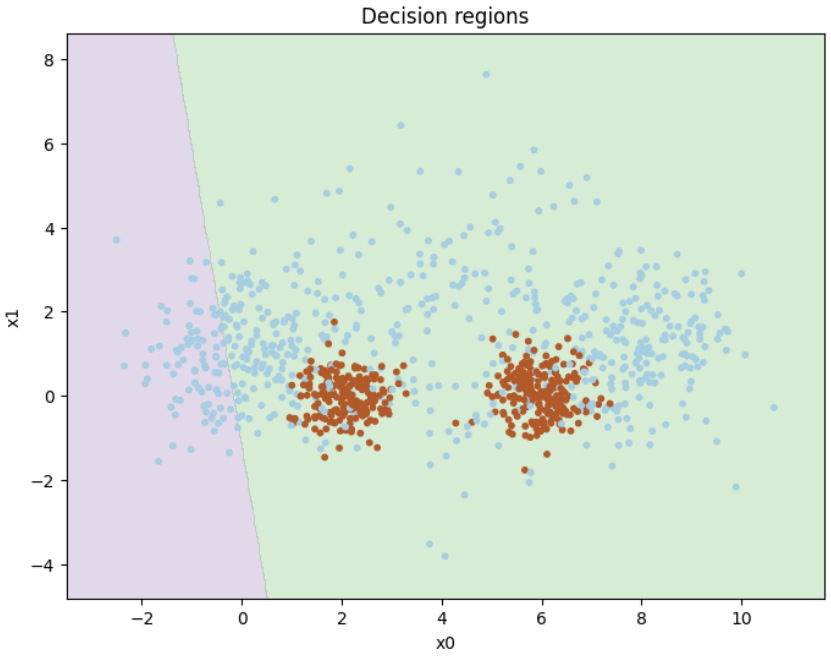
In [551…
```
cl = NumpyLinRegClass()
cl.fit(X_train, t2_train, lr=0.1, epochs=100)
print("Accuracy on the validation set:", accuracy(cl.predict(X_val), t2_val))
print("Recall:", cl.recall(cl.predict(X_val), t2_val))
print("Precision:", cl.precision(cl.predict(X_val), t2_val))
```

```
Accuracy on the validation set: 0.534
Recall: 0.0
Precision: 0.0
```

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 [552…
```
def plot_decision_regions(X, t, clf=[], size=(8,6)):
    """Plot the data set (X,t) together with the decision boundary of the classifier clf"""
    # 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 mesh point.
    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 [553…
```
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 out.

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

Solution

Here is the complete list of all tested hyper-parameters with the given accuracy:

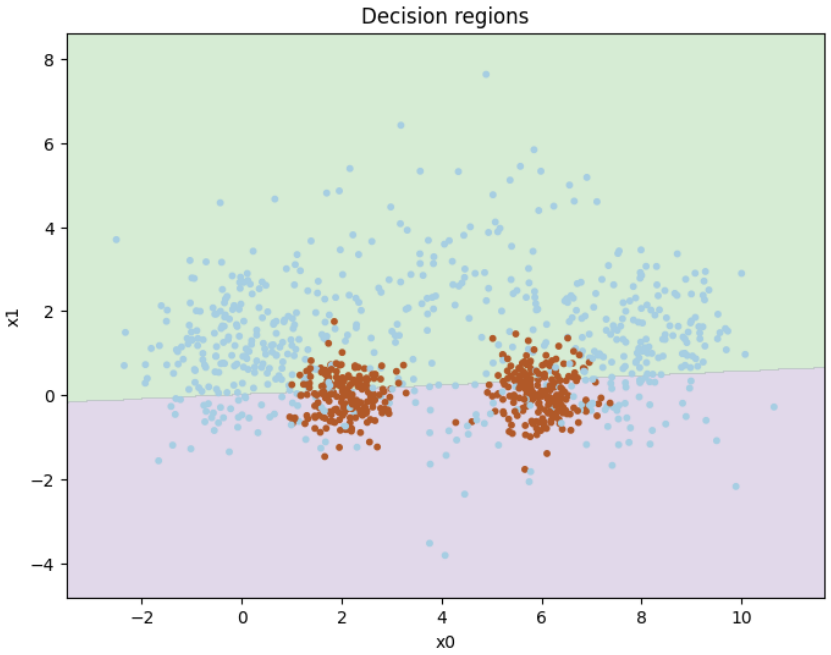
Learning Rate	Epochs	Accuracy
0.1	10	54.2%
0.1	100	53.4%
0.1	1000	53.4%
0.01	10	47%
0.01	100	56.6%
0.01	1000	75%
0.03	1000	76%
0.02	2000	76%
0.002	2000	67.4%
0.002	10000	76%

I started by trying the same learning rate with different epochs. Increasing the amount of epochs did not improve the overall accuracy. This makes sense, because more iterations with same learning rate will just go over the global optima with the same step. Also, with very few epochs we do underfitting by not allowing the model to adjust enough.

Decreasing the learning rate allows us to approach the local optima. It requires more epochs, but the performance are improved. The best accuracy was when I had was **76%**

In [554…
```
# Creating the model with parameters
hyper_parameter_model = NumpyLinRegClass()
hyper_parameter_model.fit(X_train, t2_train, lr=0.03, epochs=1000)
hyper_parameter_model_accuracy = accuracy(hyper_parameter_model.predict(X_val), t2_val)
print(f"Accuracy on the validation set: {round(hyper_parameter_model_accuracy*100,2)}%")

# Plotting the decision boundary
plot_decision_regions(X_train, t2_train, hyper_parameter_model)
```

```
Accuracy on the validation set: 76.0%
```



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.

Solution

For scaling the data, I implemented normalization which uses the minimum and maximum values as scale each data based on the upper and lower bound. It worked really well. The following was tested when trying to find a new hyper parameters:

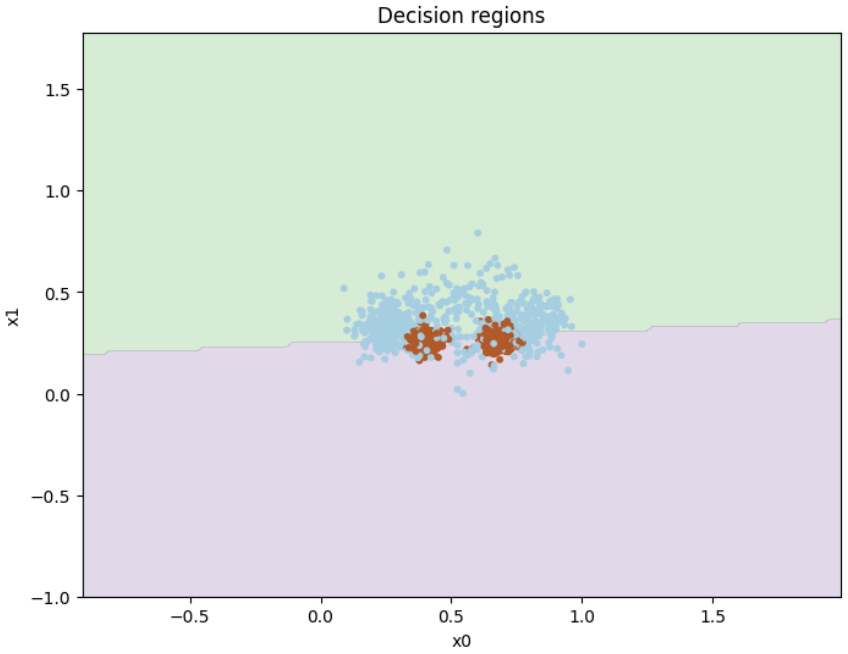
Learning Rate	Epochs	Accuracy
0.02	1000	64%
0.03	1000	65.4%
0.08	1000	73%
0.2	1000	76%
0.4	1000	77.2%
0.44	1000	77.4%

The best solution was **77.4%**. The overall performance got improved by scaling the data. It was able to perform better. The hyper parameter for the last solution was worse than the original hyper parameters, but the new one improved better.

In [555…
```
# Scaler function for normalizing the data
def scaler(data):
    data = (data - data.min()) / (data.max() - data.min())
    return data
```

```
# Scaling the training data
X_scaled = scaler(X_train)
t_scaled = scaler(t2_train)

# Create the model with scaled data
scaled_model = NumpyLinRegClass()
scaled_model.fit(X_scaled, t_scaled, lr=0.44, epochs=1000)
scaled_model_accuracy = accuracy(scaled_model.predict(X_val), t2_val)
print(f"Accuracy on the validation set: {round(scaled_model_accuracy*100,2)}%")

# Plotting the decision boundary
plot_decision_regions(X_scaled, t_scaled, scaled_model)
```

```
Accuracy on the validation set: 77.4%
```



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 [588…

```
# Code for logistic function
# It is also known as a sigmoid function
def logistic(x):
    return 1/(1+np.exp(-x))


# Code for the logistic regression classifier
class LogisticClassifier:
    def __init__(self, bias=-1):
        self.bias=bias
        self.epochs_trained = 0


    def fit(self, X_train, t_train, X_val, t_val, eta = 0.1, epochs=10, tol=1, n_epochs_no_update=5, logging=False):
        """X_train is a Nxm matrix, N data points, m features
        t_train are the targets values for training data"""

        # For task c)
        # Store loss and accuracies
        training_loss = []
        validation_loss = []
        accuracies = []

        # Add bias to training data
        (k, m) = X_train.shape
        X_train = add_bias(X_train, self.bias)

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

```
        # Attribute for keeping track of epochs trained
        self.epochs_trained = 0

        # Variable to keep track of how many epochs trained without an update in weights
        no_update_counter = 0

        # Train over the given epochs
        for e in range(epochs):

            # Increment epochs count
            self.epochs_trained += 1

            loss_before_update = self.cross_entropy_loss(x=X_train, t=t_train, add_train=False)

            # Change the weights with update rule of gradient decent
            weights -= eta / k *  X_train.T @ (self.forward(X_train) - t_train)

            # Must check that both are not none
            # If only one of the validation set is set something, then we will get an error
            assert X_val is not None and t_val is not None, "Both train and target validation set must be set"

            # Use validation set instead
            # Calculate the loss with cross entropy on the
            current_val_loss = self.cross_entropy_loss(x=X_val, t=t_val, add_train=True)
            current_train_loss = self.cross_entropy_loss(x=X_train, t=t_train, add_train=False)
            validation_loss.append(current_val_loss)
            training_loss.append(current_train_loss)

            # Calculate the accuracy over time
            current_accuracy = self.accuracy(X_val, t_val)
            accuracies.append(current_accuracy)


            # Check if the changes are less than the tolerance set
            # We take the mean of the absolute value of the difference, and check if it is less than the tolerance set
            if(np.mean(np.abs(loss_before_update - current_train_loss)) < tol):
                no_update_counter += 1
            else:
                # No reason to change to counter
                no_update_counter = 0

            if (self.epochs_trained) % 1000 == 0 and logging:
                print(f"EPOCH {self.epochs_trained}:        {current_train_loss} loss, {round(current_accuracy*100, 2)}% accuracy" )

            # Check if we exit early do to no update
            if (no_update_counter == n_epochs_no_update):
                if logging:
                    print(f"[INFO] No new change in weight for {n_epochs_no_update} epochs in a row")
                return training_loss,validation_loss, accuracies

        # Return the loss and accuracies over time
        return training_loss,validation_loss, accuracies

    # Get the amount of epochs trained
    def get_epochs_trained(self):
        return self.epochs_trained

    def forward(self, X):
        """Forward method that will do a single forward pass"""
        return logistic(X @ self.weights)

    def score(self, x):
        """Takes input and does a single forward pass"""
        z = add_bias(x)
        score = self.forward(z)
        return score

    def predict(self, x, threshold=0.5):
        """Does a prediction on the data based on """
        z = add_bias(x, self.bias)
        score = self.forward(z)
        return (score>threshold).astype('int')


    def precision(self, y, t):
        """Calculate the precision of the predictions.
        Method implemented for part 3."""

        # True Positive and False Positive
        TP = np.sum((y == 1) & (t == 1))
        FP = np.sum((y == 1) & (t == 0))

        # Avoid division by zero
        if (TP + FP) == 0:
            return 0.0

        # Calculate and return precision
        return (TP / (TP + FP))

    def recall(self, y, t):
        """Calculate the recall of the predictions.
        Method implemented for part 3."""

        # True Positive (TP) and False Negative (FN) counts
        TP = np.sum((y == 1) & (t == 1))
        FN = np.sum((y == 0) & (t == 1))

        # Avoid division by zero
        if (TP + FN) == 0:
            return 0.0

        # Calculate and return recall
        return (TP / (TP + FN))

    # Since the sigmoid function naturally outputs the probability that a class belongs to the given output
    # we can just use the score as the probability
    def predict_probability(self, x):
        """Predict the probability that the data belongs to the positive class"""
```

```
        z = add_bias(x, self.bias)
        probability = self.forward(z)
        return probability


    # For task c)
    # Formula from: https://en.wikipedia.org/wiki/Cross-entropy
    def cross_entropy_loss(self, x, t, add_train=False):
        """Cross entropy loss function for calculating the current loss"""

        # Check if we need to add bias
        if add_train:
            X = add_bias(x, self.bias)
        else:
            X = x

        y_hat = logistic(X@self.weights)
        y = t

        # fix log(0) errors that lead to infinite by adding a very small value to the y_hat
        epsilon = 1e-15
        y_hat = np.clip(y_hat, epsilon, 1 - epsilon)

        # Return the mean of cross entropy loss for each of the data points
        return np.mean(-y*np.log(y_hat)-(1-y)*np.log(1-y_hat))

    # For task c)
    # Calculate the accuracy
    def accuracy(self, x, t):
        score = self.predict(x)
        return np.mean(score == t)
```

In [589…

```
# For task F) testing and training
log_classifier = LogisticClassifier()

# Training with scaled data
train_loss_over_time, val_loss_over_time, accuracy_over_time = log_classifier.fit(X_train,t2_train, X_val, t2_val, eta=0.001, tol=1e-7, epochs=200_000
```

```
EPOCH 1000:         0.6093209157308521 loss, 66.8% accuracy
EPOCH 2000:         0.5763503846200017 loss, 73.2% accuracy
EPOCH 3000:         0.5590513299811405 loss, 75.0% accuracy
EPOCH 4000:         0.5489421283282246 loss, 76.0% accuracy
EPOCH 5000:         0.5425951259314381 loss, 76.0% accuracy
EPOCH 6000:         0.5384047422593317 loss, 75.8% accuracy
EPOCH 7000:         0.535533790595985 loss, 75.2% accuracy
EPOCH 8000:         0.533510018214683 loss, 75.4% accuracy
EPOCH 9000:         0.5320507289534201 loss, 75.4% accuracy
EPOCH 10000:         0.5309787173635966 loss, 75.4% accuracy
EPOCH 11000:         0.5301787638587935 loss, 75.4% accuracy
EPOCH 12000:         0.529573698593882 loss, 75.4% accuracy
EPOCH 13000:         0.5291105584727697 loss, 75.4% accuracy
EPOCH 14000:         0.52875224407831 loss, 75.4% accuracy
EPOCH 15000:         0.5284723143898363 loss, 75.4% accuracy
EPOCH 16000:         0.5282516425523162 loss, 75.4% accuracy
EPOCH 17000:         0.5280762127362697 loss, 75.4% accuracy
EPOCH 18000:         0.5279356369992564 loss, 75.4% accuracy
EPOCH 19000:         0.5278221379211323 loss, 75.4% accuracy
[INFO] No new change in weight for 5 epochs in a row
```

Task F) tested hyper parameters

First, I tested without scaling the data and max epochs of `2=0_000`. This would allow me to test how good performance I will get without setting an restriction on the amount of epochs.

Learning Rate	Tolerance	Epochs Trained	Accuracy
0.1	0.1	5	59.6%
0.01	0.1	5	56.2%
0.01	1e-4	354	76%
0.001	1e-6	9724	75.4%
0.001	1e-7	19103	75.4%

The best configuration was **76%**. It did this with 354 epochs.

The next testing was **with scaling** the training data. I increased the epoch maximum to `200_000` Again, I used the normalizing scaler I created for the earlier task:

Learning Rate	Tolerance	Epochs Trained	Accuracy
0.1	0.1	5	60.0%
0.01	0.1	5	60.0%
0.01	1e-4	52	60.0%
0.001	1e-6	44302	75.4%
0.001	1e-7	200000	73.6%
0.01	1e-7	168248	77.4%
0.1	1e-8	56219	77.6%
0.1	1e-9	78674	77.6%

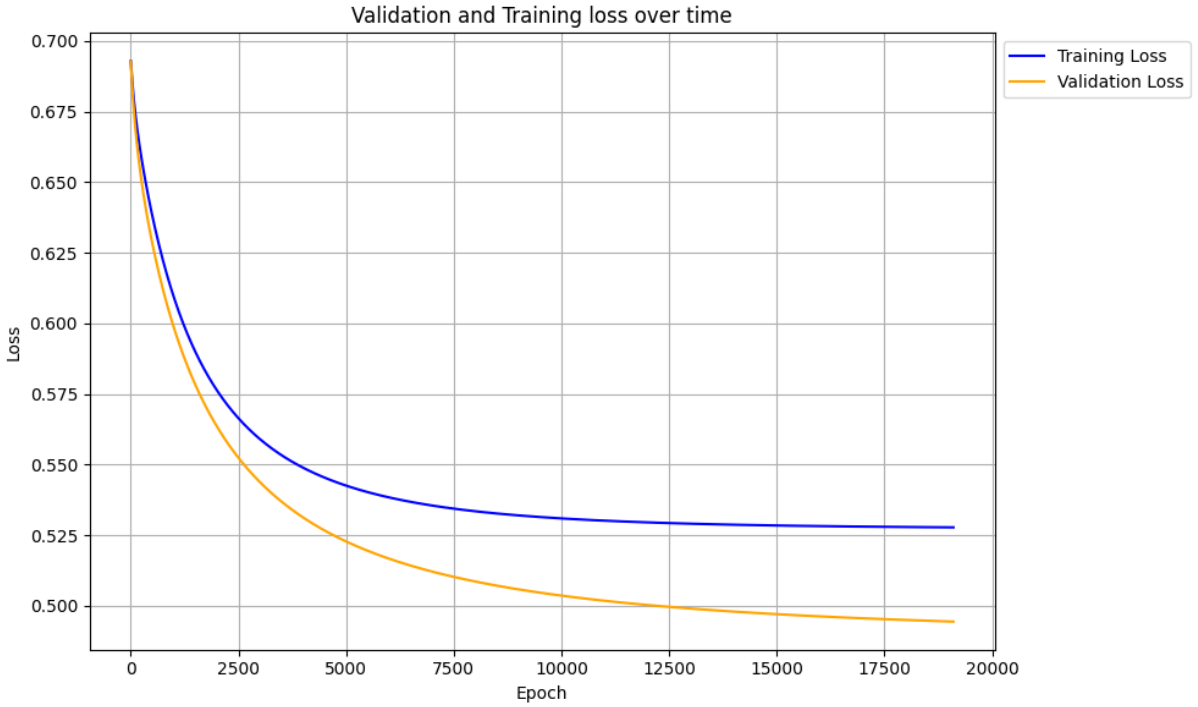
With scaled data, I got the best result when the learning rate was high and tolerance was very low. The best performance I got was `77.6%`. During the development, I got an error where loss was infinite. This happens when we take `log(0)` which lead to infinite high number. To fix this, I added a small value to `y_hat` such that the cross entropy loss would not be infinite. Also I first printing the validation cost over time, but printing the training loss is better. It matches the training

Task G)

In [558…

```
def plot_loss(train_loss_over_time, val_loss_over_time):
    # Plot validation and training loss
    x_data = np.arange(len(train_loss_over_time))
```

```
    # Assert that we have the expected length
    assert len(train_loss_over_time) == len(val_loss_over_time), "Training and validation loss does not have the same amount of entries"

    # Create the plot and show it
    plt.figure(figsize=(10,6))
    plt.plot(x_data, train_loss_over_time, label='Training Loss', color='blue')
    plt.plot(x_data, val_loss_over_time, label='Validation Loss', color='orange')

    # Add text to axis and plot
    plt.title("Validation and Training loss over time")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")

    # Add legend outside the plot with grid
    plt.legend(loc='upper left', bbox_to_anchor=(1, 1))
    plt.tight_layout()
    plt.grid(True)

    # Show the plot
    plt.show()

# Call the method
plot_loss(train_loss_over_time=train_loss_over_time, val_loss_over_time=val_loss_over_time)
```

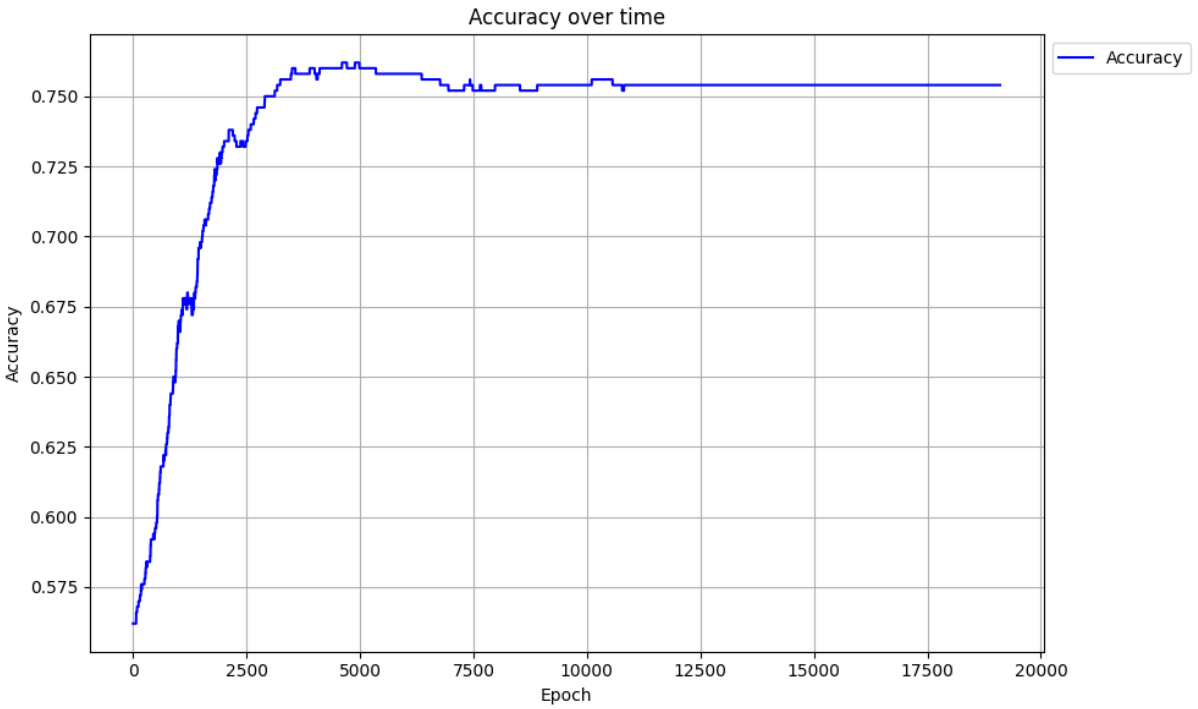


First, we see that the training loss goes down for each epoch of training. It really slows down towards the end but it still goes down. The most interesting is how the curves are not monotone. This means that we are **overfitting** when we train. The validation loss goes down very fast but then it goes up quite quickly again. It also goes above 1, which indicates that we are making very bad predictions for the validation training set.

Here is also plotting of the accuracy over time. Again this is only accuracy over time for training set, which is overfitting:

In [559…

```
def plot_accuracy(accuracy_over_time):
    """Method that simply plots accuracy over time"""
    # Plot validation and training loss
    x_data = np.arange(len(accuracy_over_time))

    # Create the plot and show it
    plt.figure(figsize=(10,6))
    plt.plot(x_data, accuracy_over_time, label='Accuracy', color='blue')

    # Add text to axis and plot
    plt.title("Accuracy over time")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")

    # Add legend outside the plot
    plt.legend(loc='upper left', bbox_to_anchor=(1, 1))

    # Show the plot
    plt.tight_layout()  # Adjust layout to make room for the legend

    plt.grid(True)
    plt.show()

plot_accuracy(accuracy_over_time=accuracy_over_time)
```



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 [560…

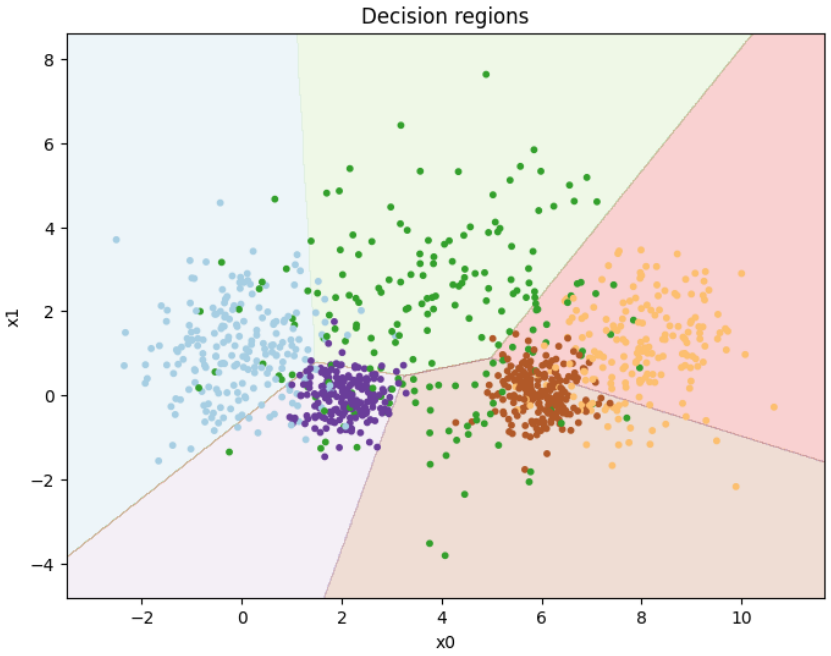
```
class OneVRestReg:
    def __init__(self, n):
        """Initializes n logistic classifiers for n classes"""
        self.classes = n

        # Create n Logistic classifiers, one for each class
        self.classifiers = [LogisticClassifier() for _ in range(n)]

    def fit(self, X, y, X_val, y_val, epochs=100, eta=0.1, tol=0.03, logging=False):
        """Fits one classifier per class using One-vs-Rest strategy"""
        # Iterate through each class and fit them individually
        for i in range(self.classes):
            # Create binary labels for the current class (1 if it's class i, 0 otherwise)
            binary_targets_train = (y == i).astype(int)
            binary_targets_val = (y_val == i).astype(int)

            # Train the i-th logistic classifier using these binary labels
            # Logging will be false for training
            # Ignore the output of the fit, since we are reusing code
            _ = self.classifiers[i].fit(X_train=X, t_train=binary_targets_train,
                                        X_val=X_val, t_val=binary_targets_val,
                                        eta=eta, epochs=epochs,tol=tol, logging=logging)

    def predict(self, X):
        """Predicts the class labels for each sample in X"""
        # Initialize an array to store the scores for each class for each sample
        scores = np.zeros((X.shape[0], self.classes))

        # Get the scores (probabilities) for each class
        for i in range(self.classes):
            # Get the probability scores for the current class
            scores[:, i] = self.classifiers[i].predict_probability(X)

        # For each sample, return the class with the probability
        # The biggest probability is assigned to that given class
        return np.argmax(scores, axis=1)


    def accuracy(self, x, t):
        score = self.predict(x)
        return np.mean(score == t)

    def precision(self, x, t):
        """Calculates recall for the multi-class classifier. For part 3"""
        predictions = self.predict(x)

        TP = 0
        FP = 0

        # Calculate TP and FP for each class
        for class_label in range(self.classes):
            TP += np.sum((predictions == class_label) & (t == class_label))
            FP += np.sum((predictions == class_label) & (t != class_label))

        # Avoid dividing by 0
```

```
        if (TP + FP) == 0:
            return 0

        # Calculate and return the precision
        return (TP / (TP + FP) )

    def recall(self, x, t):
        """Calculates recall for the multi-class classifier. For part 3"""
        predictions = self.predict(x)

        TP = 0
        FN = 0

        # Calculate TP and FP for each class
        for class_label in range(self.classes):
            TP += np.sum((predictions == class_label) & (t == class_label))
            FN += np.sum((predictions != class_label) & (t == class_label))

        # Avoid dividing by 0
        if (TP + FN) == 0:
            return 0

        # Calculate and return the recall
        return (TP / (TP + FN))
```

In [561…
```
# Train the classifier
classes = len(np.unique(t_multi))
one_v_rest_model = OneVRestReg(n=classes)

# Fit the model
one_v_rest_model.fit(X=X_train, y=t_multi_train, X_val=X_val, y_val=t_multi_val, eta=0.1, tol=0.00009, epochs=10_000)

# Print the accuracy of the model
model_accuracy = one_v_rest_model.accuracy(x=X_val, t=t_multi_val)
print(f"Model is {round(model_accuracy*100, 2)}% accurate on the validation set")
```

```
Model is 82.6% accurate on the validation set
```

In [562…
```
# Plot the decision boundaries
plot_decision_regions(X_train,t_multi_train, clf=one_v_rest_model)
```



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 [590…
```
def softmax(x):
    return np.exp(x) / np.sum(np.exp(x))


# Multinomial Logistic Regression class
class MultinomialLogisticReg:
    def __init__(self, n_classes, bias):
        """Initializes the weight matrix for n_classes."""
        # Classes represent how many classes the multinomial model is for
        self.classes = n_classes
        self.bias = bias

    def fit(self, X, y, X_val, t_val, epochs=100, eta=0.1, logging=False):
        """Fit the model using gradient descent."""
        (N, M) = X.shape

        # Init weights: (M + 1 features, n_classes)
        self.weights = np.zeros((M + 1, self.classes))

        X = add_bias(X, self.bias)

        # Iterate over epochs
        for i in range(epochs):
            # Apply softmax to get the class probabilities
```

```
            probabilities = X @ self.weights

            # Convert y to one-hot encoding
            y_one_hot = np.eye(self.classes)[y]

            # Compute gradient:
            error = probabilities - y_one_hot
            grad = (X.T @ error) / N

            self.weights -= eta * grad

            # Get accuracy
            current_accuracy = self.accuracy(X_val, t_val)
            loss = self.cross_entropy_loss(X=X_val, t=t_val)
            if (i+1) % 1000 == 0 and logging:
                print(f"Epoch {i+1}:  {loss} loss, {round(current_accuracy*100, 2)}% accuracy")

    def forward(self, x):
        X = add_bias(x, self.bias)
        Z = X @ self.weights
        return softmax(Z)

    def cross_entropy_loss(self, X, t):
        """Compute cross-entropy loss."""
        # Add bias term to the input
        X = add_bias(X, self.bias)

        # Get predicted probabilities
        y_hat = softmax(X @ self.weights)

        # One-hot encode the true labels
        y_one_hot = np.eye(self.classes)[t]

        # Avoid log(0) by clipping values
        epsilon = 1e-15
        y_hat = np.clip(y_hat, epsilon, 1 - epsilon)

        # Compute cross-entropy loss
        return -np.mean(np.sum(y_one_hot * np.log(y_hat), axis=1))


    def predict(self, X):
        """Predicts class labels for each sample in X."""
        # Get the probabilities
        probabilities = self.forward(X)

        # Get the highest probability class for each sample
        return np.argmax(probabilities, axis=1)


    def accuracy(self, X, t):
        """Calculate accuracy on the dataset."""
        y_pred = self.predict(X)
        return np.mean(y_pred == t)


    def precision(self, x, t):
        """Calculate precision for multi-class classifier. For part 3."""
        # Make prediction
        predictions = self.predict(x)

        # Store all precision calculations
        precisions = []

        for class_label in range(self.classes):
            TP = np.sum((predictions == class_label) & (t == class_label))
            FP = np.sum((predictions == class_label) & (t != class_label))

            # Calculate precision for this class
            if (TP + FP) == 0:
                precisions.append(0)
            else:
                # Append the calculation of precision
                precisions.append(TP / (TP + FP))

        # Return the average
        return np.mean(precisions)

    def recall(self, x, t):
        """Calculate recall for multi-class classifier. For part 3."""
        # Make prediction
        predictions = self.predict(x)

        # Store recalls for each class
        recalls = []

        # For each class
        for class_label in range(self.classes):
            TP = np.sum((predictions == class_label) & (t == class_label))
            FN = np.sum((predictions != class_label) & (t == class_label))

            # Calculate recall for this class
            # If the sum is  0, we know it will be 0
            if (TP + FN) == 0:
                recalls.append(0)
            else:
                # Append the calculation of recall
                recalls.append(TP / (TP + FN))

        # Return the average
        return np.mean(recalls)
```

In [591…

```
# Testing and training
classes = len(np.unique(t_multi))
multinomial_model = MultinomialLogisticReg(n_classes=classes, bias=1)

# Fit the model
multinomial_model.fit(X=X_train, y=t_multi_train, X_val=X_val, t_val=t_multi_val, eta=0.001, epochs=200_000, logging=True)
```

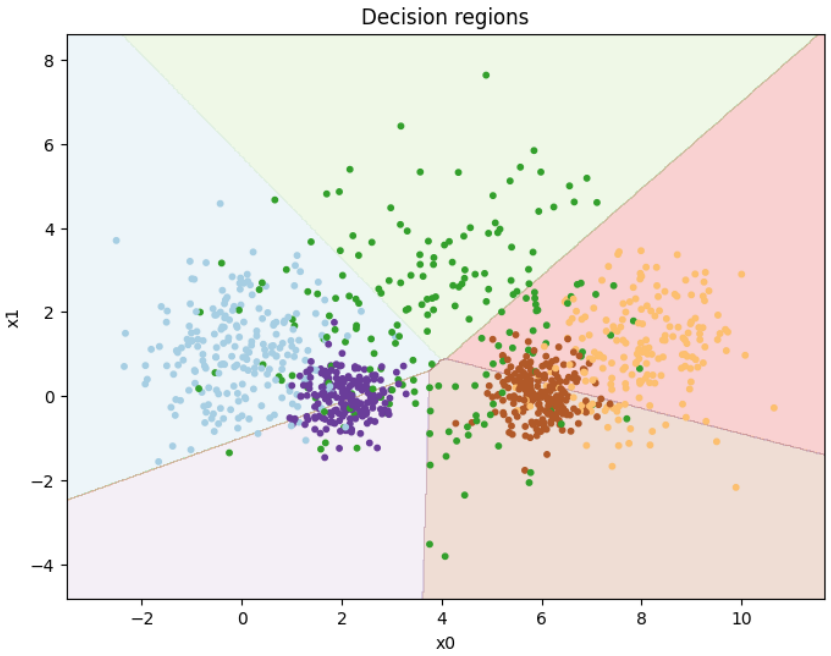
```
# Print the accuracy of the model
model_accuracy = multinomial_model.accuracy(X=X_val, t=t_multi_val)
print(f"Model is {round(model_accuracy*100, 2)}% accurate on the validation set")
```

```
Epoch 1000:  7.6860766547715205 loss, 59.8% accuracy
Epoch 2000:  7.658470256046283 loss, 70.2% accuracy
Epoch 3000:  7.643101022556722 loss, 74.4% accuracy
Epoch 4000:  7.632877043695136 loss, 75.2% accuracy
Epoch 5000:  7.6257354582891494 loss, 75.2% accuracy
Epoch 6000:  7.620668807367624 loss, 75.2% accuracy
Epoch 7000:  7.617048335187082 loss, 74.6% accuracy
Epoch 8000:  7.614449622997952 loss, 74.0% accuracy
Epoch 9000:  7.6125783682352886 loss, 74.0% accuracy
Epoch 10000:  7.611227784853638 loss, 73.6% accuracy
Epoch 11000:  7.610251318140168 loss, 73.6% accuracy
Epoch 12000:  7.609544441420162 loss, 73.4% accuracy
Epoch 13000:  7.609032248410751 loss, 73.6% accuracy
Epoch 14000:  7.6086608679036045 loss, 73.8% accuracy
Epoch 15000:  7.608391453755870 loss, 73.8% accuracy
Epoch 16000:  7.608195939194448 loss, 73.8% accuracy
Epoch 17000:  7.6080540162860215 loss, 73.8% accuracy
Epoch 18000:  7.607950975447908 loss, 73.8% accuracy
Epoch 19000:  7.607876153846211 loss, 73.8% accuracy
Epoch 20000:  7.607821817705402 loss, 73.8% accuracy
Epoch 21000:  7.607782355378443 loss, 73.8% accuracy
Epoch 22000:  7.607753693811382 loss, 73.8% accuracy
Epoch 23000:  7.607732876044813 loss, 73.8% accuracy
Epoch 24000:  7.607717755039624 loss, 73.8% accuracy
Epoch 25000:  7.607706771656189 loss, 73.8% accuracy
Epoch 26000:  7.60769879358083 loss, 73.8% accuracy
Epoch 27000:  7.6076929984285835 loss, 73.8% accuracy
Epoch 28000:  7.607688788885109 loss, 73.8% accuracy
Epoch 29000:  7.607685731095202 loss, 73.8% accuracy
Epoch 30000:  7.607683509923668 loss, 73.8% accuracy
Epoch 31000:  7.6076818964649044 loss, 73.8% accuracy
Epoch 32000:  7.607680724446047 loss, 73.8% accuracy
Epoch 33000:  7.607679873088437 loss, 73.8% accuracy
Epoch 34000:  7.607679254659291 loss, 73.8% accuracy
Epoch 35000:  7.607678805429816 loss, 73.8% accuracy
Epoch 36000:  7.607678479107478 loss, 73.8% accuracy
Epoch 37000:  7.607678242065338 loss, 73.8% accuracy
Epoch 38000:  7.607678069876714 loss, 73.8% accuracy
Epoch 39000:  7.60767794479799 loss, 73.8% accuracy
Epoch 40000:  7.6076778539401255 loss, 73.8% accuracy
Epoch 41000:  7.6076777879404744 loss, 73.8% accuracy
Epoch 42000:  7.607677739997961 loss, 73.8% accuracy
Epoch 43000:  7.607677705172249 loss, 73.8% accuracy
Epoch 44000:  7.607677679874656 loss, 73.8% accuracy
Epoch 45000:  7.60767766149834 loss, 73.8% accuracy
Epoch 46000:  7.60767764814968 loss, 73.8% accuracy
Epoch 47000:  7.607677638453136 loss, 73.8% accuracy
Epoch 48000:  7.607677631409513 loss, 73.8% accuracy
Epoch 49000:  7.607677626292984 loss, 73.8% accuracy
Epoch 50000:  7.607677622576308 loss, 73.8% accuracy
Epoch 51000:  7.607677619876494 loss, 73.8% accuracy
Epoch 52000:  7.607677617915333 loss, 73.8% accuracy
Epoch 53000:  7.607677616490734 loss, 73.8% accuracy
Epoch 54000:  7.607677615455898 loss, 73.8% accuracy
Epoch 55000:  7.607677614704186 loss, 73.8% accuracy
Epoch 56000:  7.607677614158139 loss, 73.8% accuracy
Epoch 57000:  7.607677613761487 loss, 73.8% accuracy
Epoch 58000:  7.607677613473357 loss, 73.8% accuracy
Epoch 59000:  7.607677613264057 loss, 73.8% accuracy
Epoch 60000:  7.607677613112021 loss, 73.8% accuracy
Epoch 61000:  7.607677613001581 loss, 73.8% accuracy
Epoch 62000:  7.607677612921357 loss, 73.8% accuracy
Epoch 63000:  7.607677612863082 loss, 73.8% accuracy
Epoch 64000:  7.6076776128207495 loss, 73.8% accuracy
Epoch 65000:  7.607677612789999 loss, 73.8% accuracy
Epoch 66000:  7.607677612767663 loss, 73.8% accuracy
Epoch 67000:  7.607677612751438 loss, 73.8% accuracy
Epoch 68000:  7.607677612739652 loss, 73.8% accuracy
Epoch 69000:  7.60767761273109 loss, 73.8% accuracy
Epoch 70000:  7.607677612724871 loss, 73.8% accuracy
Epoch 71000:  7.607677612720353 loss, 73.8% accuracy
Epoch 72000:  7.607677612717071 loss, 73.8% accuracy
Epoch 73000:  7.607677612714688 loss, 73.8% accuracy
Epoch 74000:  7.607677612712956 loss, 73.8% accuracy
Epoch 75000:  7.607677612711697 loss, 73.8% accuracy
Epoch 76000:  7.607677612710784 loss, 73.8% accuracy
Epoch 77000:  7.607677612710121 loss, 73.8% accuracy
Epoch 78000:  7.607677612709638 loss, 73.8% accuracy
Epoch 79000:  7.607677612709289 loss, 73.8% accuracy
Epoch 80000:  7.6076776127090335 loss, 73.8% accuracy
Epoch 81000:  7.607677612708849 loss, 73.8% accuracy
Epoch 82000:  7.607677612708715 loss, 73.8% accuracy
Epoch 83000:  7.607677612708617 loss, 73.8% accuracy
Epoch 84000:  7.607677612708547 loss, 73.8% accuracy
Epoch 85000:  7.607677612708496 loss, 73.8% accuracy
Epoch 86000:  7.607677612708458 loss, 73.8% accuracy
Epoch 87000:  7.607677612708432 loss, 73.8% accuracy
Epoch 88000:  7.607677612708414 loss, 73.8% accuracy
Epoch 89000:  7.607677612708398 loss, 73.8% accuracy
Epoch 90000:  7.607677612708387 loss, 73.8% accuracy
Epoch 91000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 92000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 93000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 94000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 95000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 96000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 97000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 98000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 99000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 100000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 101000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 102000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 103000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 104000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 105000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 106000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 107000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 108000:  7.6076776127083825 loss, 73.8% accuracy
```

```
Epoch 109000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 110000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 111000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 112000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 113000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 114000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 115000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 116000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 117000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 118000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 119000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 120000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 121000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 122000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 123000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 124000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 125000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 126000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 127000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 128000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 129000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 130000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 131000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 132000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 133000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 134000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 135000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 136000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 137000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 138000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 139000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 140000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 141000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 142000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 143000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 144000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 145000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 146000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 147000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 148000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 149000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 150000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 151000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 152000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 153000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 154000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 155000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 156000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 157000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 158000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 159000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 160000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 161000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 162000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 163000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 164000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 165000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 166000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 167000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 168000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 169000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 170000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 171000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 172000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 173000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 174000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 175000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 176000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 177000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 178000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 179000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 180000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 181000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 182000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 183000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 184000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 185000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 186000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 187000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 188000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 189000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 190000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 191000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 192000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 193000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 194000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 195000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 196000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 197000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 198000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 199000:  7.6076776127083825 loss, 73.8% accuracy
Epoch 200000:  7.6076776127083825 loss, 73.8% accuracy
Model is 73.8% accurate on the validation set
```

In [565…

```
plot_decision_regions(X_train,t_multi_train, clf=multinomial_model)
```



Was not able to find parameters that lead to better result. However, 73.8% is still very good!

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 [566…
```
# 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 [592…
```
class MLPBinaryLinRegClass(NumpyClassifier):
    """A multi-layer neural network with one hidden layer"""

    def __init__(self, bias=-1, dim_hidden = 6, classes=2, with_tol=False, n_epochs_no_update=5):
        """Initialize the hyperparameters"""
        self.bias = bias
        # Dimensionality of the hidden layer
        self.dim_hidden = dim_hidden

        self.activ = logistic

        self.activ_diff = logistic_diff

        # Binary classification, so two classes
        self.classes = classes

        # For the improved MLP, we use tolerance to terminate,
        # But the normal MLP should not. By default, to tolerance
        self.with_tol = with_tol

        # Used for terminating after the given model has not been able to change the loss over the given amount of epochs
        self.n_epochs_no_update = n_epochs_no_update

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

        # From input layer to hidden layer
        # Get the hidden layer output
        hidden_outs = self.activ(X @ self.weights1)

        # Then add the bias
        hidden_outs = add_bias(hidden_outs, self.bias)

        # From the hidden layer output to the final output
        outputs = self.activ(hidden_outs @ self.weights2)

        return hidden_outs, outputs

    def fit(self, X_train, t_train, X_val, t_val, lr=0.001, epochs = 100, tol=0.001, logging=False):
        """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)

    # Init arrays to keep track of loss and accuracy over each epoch
    val_loss_over_time = []
    train_loss_over_time = []
    accuracy_over_time = []

    # For tolerance checking
    no_update_counter = 0

    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:]))


        loss_before = self.cross_entropy_loss(X=X_train, t=t_train)

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

        # Calculate accuracy on the validation dataset
        current_val_accuracy = self.accuracy(X=X_val, t=t_val)

        # Calculate loss on the validation set
        current_val_loss = self.cross_entropy_loss(X=X_val, t=t_val)
        current_train_loss = self.cross_entropy_loss(X=X_train, t=t_train)


        # If tolerance is enabled
        if self.with_tol:
            # Check if we terminate early due being less than tolerance
            if(np.mean(np.abs(loss_before - current_train_loss)) < tol):
                no_update_counter += 1
            else:
                no_update_counter = 0

        # Append both the array of histories
        accuracy_over_time.append(current_val_accuracy)
        val_loss_over_time.append(current_val_loss)
        train_loss_over_time.append(current_train_loss)

        # Print the validation loss and validation accuracy for each 1000th epoch
        # We do this because we have to train a lot of epochs and printing every epoch can be to much
        if (e+1)%1000 == 0 and logging:
            print(f"Epoch {e+1}:   {current_val_loss} loss, {round(current_val_accuracy*100, 2)}% accuracy")

        # Terminate if no update in a long time
        if self.with_tol:
            if no_update_counter == self.n_epochs_no_update:
                return train_loss_over_time, val_loss_over_time, accuracy_over_time

    # Return after all epochs over, to termination
    return train_loss_over_time, val_loss_over_time, accuracy_over_time


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


# For task a)
def predict_probability(self, X):
    """Predict the class for the members of X.
    The given X should already have a bias appended.
    """
    current_forward = self.forward(X)[1]
    score= current_forward[:, 0]

    # Score as the probability
    # For predicting, we assumed that over 0.5 the class was positive
    # This means that we can just use the scores as probabilities
    return score


def cross_entropy_loss(self, X, t):
    """Compute cross-entropy loss."""
    # Add bias term to the input
    X = add_bias(X, self.bias)
```

```
        # Feed forward to get the output
        y_hat = self.predict_probability(X)

        # Avoid log(0) by clipping values
        epsilon = 1e-15
        y_hat = np.clip(y_hat, epsilon, 1 - epsilon)

        # Compute cross-entropy loss
        return -np.mean(t*np.log(y_hat) + (1-t)*np.log(1-y_hat))


    def accuracy(self, X, t):
        """Calculate accuracy on the dataset."""
        y_pred = self.predict(X)
        return np.mean(y_pred == t)


    def precision(self, x, t):
        """Calculate precision for multi-class classifier. For part 3."""
        # Make prediction
        predictions = self.predict(x)

        # Store all precision calculations
        precisions = []

        for class_label in range(self.classes):
            TP = np.sum((predictions == class_label) & (t == class_label))
            FP = np.sum((predictions == class_label) & (t != class_label))

            # Calculate precision for this class
            if (TP + FP) == 0:
                precisions.append(0)
            else:
                # Append the calculation of precision
                precisions.append(TP / (TP + FP))

        # Return the average
        return np.mean(precisions)

    def recall(self, x, t):
        """Calculate recall for multi-class classifier. For part 3."""
        # Make prediction
        predictions = self.predict(x)

        # Store recalls for each class
        recalls = []

        # For each class
        for class_label in range(self.classes):
            TP = np.sum((predictions == class_label) & (t == class_label))
            FN = np.sum((predictions != class_label) & (t == class_label))

            # Calculate recall for this class
            # If the sum is  0, we know it will be 0
            if (TP + FN) == 0:
                recalls.append(0)
            else:
                # Append the calculation of recall
                recalls.append(TP / (TP + FN))

        # Return the average
        return np.mean(recalls)
```

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 [593…

```
# Train the model and plot
mlp_model = MLPBinaryLinRegClass(dim_hidden=9)
mlp_model.fit(X_train=X_train, t_train=t2_train, X_val=X_val, t_val=t2_val, lr=0.001, epochs=100000, logging=True)

# Print model accuracy
# Print the accuracy of the model
model_accuracy = mlp_model.accuracy(X=X_val, t=t2_val)
print(f"Model is {round(model_accuracy*100, 2)}% accurate on the validation set")
```

```
Epoch 1000:   0.31208934627072055 loss, 87.6% accuracy
Epoch 2000:   0.27863040365449854 loss, 88.8% accuracy
Epoch 3000:   0.25717608474492293 loss, 89.2% accuracy
Epoch 4000:   0.24568598597222693 loss, 90.0% accuracy
Epoch 5000:   0.2397644854457479 loss, 90.4% accuracy
Epoch 6000:   0.23749184724534006 loss, 90.4% accuracy
Epoch 7000:   0.23269334076318768 loss, 90.6% accuracy
Epoch 8000:   0.22479611722347798 loss, 90.8% accuracy
Epoch 9000:   0.21369036217445023 loss, 92.8% accuracy
Epoch 10000:   0.2033001622248818 loss, 93.0% accuracy
Epoch 11000:   0.19809235726677324 loss, 93.6% accuracy
Epoch 12000:   0.195642869888249 loss, 93.4% accuracy
Epoch 13000:   0.19421935276798385 loss, 93.4% accuracy
Epoch 14000:   0.19325506020324346 loss, 93.4% accuracy
Epoch 15000:   0.1926364589017486 loss, 93.4% accuracy
Epoch 16000:   0.1922373198723809 loss, 93.4% accuracy
Epoch 17000:   0.1919626362974126 loss, 93.4% accuracy
Epoch 18000:   0.19177023017605763 loss, 93.4% accuracy
Epoch 19000:   0.19163926032177 loss, 93.4% accuracy
Epoch 20000:   0.19155663554045343 loss, 93.4% accuracy
Epoch 21000:   0.1915130991955865 loss, 93.4% accuracy
Epoch 22000:   0.19150175676549866 loss, 93.4% accuracy
Epoch 23000:   0.19151722907828736 loss, 93.4% accuracy
Epoch 24000:   0.19155506654760826 loss, 93.4% accuracy
Epoch 25000:   0.1916113196242397 loss, 93.4% accuracy
Epoch 26000:   0.19168222687963385 loss, 93.4% accuracy
Epoch 27000:   0.1917640237397357 loss, 93.4% accuracy
Epoch 28000:   0.19185288954104054 loss, 93.4% accuracy
Epoch 29000:   0.1919450277196187 loss, 93.4% accuracy
Epoch 30000:   0.19203683678746353 loss, 93.4% accuracy
Epoch 31000:   0.19212511642742067 loss, 93.4% accuracy
Epoch 32000:   0.19220726710853897 loss, 93.4% accuracy
Epoch 33000:   0.1922814526118663 loss, 93.4% accuracy
Epoch 34000:   0.1923466911053382 loss, 93.4% accuracy
Epoch 35000:   0.19240284431800142 loss, 93.4% accuracy
Epoch 36000:   0.19245050147053833 loss, 93.6% accuracy
Epoch 37000:   0.19249078926741459 loss, 93.6% accuracy
Epoch 38000:   0.192525157048166 loss, 93.6% accuracy
Epoch 39000:   0.19255518071496616 loss, 93.6% accuracy
Epoch 40000:   0.19258241097660045 loss, 93.6% accuracy
Epoch 41000:   0.19260827335364353 loss, 93.6% accuracy
Epoch 42000:   0.1926340152569278 loss, 93.6% accuracy
Epoch 43000:   0.1926606895612082 loss, 93.6% accuracy
Epoch 44000:   0.19268916260327013 loss, 93.6% accuracy
Epoch 45000:   0.1927201356147836 loss, 93.6% accuracy
Epoch 46000:   0.19275417097076616 loss, 93.6% accuracy
Epoch 47000:   0.192791717420724 loss, 93.6% accuracy
Epoch 48000:   0.1928331310638357 loss, 93.6% accuracy
Epoch 49000:   0.19287869086310958 loss, 93.6% accuracy
Epoch 50000:   0.1929286088367693 loss, 93.6% accuracy
Epoch 51000:   0.19298303578179413 loss, 93.6% accuracy
Epoch 52000:   0.1930420636420423 loss, 93.6% accuracy
Epoch 53000:   0.19310572561482592 loss, 93.6% accuracy
Epoch 54000:   0.19317399493911425 loss, 93.6% accuracy
Epoch 55000:   0.19324678311401366 loss, 93.6% accuracy
Epoch 56000:   0.19332393809886422 loss, 93.6% accuracy
Epoch 57000:   0.19340524285793573 loss, 93.6% accuracy
Epoch 58000:   0.19349041443232143 loss, 93.6% accuracy
Epoch 59000:   0.19357910354697194 loss, 93.6% accuracy
Epoch 60000:   0.19367089459446532 loss, 93.6% accuracy
Epoch 61000:   0.19376530568835926 loss, 93.6% accuracy
Epoch 62000:   0.19386178836328471 loss, 93.6% accuracy
Epoch 63000:   0.19395972643610604 loss, 93.6% accuracy
Epoch 64000:   0.19405843355498661 loss, 93.6% accuracy
Epoch 65000:   0.19415714907536258 loss, 93.6% accuracy
Epoch 66000:   0.19425503214065107 loss, 93.6% accuracy
Epoch 67000:   0.19435115424190866 loss, 93.6% accuracy
Epoch 68000:   0.19444449111849213 loss, 93.6% accuracy
Epoch 69000:   0.19453391566899814 loss, 93.6% accuracy
Epoch 70000:   0.19461819456514542 loss, 93.6% accuracy
Epoch 71000:   0.19469599241628663 loss, 93.6% accuracy
Epoch 72000:   0.19476588837703115 loss, 93.6% accuracy
Epoch 73000:   0.19482641054509592 loss, 93.6% accuracy
Epoch 74000:   0.1948760926246177 loss, 93.6% accuracy
Epoch 75000:   0.19491355430185756 loss, 93.6% accuracy
Epoch 76000:   0.19493760113750705 loss, 93.6% accuracy
Epoch 77000:   0.19494733216755883 loss, 93.6% accuracy
Epoch 78000:   0.19494223615771647 loss, 93.6% accuracy
Epoch 79000:   0.1949222542703847 loss, 93.6% accuracy
Epoch 80000:   0.19488779105106113 loss, 93.6% accuracy
Epoch 81000:   0.1948396674118707 loss, 93.6% accuracy
Epoch 82000:   0.19477902470956054 loss, 93.6% accuracy
Epoch 83000:   0.19470720156972585 loss, 93.4% accuracy
Epoch 84000:   0.19462560965183537 loss, 93.4% accuracy
Epoch 85000:   0.19453563029102616 loss, 93.4% accuracy
Epoch 86000:   0.19443854423609563 loss, 93.4% accuracy
Epoch 87000:   0.19433549629258698 loss, 93.4% accuracy
Epoch 88000:   0.19422748906780435 loss, 93.4% accuracy
Epoch 89000:   0.19411539639234957 loss, 93.4% accuracy
Epoch 90000:   0.19399998674583968 loss, 93.4% accuracy
Epoch 91000:   0.19388194877492787 loss, 93.4% accuracy
Epoch 92000:   0.19376191348316685 loss, 93.4% accuracy
Epoch 93000:   0.1936404700851417 loss, 93.4% accuracy
Epoch 94000:   0.19351817451029468 loss, 93.4% accuracy
Epoch 95000:   0.19339555107000758 loss, 93.4% accuracy
Epoch 96000:   0.1932730889242395 loss, 93.4% accuracy
Epoch 97000:   0.1931512357320032 loss, 93.4% accuracy
Epoch 98000:   0.19303039121123983 loss, 93.4% accuracy
Epoch 99000:   0.192910903236555 loss, 93.4% accuracy
Epoch 100000:   0.1927930686295301 loss, 93.4% accuracy
Model is 93.4% accurate on the validation set
```

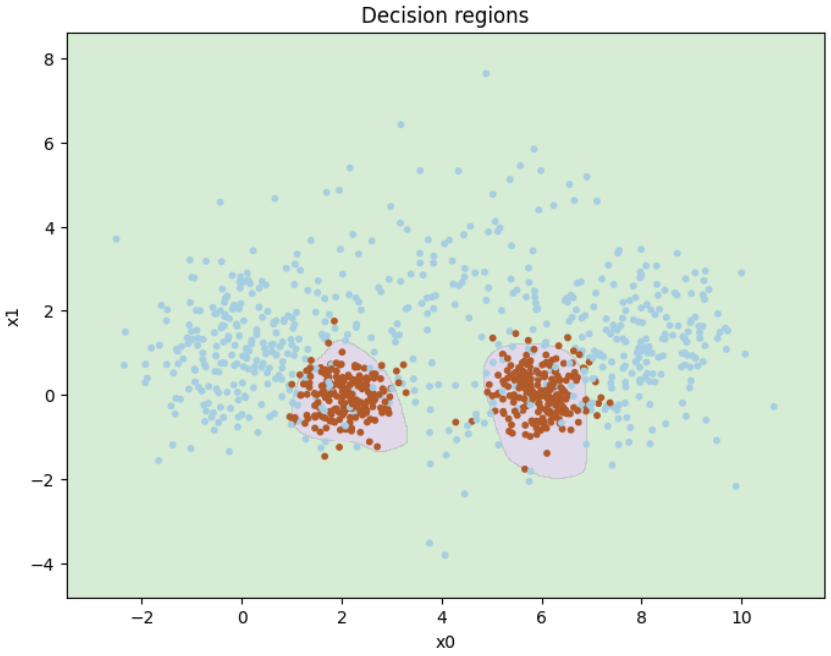
Tested for MPL Binary regression (before changes)

Learning Rate	Hidden Layer dim	Epochs	Accuracy
0.001	6	100	81.2%

Learning Rate	Hidden Layer dim	Epochs	Accuracy
0.01	6	1000	81.9%
0.01	6	1000000	90.0%
0.000001	6	1000000	90.0%
0.01	7	1000	82.0%
0.001	7	1000	85.6%
0.001	7	10000	91.0%
0.001	8	10000	90.8%
0.001	8	100000	93.6%
0.001	8	1000000	93.4%
0.001	9	100000	93.8%

Due to overfitting, I stopped on the last configuration. It had a really high validation accuracy and took 20 seconds to train. Note that the running the last configuration will lead to accuracy between 93.2-93.8%

In [594…
```
# Plot model
plot_decision_regions(X_train,t2_train, clf=mlp_model)
```



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 [570…
```
# The best configuration for the improved MLP
# Train the model and plot
mlp_model = MLPBinaryLinRegClass(dim_hidden=9, with_tol=True)

# Fit the model
train_loss_over_time, val_loss_over_time, accuracy_over_time = mlp_model.fit(X_train=X_train, t_train=t2_train, X_val=X_val, t_val=t2_val, lr=0.0005,

# Print model accuracy
# Print the accuracy of the model
model_accuracy = mlp_model.accuracy(X=X_val, t=t2_val)
print(f"Model is {round(model_accuracy*100, 2)}% accurate on the validation set")
```

```
Model is 93.6% accurate on the validation set
```

Improved MLP Hyper Parameters training

First without scaling the data. Note that epochs trained are to the nearest 100th. This is only a variable to have an idea of how many epochs was trained. Also the max amount of epochs tested are set to `300_000`. I found that during training, the model does not improve much after this number of epochs (even though the cost function changes). These hyper

parameters was tested:

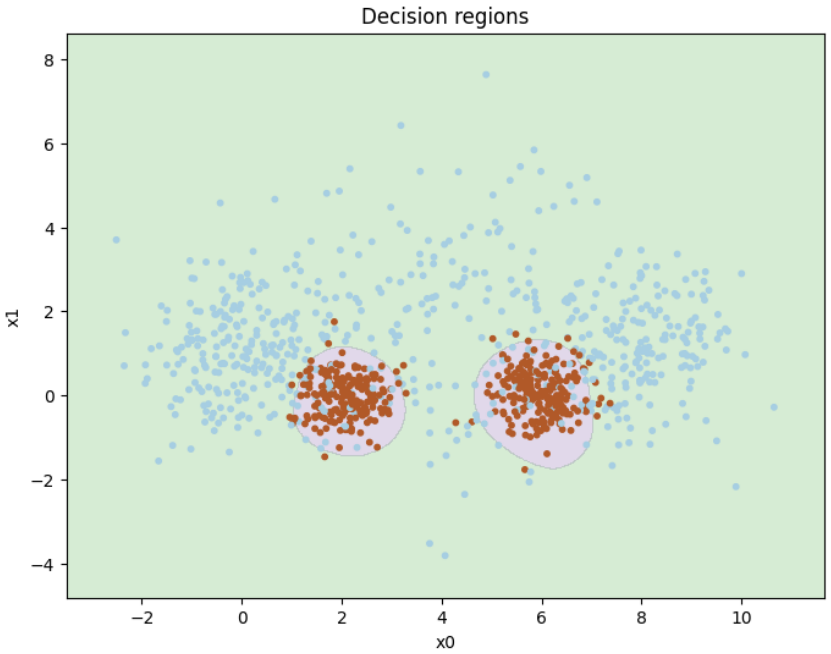
Learning Rate	Hidden Layer dim	Epochs Trained	Tolerance	Accuracy
0.01	6	800	1e-3	86.2%
0.01	6	600	0.01	80.0%
0.01	6	1000	1e-4	81.4%
0.001	6	500	1e-4	82.2%
0.01	7	5400	1e-4	90.8%
0.001	7	6300	1e-5	90.8%
0.001	8	5700	1e-5	88.6%
0.001	9	95100	1e-5	93.8%
0.0001	9	5900	1e-5	90.6%
0.0003	9	27000	1e-6	93.6%
0.0005	9	17900	1e-6	93.8%

Stopped testing after I got `93.8%`. This is very accurate, might even be overfitting. Then I scaled the data and tested again:

Learning Rate	Hidden Layer dim	Epochs Trained	Tolerance	Accuracy
0.01	6	3100	1e-5	60.0%
0.001	6	3100	1e-5	60.8%
0.01	7	23100	1e-5	60.0%
0.001	7	3600	1e-5	60.2%
0.01	8	4600	1e-5	59.4%
0.001	8	4600	1e-5	59.4%
0.000001	8	3000	1e-6	60.4%

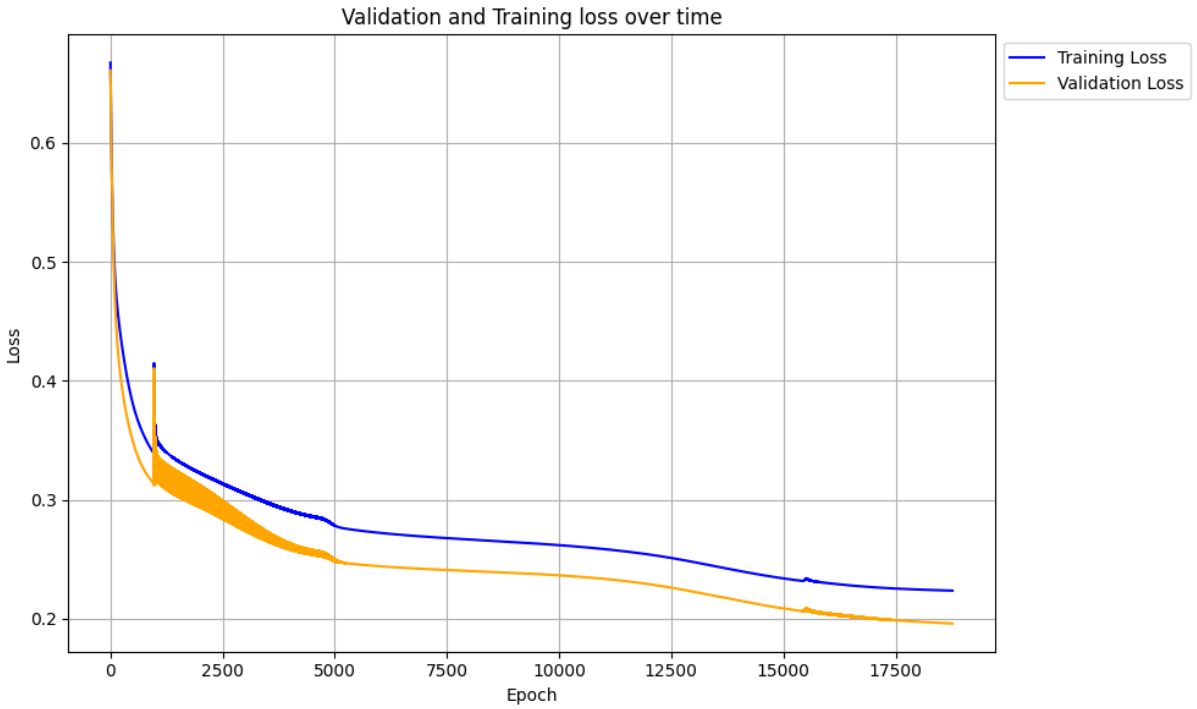
The scaled data did not perform well. In the log, I see the cost of the validation becomes lower than 0.7, but then, since the training data loss is changing, it does **overfitting** to the training data. This leads to a bad performance on the validation data. The scaled training data is scaled in such a way where the validation data does not fit. Therefor, the best model is one without scaling

In [571…
```
# Plot the decision boundary
plot_decision_regions(X_train,t2_train, clf=mlp_model)
```



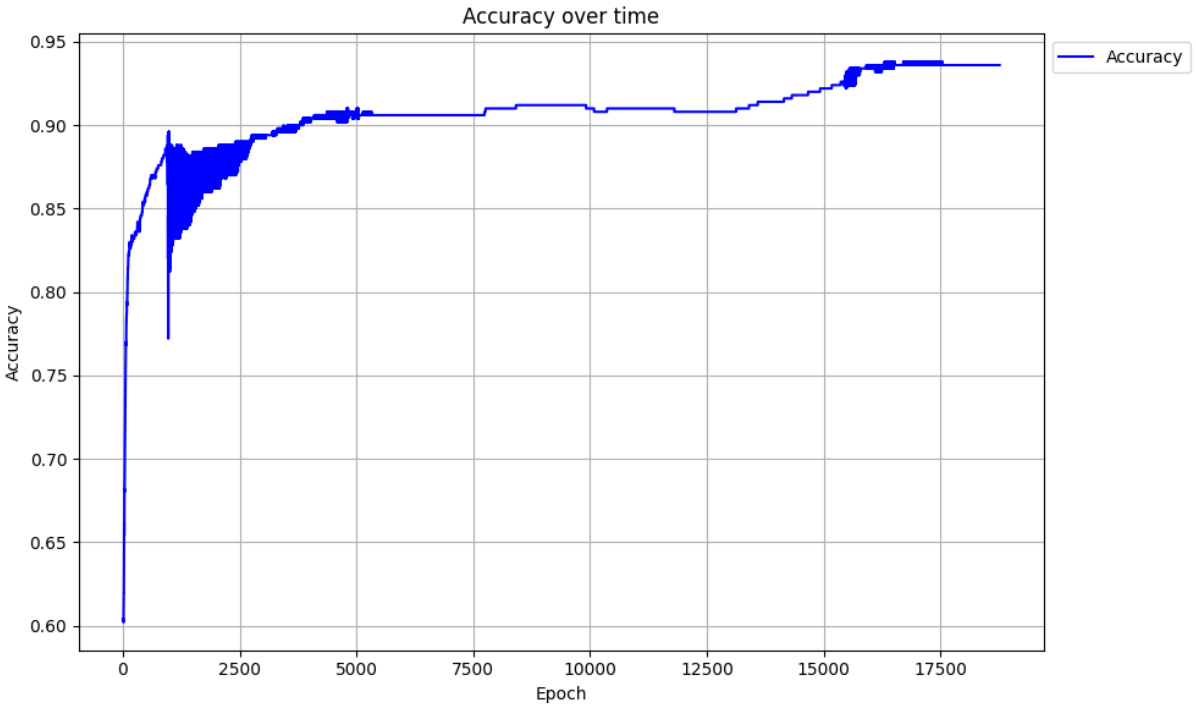
The plot shows that it is able to learn the features of the data quite well. It seems to also not overfit.

In [572…
```
# Plot validation, and training loss
plot_loss(train_loss_over_time=train_loss_over_time, val_loss_over_time=val_loss_over_time)
```



The validation and training loss follows each other quite well. The validation loss is even lower than the training loss. Also, it seems to be learning even better towards the end.

In [573…
```
# Plot accuracy
plot_accuracy(accuracy_over_time=accuracy_over_time)
```



Accuracy is going very fast up and down. This can be due to the fact that it is looking for the optima, and is exploring a lot of options turing tuning. Then the accuracy slowly improves until it does not improve anymore.

Task g) running 10 times

In [584…
```
# Variable for control runs
runs = 10

accuracies = []

for i in range(runs):
    # Create a new model
    current_model = MLPBinaryLinRegClass(dim_hidden=9, with_tol=True)

    # Fit the model
    _ = current_model.fit(X_train=X_train, t_train=t2_train, X_val=X_val, t_val=t2_val, lr=0.0005, tol=1e-6, epochs=300_000)

    # Calculate the model
    mlp_model_accuracy = current_model.accuracy(X=X_val, t=t2_val)
    print(f"Run {i+1}: {round(mlp_model_accuracy*100, 2)}% accuracy")

    accuracies.append(mlp_model_accuracy)


# Calculate and print the mean and std of the accuracy for the 10 runs
mean_accuracy = np.mean(accuracies)
std_accuracy = np.std(accuracies)

print("\nFinal Statistics:")
print(f"Mean accuracy: {round(mean_accuracy*100, 2)}%")
```

```
print(f"Standard deviation: {std_accuracy}")
```

```
Run 1: 93.4% accuracy
Run 2: 93.8% accuracy
Run 3: 93.8% accuracy
Run 4: 91.0% accuracy
Run 5: 93.8% accuracy
Run 6: 93.4% accuracy
Run 7: 93.6% accuracy
Run 8: 93.6% accuracy
Run 9: 91.0% accuracy
Run 10: 93.6% accuracy

Final Statistics:
Mean accuracy: 93.1%
Standard deviation: 0.010592450141492276
```

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 [574…

```
# For multi-class classification we use softmax as the activation function
def softmax(x):
    # Here we create the same softmax.
    # This time, during training, I got an error for overflowing.
    # After research, I found that we can solve this by reducing it by the max value
    # https://stackoverflow.com/questions/34968722/how-to-implement-the-softmax-function-in-python
    # Apply numerical stability trick to avoid overflow/underflow
    exps = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exps / np.sum(exps, axis=1, keepdims=True)

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

class MultiClassFeedForwardNetwork(NumpyClassifier):
    """A multi-layer neural network with one hidden layer.
    I changed the code, so that I add the activation functions, explicitly."""

    def __init__(self, bias=-1, dim_hidden = 6, classes=2, with_tol=False, n_epochs_no_update=5):
        """Initialize the hyperparameters"""
        self.bias = bias
        # Dimensionality of the hidden layer
        self.dim_hidden = dim_hidden

        # Binary classification, so two classes
        self.classes = classes

        # For the improved MLP, we use tolerance to terminate,
        # But the normal MLP should not. By default, to tolerance
        self.with_tol = with_tol

        # Used for terminating after the given model has not been able to change the loss over the given amount of epochs
        self.n_epochs_no_update = n_epochs_no_update

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

        X_bias = add_bias(X, self.bias)

        # From input layer to hidden layer
        hidden_outs = sigmoid(X_bias @ self.weights1)

        # Then add the bias
        hidden_outs = add_bias(hidden_outs, self.bias)

        # Then use softmax to get the final values.
        outputs = softmax(hidden_outs @ self.weights2)

        return hidden_outs, outputs

    def fit(self, X_train, t_train, X_val, t_val, lr=0.001, epochs = 100, tol=0.001, logging=False):
        """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 = np.eye(self.classes)[t_train]

        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)


        # Init arrays to keep track of loss and accuracy over each epoch
        val_loss_over_time = []
        train_loss_over_time = []
```

```
    accuracy_over_time = []

    # For tolerance checking
    no_update_counter = 0

    for e in range(epochs):
        X_bias = add_bias(X_train, self.bias)

        # Do a single feed forward
        hidden_outs, outputs = self.forward(X_train)

        # Output layer error
        out_deltas = outputs - T_train

        # Hidden layer errors
        hiddenout_diffs = out_deltas @ self.weights2.T

        # Calculate the sigmoid derivative and multiply it to the output layer
        hiddenact_deltas = hiddenout_diffs[:, 1:] * (hidden_outs[:, 1:] * (1 - hidden_outs[:, 1:]))

        # Weight updates with gradient clipping
        grad1 = X_bias.T @ hiddenact_deltas
        grad2 = hidden_outs.T @ out_deltas

        # Add gradient clipping to fix bug for overflowing
        clip_value = 5
        grad1 = np.clip(grad1, -clip_value, clip_value)
        grad2 = np.clip(grad2, -clip_value, clip_value)


        # Calculate the loss before changing the weights
        loss_before = self.cross_entropy_loss(X=X_train, t=t_train)

        self.weights1 -= self.lr * grad1
        self.weights2 -= self.lr * grad2

        # Calculate accuracy on the validation dataset
        current_val_accuracy = self.accuracy(X=X_val, t=t_val)

        # Calculate loss on the validation
        current_val_loss = self.cross_entropy_loss(X=X_val, t=t_val)
        current_train_loss = self.cross_entropy_loss(X=X_train, t=t_train)


        # If tolerance is enabled
        if self.with_tol:
            # Check if we terminate early due being less than tolerance
            if(np.mean(np.abs(loss_before - current_train_loss)) < tol):
                no_update_counter += 1
            else:
                no_update_counter = 0

        # Append both the array of histories
        accuracy_over_time.append(current_val_accuracy)
        val_loss_over_time.append(current_val_loss)
        train_loss_over_time.append(current_train_loss)

        # Print the validation loss and validation accuracy for each 1000th epoch
        if (e+1)%1000 == 0 and logging:
            print(f"Epoch {e+1}:   {current_val_loss} loss, {round(current_val_accuracy*100, 2)}% accuracy")

        # Terminate if no update in a long time
        if self.with_tol:
            if no_update_counter == self.n_epochs_no_update:
                return train_loss_over_time, val_loss_over_time, accuracy_over_time

    # Return after all epochs over, to termination
    return train_loss_over_time, val_loss_over_time, accuracy_over_time

def predict(self, X):
    """Predict the class for the members of X"""

    # Do forward step, and only use the final output of the last layer
    output = self.forward(X)[1]

    # Return the maximum of all probabilities
    return np.argmax(output, axis=1)


def predict_probability(self, X):
    """Predict the class for the members of X.
    The given X should already have a bias appended."""

    # Do forward step, and only use the final output of the last layer
    output = self.forward(X)[1]

    # Return the maximum of all probabilities
    return output


def cross_entropy_loss(self, X, t):
    """Compute cross-entropy loss."""
    # Feed forward to get the output
    y_hat = self.predict_probability(X)

    # Avoid log(0) by clipping values
    epsilon = 1e-15
    y_hat = np.clip(y_hat, epsilon, 1 - epsilon)

    # Convert true labels to one-hot encoding
    T = np.eye(self.classes)[t]

    # Compute cross-entropy loss
    return -np.mean(T*np.log(y_hat))


def accuracy(self, X, t):
    """Calculate accuracy on the dataset."""
```

```
        y_pred = self.predict(X)
        return np.mean(y_pred == t)

    def precision(self, x, t):
        """Calculate precision for multi-class classifier. For part 3."""
        # Make prediction
        predictions = self.predict(x)

        # Store all precision calculations
        precisions = []

        for class_label in range(self.classes):
            TP = np.sum((predictions == class_label) & (t == class_label))
            FP = np.sum((predictions == class_label) & (t != class_label))

            # Calculate precision for this class
            if (TP + FP) == 0:
                precisions.append(0)
            else:
                # Append the calculation of precision
                precisions.append(TP / (TP + FP))

        # Return the average
        return np.mean(precisions)

    def recall(self, x, t):
        """Calculate recall for multi-class classifier. For part 3."""
        # Make prediction
        predictions = self.predict(x)

        # Store recalls for each class
        recalls = []

        # For each class
        for class_label in range(self.classes):
            TP = np.sum((predictions == class_label) & (t == class_label))
            FN = np.sum((predictions != class_label) & (t == class_label))

            # Calculate recall for this class
            # If the sum is  0, we know it will be 0
            if (TP + FN) == 0:
                recalls.append(0)
            else:
                # Append the calculation of recall
                recalls.append(TP / (TP + FN))

        # Return the average
        return np.mean(recalls)
```

Trained Hyper parameters for Multi-class feed forward Neural Network

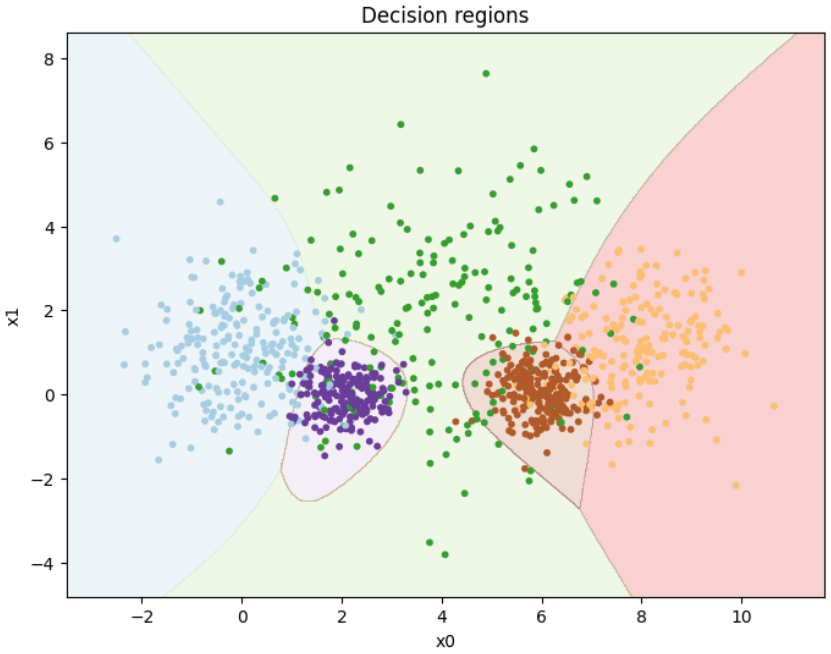
Learning Rate	Hidden Layer dim	Epochs Trained	Tolerance	Accuracy
1e-3	6	1000	1e-3	49.6%
1e-3	6	1000	1e-4	86.4%
1e-2	6	2000	1e-4	84.2%
1e-3	6	3000	1e-5	88.2%
1e-3	7	3000	1e-5	88.2%
1e-3	7	7000	1e-6	87.8%
1e-3	8	4000	1e-5	87.6%
1e-4	8	3000	1e-6	87.8%
1e-4	8	11000	1e-6	87.6%
1e-4	8	26000	1e-7	88.0%

In [575…

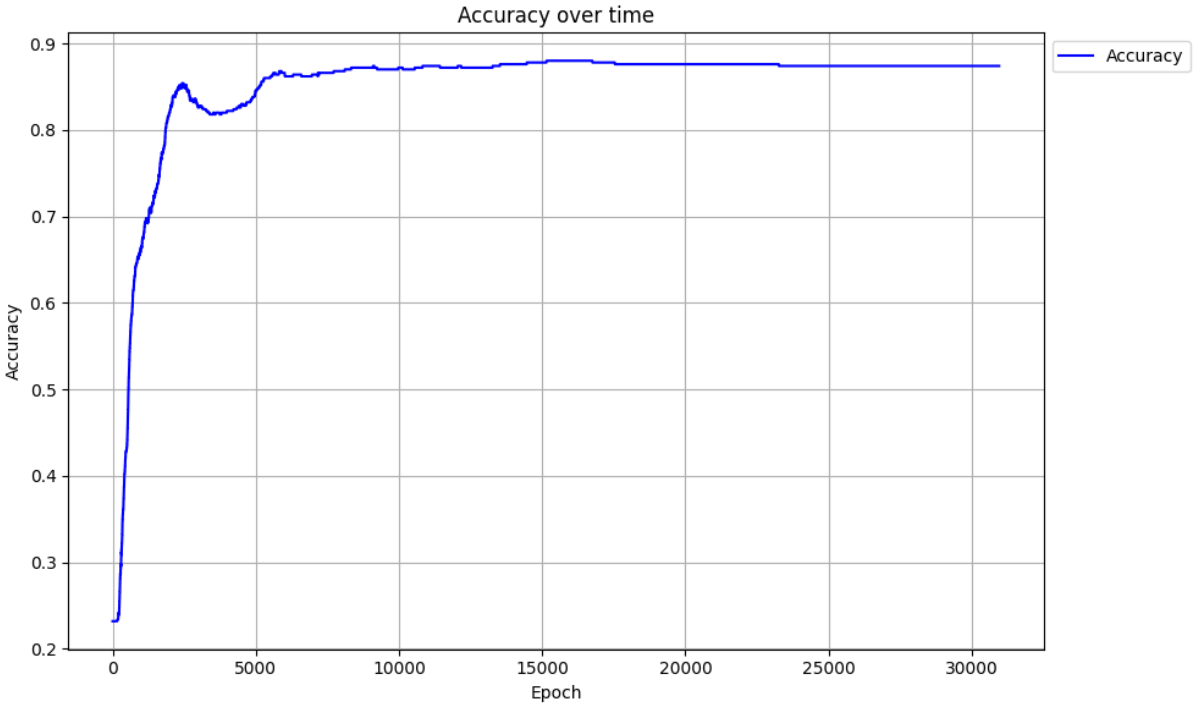
```
# Find out how many classes
classes = len(np.unique(t_multi_train))

# Train the model
multi_class_model = MultiClassFeedForwardNetwork(bias=-1, dim_hidden=8, classes=classes ,with_tol=True, n_epochs_no_update=5)

# Fit the model
train_loss_over_time, val_loss_over_time, accuracy_over_time = multi_class_model.fit(X_train=X_train, t_train=t_multi_train, X_val=X_val, t_val=t_multi

# Print the accuracy of the model
model_accuracy = multi_class_model.accuracy(X=X_val, t=t_multi_val)
print(f"Model is {round(model_accuracy*100, 2)}% accurate on the validation set")
```

```
Epoch 1000:   0.24198091589260962 loss, 66.4% accuracy
Epoch 2000:   0.17363310855570413 loss, 82.0% accuracy
Epoch 3000:   0.13686391699492295 loss, 82.6% accuracy
Epoch 4000:   0.11644126456521317 loss, 82.2% accuracy
Epoch 5000:   0.10458243205933236 loss, 84.6% accuracy
Epoch 6000:   0.09730816369892423 loss, 86.6% accuracy
Epoch 7000:   0.09193501509748546 loss, 86.4% accuracy
Epoch 8000:   0.08839041130047787 loss, 86.8% accuracy
Epoch 9000:   0.08614806343592668 loss, 87.2% accuracy
Epoch 10000:   0.08465300918016128 loss, 87.2% accuracy
Epoch 11000:   0.08359188038550248 loss, 87.4% accuracy
Epoch 12000:   0.08278341175636551 loss, 87.2% accuracy
Epoch 13000:   0.08212535784830992 loss, 87.2% accuracy
Epoch 14000:   0.08156219912086438 loss, 87.6% accuracy
Epoch 15000:   0.08106359029926473 loss, 87.8% accuracy
Epoch 16000:   0.08061263461521725 loss, 88.0% accuracy
Epoch 17000:   0.08020071795688928 loss, 87.8% accuracy
Epoch 18000:   0.07982482683750802 loss, 87.6% accuracy
Epoch 19000:   0.07948494556744709 loss, 87.6% accuracy
Epoch 20000:   0.07918135118123801 loss, 87.6% accuracy
Epoch 21000:   0.07891282396419033 loss, 87.6% accuracy
Epoch 22000:   0.078676251000687 loss, 87.6% accuracy
Epoch 23000:   0.07846716754406256 loss, 87.6% accuracy
Epoch 24000:   0.07828060088786541 loss, 87.4% accuracy
Epoch 25000:   0.07811180632121081 loss, 87.4% accuracy
Epoch 26000:   0.0779567112605494 loss, 87.4% accuracy
Epoch 27000:   0.07781206563659919 loss, 87.4% accuracy
Epoch 28000:   0.0776753978737528 loss, 87.4% accuracy
Epoch 29000:   0.0775448837938771 loss, 87.4% accuracy
Epoch 30000:   0.07741919851158025 loss, 87.4% accuracy
Model is 87.4% accurate on the validation set
```

In [576…
```
# Plot the decision boundary
plot_decision_regions(X_train,t_multi_train, clf=multi_class_model)
```



In [577…
```
# Plot accuracy for the model
plot_accuracy(accuracy_over_time=accuracy_over_time)
```



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 [578…

```
# Linear regression
# This model had the best result when it was scaled, so we scale the data here as well.
linear_cl = NumpyLinRegClass()
X_scaled = scaler(X_train)
t_scaled = scaler(t2_train)
linear_cl.fit(X_scaled, t_scaled, lr=0.03, epochs=1000)

linear_cl_train_precision = linear_cl.precision(linear_cl.predict(X_train),t2_train)
linear_cl_train_recall = linear_cl.recall(linear_cl.predict(X_train),t2_train)
linear_cl_train_accuracy = accuracy(linear_cl.predict(X_train), t2_train)

linear_cl_val_precision = linear_cl.precision(linear_cl.predict(X_val),t2_val)
linear_cl_val_recall = linear_cl.recall(linear_cl.predict(X_val),t2_val)
linear_cl_val_accuracy = accuracy(linear_cl.predict(X_val), t2_val)

linear_cl_test_precision = linear_cl.precision(linear_cl.predict(X_test),t2_test)
linear_cl_test_recall = linear_cl.recall(linear_cl.predict(X_test),t2_test)
linear_cl_test_accuracy = accuracy(linear_cl.predict(X_test), t2_test)

print("[INFO] Linear regression trained and calculated")

# Logistic classifier
logistic_cl = LogisticClassifier()
_ = logistic_cl.fit(X_train,t2_train, X_val, t2_val, eta=0.001, tol=1e-7, epochs=200_000, logging=False)

logistic_cl_train_precision = logistic_cl.precision(logistic_cl.predict(X_train),t2_train)
logistic_cl_train_recall = logistic_cl.recall(logistic_cl.predict(X_train),t2_train)
logistic_cl_train_accuracy = logistic_cl.accuracy(X_train, t2_train)

logistic_cl_val_precision = logistic_cl.precision(logistic_cl.predict(X_val),t2_val)
logistic_cl_val_recall = logistic_cl.recall(logistic_cl.predict(X_val),t2_val)
logistic_cl_val_accuracy = logistic_cl.accuracy(X_val, t2_val)

logistic_cl_test_precision = logistic_cl.precision(logistic_cl.predict(X_test),t2_test)
logistic_cl_test_recall = logistic_cl.recall(logistic_cl.predict(X_test),t2_test)
logistic_cl_test_accuracy = logistic_cl.accuracy(X_test, t2_test)

print("[INFO] Logistic classifier trained and calculated")


# Multi-layer classifier
mlp_cl = MLPBinaryLinRegClass(dim_hidden=9)
_ = mlp_cl.fit(X_train=X_train, t_train=t2_train, X_val=X_val, t_val=t2_val, lr=0.001, epochs=100000, logging=False)

mlp_cl_train_precision = mlp_cl.precision(X_train,t2_train)
mlp_cl_train_recall = mlp_cl.recall(X_train,t2_train)
mlp_cl_train_accuracy = mlp_cl.accuracy(X=X_val, t=t2_val)

mlp_cl_val_precision = mlp_cl.precision(X_val, t2_val)
mlp_cl_val_recall = mlp_cl.recall(X_val,t2_val)
mlp_cl_val_accuracy = mlp_cl.accuracy(X=X_val, t=t2_val)

mlp_cl_test_precision = mlp_cl.precision(X_test,t2_test)
mlp_cl_test_recall = mlp_cl.recall(X_test,t2_test)
mlp_cl_test_accuracy = mlp_cl.accuracy(X=X_test, t=t2_test)

print("[INFO] Multi-layer classifier trained and calculated")


# Printing accuracy info
print("\nAccuracy")
print(f"{'Classifier':<25}{'Training Set':<20}{'Validation Set':<20}{'Test Set':<20}")
print("="*80)
print(f"{'Linear Regression':<25}{linear_cl_train_accuracy:<20.4f}{linear_cl_val_accuracy:<20.4f}{linear_cl_test_accuracy:<20.4f}")
print(f"{'Logistic Regression':<25}{logistic_cl_train_accuracy:<20.4f}{logistic_cl_val_accuracy:<20.4f}{logistic_cl_test_accuracy:<20.4f}")
print(f"{'Multi-Layer Classifier':<25}{mlp_cl_train_accuracy:<20.4f}{mlp_cl_val_accuracy:<20.4f}{mlp_cl_test_accuracy:<20.4f}")

# Print precision info
print("\nPrecision")
print(f"{'Classifier':<25}{'Training Set':<20}{'Validation Set':<20}{'Test Set':<20}")
print("="*80)
print(f"{'Linear Regression':<25}{linear_cl_train_precision:<20.4f}{linear_cl_val_precision:<20.4f}{linear_cl_test_precision:<20.4f}")
print(f"{'Logistic Regression':<25}{logistic_cl_train_precision:<20.4f}{logistic_cl_val_precision:<20.4f}{logistic_cl_test_precision:<20.4f}")
print(f"{'Multi-Layer Classifier':<25}{mlp_cl_train_precision:<20.4f}{mlp_cl_val_precision:<20.4f}{mlp_cl_test_precision:<20.4f}")

# Print recall info
print("\nRecall")
print(f"{'Classifier':<25}{'Training Set':<20}{'Validation Set':<20}{'Test Set':<20}")
print("="*80)
print(f"{'Linear Regression':<25}{linear_cl_train_recall:<20.4f}{linear_cl_val_recall:<20.4f}{linear_cl_test_recall:<20.4f}")
print(f"{'Logistic Regression':<25}{logistic_cl_train_recall:<20.4f}{logistic_cl_val_recall:<20.4f}{logistic_cl_test_recall:<20.4f}")
print(f"{'Multi-Layer Classifier':<25}{mlp_cl_train_recall:<20.4f}{mlp_cl_val_recall:<20.4f}{mlp_cl_test_recall:<20.4f}")
```

```
[INFO] Linear regression trained and calculated
[INFO] Logistic classifier trained and calculated
[INFO] Multi-layer classifier trained and calculated

Accuracy
Classifier               Training Set        Validation Set      Test Set
=============================================================================
Linear Regression        0.6850              0.6540              0.6500
Logistic Regression      0.7200              0.7540              0.7240
Multi-Layer Classifier   0.9360              0.9360              0.9060

Precision
Classifier               Training Set        Validation Set      Test Set
=============================================================================
Linear Regression        0.5658              0.5346              0.5322
Logistic Regression      0.6623              0.6866              0.6595
Multi-Layer Classifier   0.9054              0.9304              0.8990

Recall
Classifier               Training Set        Validation Set      Test Set
=============================================================================
Linear Regression        0.9556              0.9747              0.9239
Logistic Regression      0.6296              0.6970              0.6193
Multi-Layer Classifier   0.9146              0.9383              0.9100
```

The codeblock above will run the algorithms on the hyper tuned parameters. See a sample run in `README.md`. There are clear differences between the algorithms. The Linear Regression model and the Logistic regression models are similar in all three metrics. However, the Multi-Layer Classifier is much better in all three metrics. The metrics indicate that it has learned a more complex pattern of the dataset, and is able to better classify the data. It is able to have high scores on all metrics. All models seems to do worse on precision. Precision shows how much of our predictions was correct based on all correct prediction and all false positive prediction. This could mean that the model may over predict class 1.

Notice also how the Linear regression model hav high recall and low precision. This could mean that model predicts very well class 1 data points, but also make a lot of incorrect predictions of the class 1. This might be bad where false positives are very important in a dataset, and can lead to bad results.

Between the dataset splits there is a small difference in all three metrics. We would be worried if there was a large difference because that could mean that we had a bad split of data. For example, if the dataset for training have features in a "upper region" (or just a different region on the plot), then the training would learn the pattern of this region and not perform well on the other dataset.

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 [579…

```python
# Classes
classes = len(np.unique(t_multi))

# One vs. Rest
one_v_rest_model = OneVRestReg(n=classes)
one_v_rest_model.fit(X=X_train, y=t_multi_train, X_val=X_val, y_val=t_multi_val, eta=0.1, tol=0.00009, epochs=10_000)

one_v_rest_model_train_accuracy = one_v_rest_model.accuracy(x=X_train, t=t_multi_train)
one_v_rest_model_train_precision = one_v_rest_model.precision(x=X_train, t=t_multi_train)
one_v_rest_model_train_recall = one_v_rest_model.recall(x=X_train, t=t_multi_train)

one_v_rest_model_val_accuracy = one_v_rest_model.accuracy(x=X_val, t=t_multi_val)
one_v_rest_model_val_precision = one_v_rest_model.precision(x=X_val, t=t_multi_val)
one_v_rest_model_val_recall = one_v_rest_model.recall(x=X_val, t=t_multi_val)

one_v_rest_model_test_accuracy = one_v_rest_model.accuracy(x=X_test, t=t_multi_test)
one_v_rest_model_test_precision = one_v_rest_model.precision(x=X_test, t=t_multi_test)
one_v_rest_model_test_recall = one_v_rest_model.recall(x=X_test, t=t_multi_test)

print("[INFO] One Vs. Rest trained and calculated")

# Multinomial logistic regression
multinomial_logistic_model = MultinomialLogisticReg(n_classes=classes, bias=-1)
multinomial_logistic_model.fit(X=X_train, y=t_multi_train, X_val=X_val, t_val=t_multi_val, eta=0.001, epochs=200_000)

multinomial_logistic_model_train_accuracy = multinomial_logistic_model.accuracy(X=X_train, t=t_multi_train)
multinomial_logistic_model_train_precision = multinomial_logistic_model.precision(x=X_train, t=t_multi_train)
multinomial_logistic_model_train_recall = multinomial_logistic_model.recall(x=X_train, t=t_multi_train)

multinomial_logistic_model_val_accuracy = multinomial_logistic_model.accuracy(X=X_val, t=t_multi_val)
multinomial_logistic_model_val_precision = multinomial_logistic_model.precision(x=X_val, t=t_multi_val)
multinomial_logistic_model_val_recall = multinomial_logistic_model.recall(x=X_val, t=t_multi_val)

multinomial_logistic_model_test_accuracy = multinomial_logistic_model.accuracy(X=X_test, t=t_multi_test)
multinomial_logistic_model_test_precision = multinomial_logistic_model.precision(x=X_test, t=t_multi_test)
multinomial_logistic_model_test_recall = multinomial_logistic_model.recall(x=X_test, t=t_multi_test)

print("[INFO] Multinomial Logistic Regression trained and calculated")

# Multi-class neural network
multi_class_feed_forward_model = MultiClassFeedForwardNetwork(bias=-1, dim_hidden=8, classes=classes ,with_tol=True, n_epochs_no_update=5)
_ = multi_class_feed_forward_model.fit(X_train=X_train, t_train=t_multi_train, X_val=X_val, t_val=t_multi_val, lr=1e-4, tol=1e-7, epochs=500_000, logg


multi_class_feed_forward_model_train_accuracy = multi_class_feed_forward_model.accuracy(X=X_train, t=t_multi_train)
multi_class_feed_forward_model_train_precision = multi_class_feed_forward_model.precision(x=X_train, t=t_multi_train)
multi_class_feed_forward_model_train_recall = multi_class_feed_forward_model.recall(x=X_train, t=t_multi_train)

multi_class_feed_forward_model_val_accuracy = multi_class_feed_forward_model.accuracy(X=X_val, t=t_multi_val)
multi_class_feed_forward_model_val_precision = multi_class_feed_forward_model.precision(x=X_val, t=t_multi_val)
multi_class_feed_forward_model_val_recall = multi_class_feed_forward_model.recall(x=X_val, t=t_multi_val)

multi_class_feed_forward_model_test_accuracy = multi_class_feed_forward_model.accuracy(X=X_test, t=t_multi_test)
multi_class_feed_forward_model_test_precision = multi_class_feed_forward_model.precision(x=X_test, t=t_multi_test)
multi_class_feed_forward_model_test_recall = multi_class_feed_forward_model.recall(x=X_test, t=t_multi_test)

print("[INFO] Multi-class feed forward neural network trained and calculated")

# Accuracy
```

```
print("\nAccuracy")
print(f"{'Classifier':<35}{'Training Set':<20}{'Validation Set':<20}{'Test Set':<20}")
print("="*100)
print(f"{'One-vs-Rest Classifier':<35}{one_v_rest_model_train_accuracy:<20.4f}{one_v_rest_model_val_accuracy:<20.4f}{one_v_rest_model_test_accuracy:<2
print(f"{'Multinomial Logistic Regression':<35}{multinomial_logistic_model_train_accuracy:<20.4f}{multinomial_logistic_model_val_accuracy:<20.4f}{multi
print(f"{'Multi-Class Neural Network':<35}{multi_class_feed_forward_model_train_accuracy:<20.4f}{multi_class_feed_forward_model_val_accuracy:<20.4f}{m

# Precision
print("\nPrecision")
print(f"{'Classifier':<35}{'Training Set':<20}{'Validation Set':<20}{'Test Set':<20}")
print("="*100)
print(f"{'One-vs-Rest Classifier':<35}{one_v_rest_model_train_precision:<20.4f}{one_v_rest_model_val_precision:<20.4f}{one_v_rest_model_test_precision
print(f"{'Multinomial Logistic Regression':<35}{multinomial_logistic_model_train_precision:<20.4f}{multinomial_logistic_model_val_precision:<20.4f}{mu
print(f"{'Multi-Class Neural Network':<35}{multi_class_feed_forward_model_train_precision:<20.4f}{multi_class_feed_forward_model_val_precision:<20.4f}

# Recall
print("\nRecall")
print(f"{'Classifier':<35}{'Training Set':<20}{'Validation Set':<20}{'Test Set':<20}")
print("="*100)
print(f"{'One-vs-Rest Classifier':<35}{one_v_rest_model_train_recall:<20.4f}{one_v_rest_model_val_recall:<20.4f}{one_v_rest_model_test_recall:<20.4f}"
print(f"{'Multinomial Logistic Regression':<35}{multinomial_logistic_model_train_recall:<20.4f}{multinomial_logistic_model_val_recall:<20.4f}{multinom
print(f"{'Multi-Class Neural Network':<35}{multi_class_feed_forward_model_train_recall:<20.4f}{multi_class_feed_forward_model_val_recall:<20.4f}{multi
```

```
[INFO] One Vs. Rest trained and calculated
[INFO] Multinomial Logistic Regression trained and calculated
[INFO] Multi-class feed forward neural network trained and calculated

Accuracy
Classifier                          Training Set        Validation Set      Test Set
====================================================================================================
One-vs-Rest Classifier              0.7830              0.8260              0.7680
Multinomial Logistic Regression     0.6710              0.7380              0.6600
Multi-Class Neural Network          0.8640              0.8820              0.8380

Precision
Classifier                          Training Set        Validation Set      Test Set
====================================================================================================
One-vs-Rest Classifier              0.7830              0.8260              0.7680
Multinomial Logistic Regression     0.7430              0.8049              0.7319
Multi-Class Neural Network          0.8665              0.8792              0.8407

Recall
Classifier                          Training Set        Validation Set      Test Set
====================================================================================================
One-vs-Rest Classifier              0.7830              0.8260              0.7680
Multinomial Logistic Regression     0.6676              0.7240              0.6794
Multi-Class Neural Network          0.8626              0.8797              0.8484
```

The code block above does again train all models and print information in a formatted table. See a sample run in `README.md`. The models performed very well. The best model was the Multi-class Neural Network. I think this is also because the network was able to learn a more complicated structure of the data, and therefor become more accurate. One vs. Rest and the Multinomial logistic classifier has similar result. One vs. Rest where better than Multinomial logistic regression on all metrics. Again there are some deviation between the datasets, but not much.