TBMI26 – Computer Assignment Report Supervised Learning

Deadline – March 14 2022

Author/-s:

Chayan Shrang Raj (chash345)
Theodor Emanuelsson (theem089)

In order to pass the assignment you will need to answer the following questions and upload the document to LISAM. Please upload the document in PDF format. **You will also need to upload all code in .m-file format**. We will correct the reports continuously so feel free to send them as soon as possible. If you meet the deadline you will have the lab part of the course reported in LADOK together with the exam. If not, you'll get the lab part reported during the re-exam period.

1. **Give an overview of the four datasets from a machine learning perspective. Consider if you need linear or non-linear classifiers etc.**

All datasets include one array of predictors, one array of labels and one array of desired outputs.

The first dataset contains two features, two classes (1 and 2) and 2000 observations. If the data is plotted it is possible to see that the two classes are linearly separable based on the two features. The dataset is very well suited for a simple model and using multi-layered neural networks is unnecessarily complex. The number of labels from each class is identical.

The second dataset contains two features, two classes (1 and 2) and 2000 observations. If the data is plotted it is possible to see that the two classes are not linearly separable based on the two features, so the problem requires a non-linear classifier to be solved with high accuracy. The number of labels from each class is identical.

The third dataset contains two features, three classes (1, 2 and 3) and 2000 observations. If the data is plotted it is possible to see that the three classes are not linearly separable based on the two features, so we require a non-linear classifier to solve the problem with high accuracy. The number of labels from each class is identical.

The fourth dataset contains digits in the form 8x8 arrays which have then been flattened into 64 features, with classes from 1 to 10 (representing digits from 0-9) and 5620 observations. Due to the complexity of the problem, it can be assumed that we require a non-linear decision boundary. The features are integer values between 0 and 16, representing the grayscale of the pixel value. The number of observations in this dataset does on one hand seem large but that is in fact only due to the preprocessing done. However, one could argue that, given the training data only comes from 30 people and that the test data only comes from 13, that we cannot be certain how well it generalizes to a larger population. The number of labels from each class is almost equally distributed.

2. **Explain why the down sampling of the OCR data (done as pre-processing) result in a more robust feature representation. See http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits**

The image set was originally collected from 43 people and the original images was converted from 32x32 bitmaps and this was divided into 4x4 nonoverlapping blocks. The blocks include the count of the number of black (on) pixels. This reduced the dimensionality of the data into a 8x8 image, where each cell represents the level of grayscale (between 0 and 16). The pre-processing done reduces the number of features by one fourth and accounts for the inconsistency in how and where on the image the people write the digits. In classifying images, without using convolution, it is a typical issue that what we are trying to classify could be positioned differently and this creates issue for identifying the common pattern in the image using regular neural networks.

3. **Give a short summary of how you implemented the kNN algorithm.**

In this case we used kNN for classification. It is a non-parametric classification algorithm so to make new predictions, we need to have the data available. First, we calculate the distances from a given input to all the training points. Then the distances are sorted in ascending order and, for a user given hyperparameter *k,* we select the *k* nearest training points. The prediction is then done using a majority vote scheme from the *k* nearest training point labels. We include some checks for ties in distances and in the majority vote. If we find that there is a tie in the sorted list of distances between k and k+1 we run a helper function built to check the sorted list for more ties. We increase k to the number of found ties in distance. We perform the the majority vote scheme in a helper function. There we retrieve the unique labels and their count given the list of *k* labels. If there is only one label, we return it directly to the kNN function. If there is more than one label, we compute the maximum and check if there are any other counts with the same value. If not, we return the label. Else we run the majority vote scheme again with k + 1. This solution does increase the run time of the algorithm since we check many extra conditions.

4. **Explain how you handle draws in kNN, e.g. with two classes (k = 2)?**

There are two possible types of draws that should be considered in kNN. The first one is when the distances between the *k* and *k+1* neighbor is the same to the training point. If this is the case, we increment *k* by one and check if there are any more ties in distance measures. When we do not find any more ties in distance, we increment *k* by the number of ties. This is not a common issue with using the algorithm as two training points with the same features value are unlikely to have different labels. I could however tip the balance in a majority vote.

The second kind of draw is a tie in the majority vote. If a situation arises where the *k* number of nearest labels are the same for the two classes, we again increment *k* by one. This is a good strategy because we prevent random assignment to a class in addition to making the model less complex. The model is less complex in the sense that the prediction utilizes more information from more training points. We argue that this is a better strategy than decreasing *k* by one, since this prevents the decision boundary from becoming “dirty” in case of outliers.

5. **Explain how you selected the best k for each dataset using cross validation. Include the accuracy and images of your results for each dataset.**

We perform 10-fold cross-validation for hyperparameters k = 1, 2, … , 10 for the three first dataset. Since the fourth dataset is considerably larger we use 5-fold cross-validation for k = 1, 2, … , 10.

For the implementation of cross-validation, we start by shuffling the training data, in case there is some particular structure to how the data was entered. We then split the array of training data and of training labels into different small sections, where the number of sections depends on the number of folds. If the split is not done evenly, then the last array will receive the extra observations. We then iterate through the number of folds, selecting one dataset to be the in-batch validation data and the rest to be the in-batch training data. We then run the k-NN algorithm using the in-batch training data and compute the accuracy for the in-batch validation data. We store the accuracy for each fold in a list and return the mean accuracy.

Dataset 1:

Cross-validation gave us the best mean accuracy of the 10 folds when k = 7. The overall accuracy achieved on the test data was then 99.5 %, with only 2 out of 400 testing points becoming misclassified. We use 80% of the data for training and the remaining 20% for testing. Since the decision boundary seems to be simple, this might be unnecessary as we can achieve similar performance using substantially less training data. However, for a well performing model, one should always try to use as much of the available data as possible, while keeping a reasonable amount for testing. Therefore, we present our results based on the modeling utilizing a substantial portion of the data.

Graphs of accuracy: Dataset 1

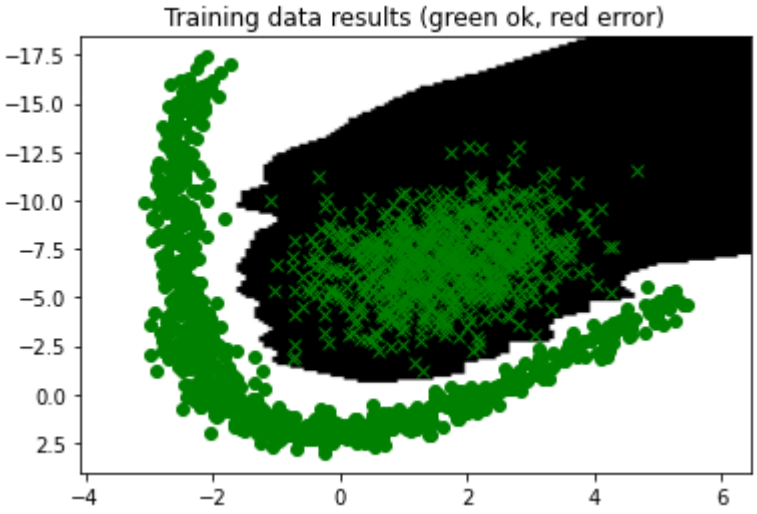
Confusion matrix for test data: Dataset 1

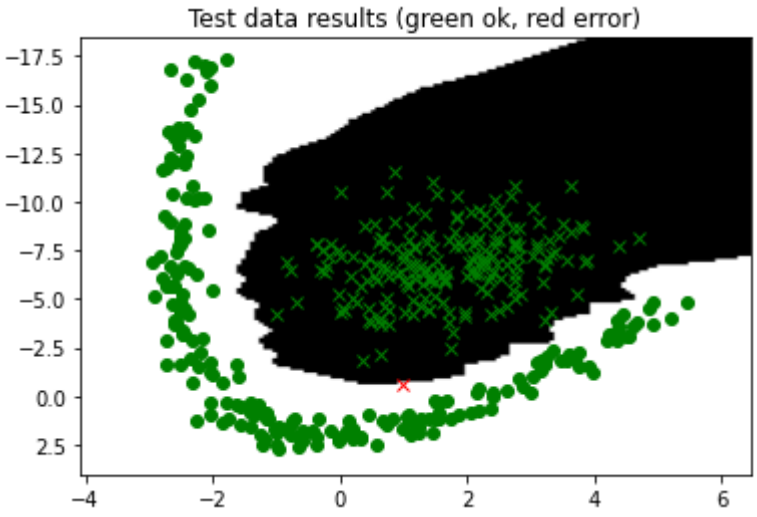
```
Confusion matrix:
[[199.   1.]
 [  1. 199.]]
Accuracy: 0.9950
```

Dataset 2:

Cross-validation gave us the best mean accuracy of the 10 folds when k = X. The overall accuracy achieved on the test data was then 99.75%, meaning that the model only misclassified one of the training points. Again, we use 80% of the data for training and 20% for testing.

Graphs of accuracy: Dataset 2





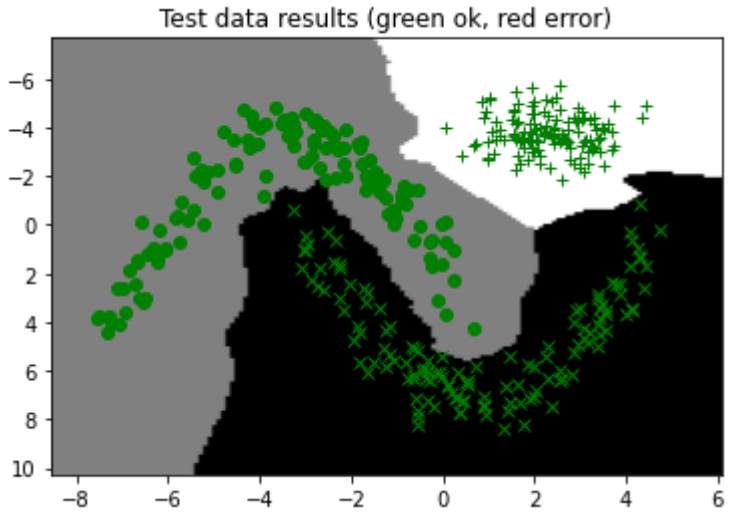
Confusion matrix for test data: Dataset 2

```
Confusion matrix:
[[199.   0.]
 [  1. 200.]]
Accuracy: 0.9975
```

Dataset 3:

Cross-validation gave us the best mean accuracy of the 10 folds when k = 1. The overall accuracy achieved on the test data was then 100%. Yet again, we use 80% of the data for training and 20% for testing.

Graphs of accuracy: Dataset 3



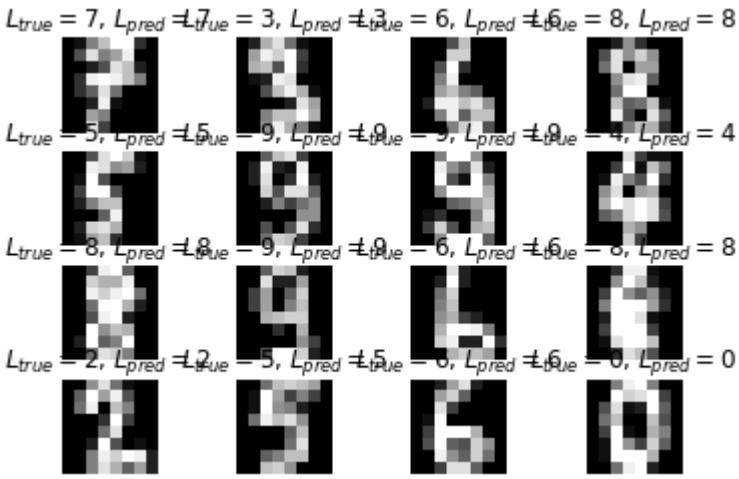
Confusion matrix for test data: Dataset 3

```
Confusion matrix:
[[133.   0.   0.]
 [  0. 133.   0.]
 [  0.   0. 133.]]
Accuracy: 1.0000
```

Dataset 4:

Since this dataset is larger and we noticed experimentally that it contains a lot of ties, we use fewer cross-validation folds. This is done mostly because cross-validation takes some time if we use 5 folds. For this dataset we notice that there are many ties in both distance and in the majority vote. Having a proper strategy on how to handle these sorts of ties is therefore important for this problem. If the ties are decided upon randomly, many border case decisions would become random and rerunning the model could produce significant difference in performance. The 5-fold cross-validation yielded a k = 1. The final model achieves an accuracy of 97.77% with most misclassifications being for 8, where the model on a few occasions predicts 1.

Graphs of accuracy: Dataset 4



Confusion matrix for test data: Dataset 4

```
Confusion matrix:
[[220.   0.   0.   0.   0.   0.   0.   0.   0.   1.]
 [  0. 217.   0.   1.   0.   0.   0.   0.  11.   1.]
 [  0.   0. 220.   0.   0.   0.   0.   0.   0.   1.]
 [  0.   0.   0. 213.   0.   0.   0.   0.   2.   1.]
 [  0.   0.   0.   0. 220.   0.   0.   0.   0.   3.]
 [  0.   0.   0.   1.   0. 214.   0.   0.   0.   2.]
 [  0.   1.   0.   0.   0.   2. 220.   0.   2.   0.]
 [  0.   2.   0.   1.   0.   0.   0. 218.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0.   0. 203.   5.]
 [  0.   0.   0.   4.   0.   4.   0.   2.   2. 206.]]
Accuracy: 0.9777
```

6. **Give a short summary of your backprop implementations (single + multi). You do not need to derive the update rules.**

The algorithms are given some training data, desired training outcomes, test data, desired test labels, an initial weight matrix (or for single layer, two for multilayer), the number of iterations to run and the learning rate.

First some initial predictions are made based on the training data and initial weight matrix. These are then used to calculate the mean squared error between the predictions and the actual training labels. We can call the mean square error our cost function. It is also computed for the test data, but only so that we can track the accuracy of the test data for the iterations.

For each iteration we calculate the gradient of the cost function. In the single layer network, it is a rather simple expression. We have only one layer, so the gradient is only the derivative of cost function and averaged over the whole training set which is then multiplied by the learning rate to take a step and finally subtracted by old weights to give us new weights. This is repeated until number of iterations are completed or if the tolerance level is reached (optional parameter).

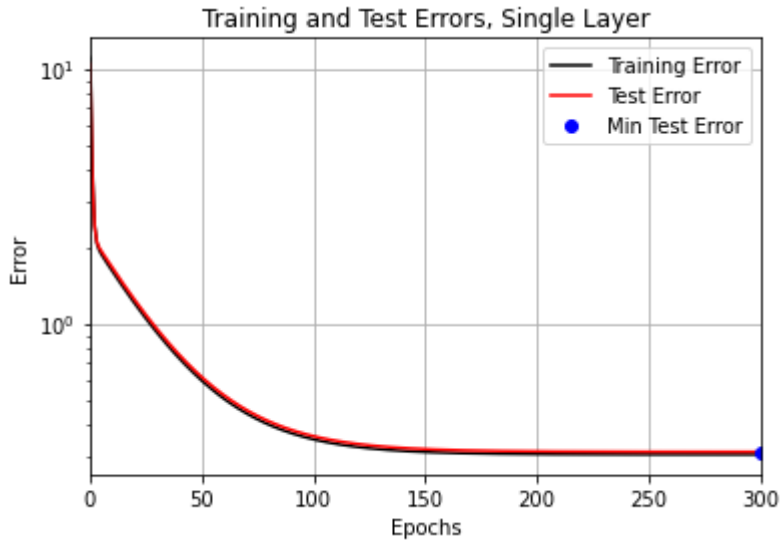
Multi-Layer Network -
For multi-layer network, performing back propagation can be tricky as it involves weight matrices of different sizes for different layers. There are two directions in this case known as feedforward network and backpropagation network. For feedforward network, we simply calculate the forward direction of the model, where initialized weights are combined with desired number of neurons and layers, and then we compare it with our true predictions and calculate train and test error. So, imagine, this has been done for the first iteration, but it is not over yet. Now, we perform partial derivatives from the output layer to the hidden layer(s)with respect to the weights and biases exchanged in those two layers. Now, hidden layers can be many, and so we traverse back from one layer to another, generating arrays of partial derivatives and updating the new weights. This is done till we reach our input layer and from there, we have our new updated weights and biases. This completes our first iteration. Now, this is done till our number of iterations are exhausted and parameters are keep getting updated to reduce the error between true classes and predicted classes. Note, it is better to use nonlinear activation functions in hidden layers, so that model can learn more complex patterns but that equally demands calculating partial derivatives with respect to different functions and between different layers.

7. **Present the results from the neural network training and how you reached the accuracy criteria for each dataset. Motivate your choice of network for each dataset. Explain how you selected good values for the learning rate, iterations and number of hidden neurons. Include images of your best result for each dataset, including parameters etc.**

Dataset 1:

For this problem the classes are linearly separable, so a single layer neural network is enough. We used 300 iterations and a learning rate is set to 0.01. We used a rather high learning rate in order to avoid having the model stuck at a local minimum. In general, we experimented with a couple of different combinations of iterations and learning rate. We initialized the weights randomly by drawing from a uniform distribution [0,1). Here we are using 50% of the data for training and 50% of the data for testing. We did this training and testing split because the decision boundary is so simple and having more testing data gives extra confidence in how well the model generalizes to new unseen data.

Graph of learning rate: Dataset 1



Graphs of accuracy: Dataset 1

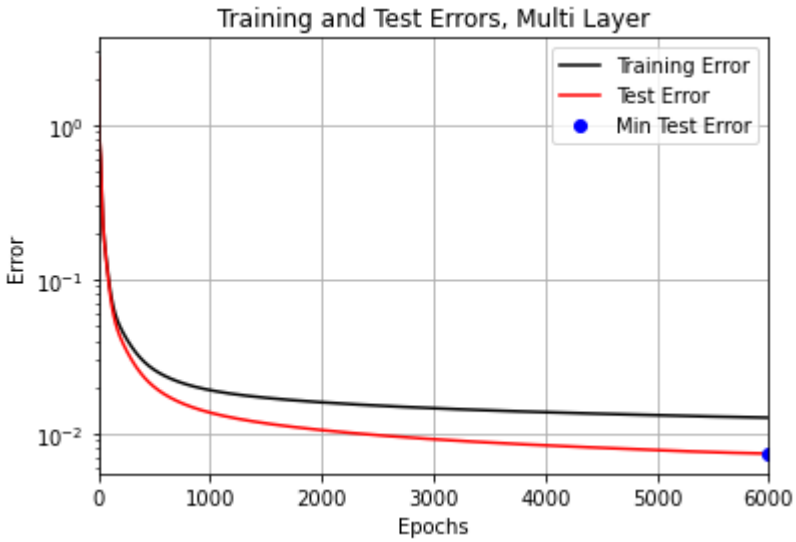
Confusion matrix for test data: Dataset 1

```
Confusion matrix:
[[498.   6.]
 [  2. 494.]]
Accuracy: 0.9920
```

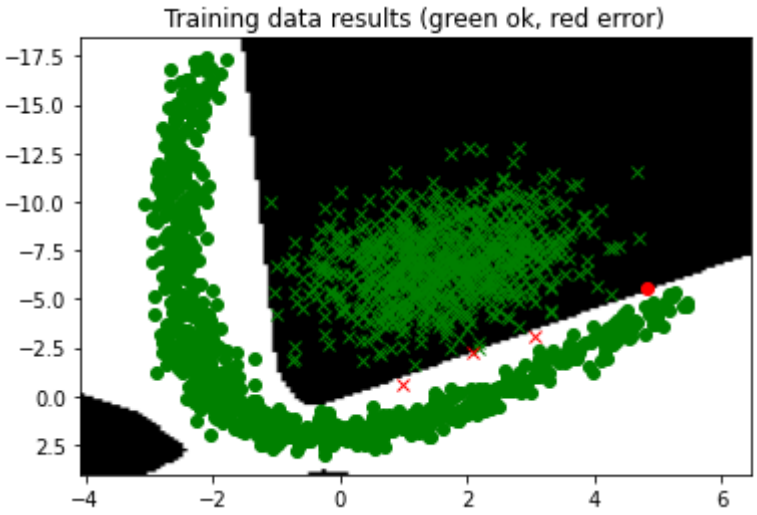
Dataset 2:

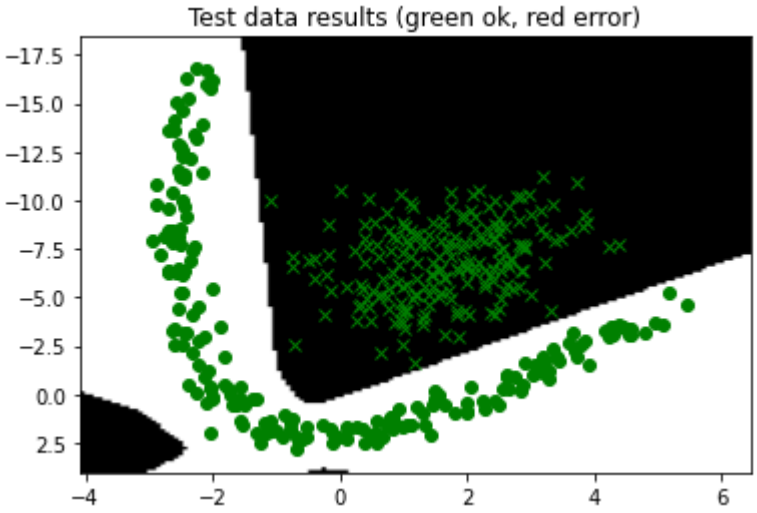
For this dataset, we are using 80% of the data for training and 20 % for testing. In this case, as we see from the data plot, the classes are not linearly separable, and thus it is better to choose multi-layer neural network where we introduce nonlinear activation functions. For learning rate, we choose 0.05 as the good rate since using a lower learning rate might cause the algorithm to get stuck in a local minimum and not approach desired optimization. We tried a couple of combinations of iterations, learning rate and number of hidden units. We finally achieved a well performing model using 6000 iterations and 4 hidden units. It has 100% accuracy on the test data while having a few misclassifications on the training data.

Graph of learning rate: Dataset 2



Graphs of accuracy: Dataset 2





Confusion matrix for test data: Dataset 2

```
Confusion matrix:
[[200.   0.]
 [  0. 200.]]
Accuracy: 1.0000
```

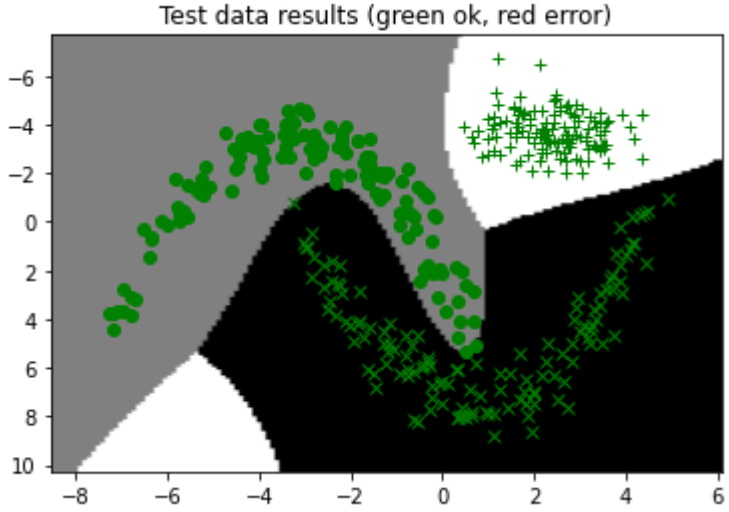
Dataset 3:

This dataset is a bit more complicated as it has three classes. We experimented with combinations of hidden units, learning rate and epochs for a multilayer model. We tried to increase the learning rate as much as possible while still achieving good performance and a reasonable training error curve. We were able to reduce the learning rate to 0.02 for 7000 epochs while having 7 hidden units. As we increased to number of hidden units, it was possible to see how the decision boundary became better. Our final model achieves 100% accuracy on the testing data while having a few misclassifications on the training data. We used a roughly 80-20% training-testing split.

Here we used the Xavier initialization for the weight matrix, which could account for why the learning rate can be increased so much. That is, drawing from a normal distribution with mean = 0 and variance = 1 / number of hidden units. For the bias we initialize the weights to 0. Using this strategy, we can avoid the vanishing gradient problem by having the mean of the initial weights close to zero and the variance low.

Graph of learning rate: Dataset 3

Graphs of accuracy: Dataset 3



Confusion matrix for test data: Dataset 3

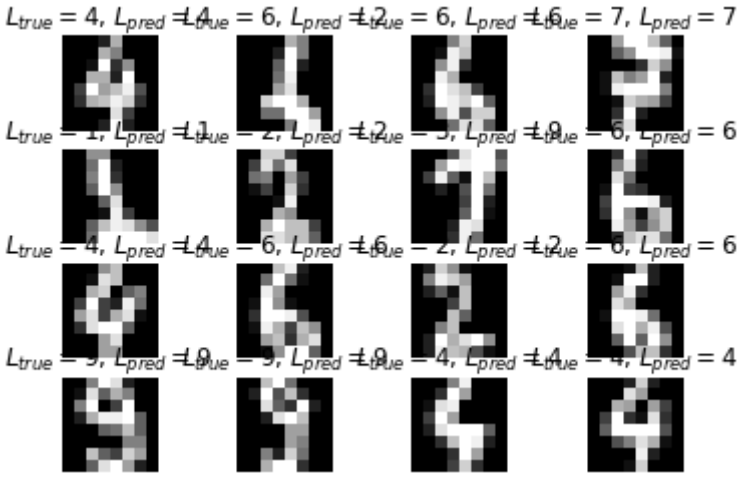
```
Confusion matrix:
[[133.   0.   0.]
 [  0. 133.   0.]
 [  0.   0. 133.]]
Accuracy: 1.0000
```

Dataset 4:

This dataset is complex for classification as we have 10 classes. Here use 24 neurons, a learning rate of 0.02 and 8000 epochs. Again, we had to use the Xavier initialization method to get a good model. Since we are using 24 neurons, the vanishing gradient could be an issue if weights are not initialized in a considered manner. We experimented a bit with the model, trying to increase learning rate and decrease the number of hidden units. In the plot below we can see that the model is fairly unstable in terms of error for the first 1000 epochs and then starts to stabilize. The difference between 5000 and 8000 epochs is not very different and one could perhaps use fewer epochs and still achieve a good result.

Graph of learning rate: Dataset 4

Graph of accuracy: Dataset 4



Confusion matrix for test data: Dataset 4

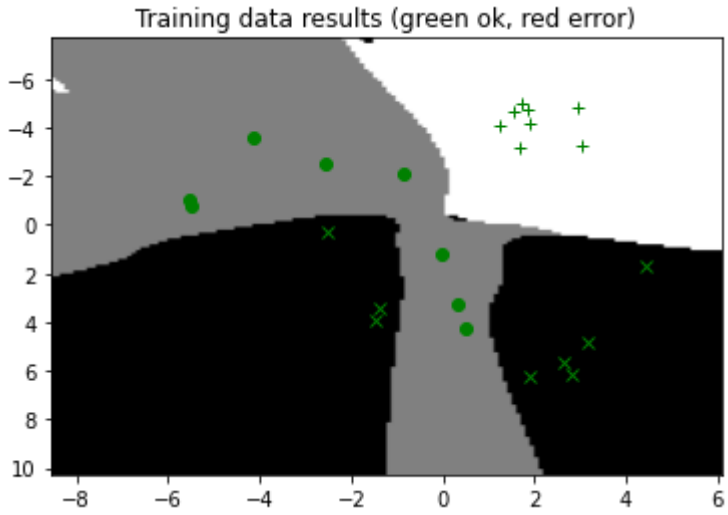
```
Confusion matrix:
[[110.   0.   0.   0.   0.   0.   0.   0.   0.   0.]
 [  0. 109.   0.   0.   0.   0.   1.   0.   4.   0.]
 [  0.   0. 110.   0.   0.   0.   1.   0.   0.   0.]
 [  0.   0.   0. 106.   0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0. 110.   0.   0.   0.   0.   1.]
 [  0.   0.   0.   0.   0. 110.   0.   1.   0.   0.]
 [  0.   0.   0.   0.   0.   0. 108.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.   0. 107.   0.   0.]
 [  0.   0.   0.   2.   0.   0.   0.   1. 106.   0.]
 [  0.   1.   0.   2.   0.   0.   0.   1.   0. 109.]]
Accuracy: 0.9864
```

8. **Present the results, including images, of your example of a non-generalizable backprop solution. Explain why this example is non-generalizable.**

To create a non-generalizable neural network, we used very few datapoints for training, specifically 24. We also increased the number of hidden units to 200, in order to create highly non-linear decision boundaries. We ran the model for 4000 iterations in order to get the training accuracy high. The model has 100% accuracy on the training data while having 82.9% accuracy on the test data. The model was still able to accurately predict the small dense cluster of one class visible in the upper right corner of the below plots. However, the model does not generalize well to new unseen data as it fails to generate decision boundaries that can well separate the classes based on the features. One could argue that the example is non-generalizable due to multiple things. For one, having training data as small as this, we cannot assume the data to be a valid representation of the complete data even though it is randomly sampled. In addition, since we are training for a high accuracy on the small training data and making the model complex, the decision boundaries will adapt to separate the training data.

Graph of learning rate: non-generalizable example

Graph of accuracy: non-generalizable example



Confusion matrix on test data: non-generalizable example

```
Confusion matrix:
[[391.  81.   0.]
 [210. 550.   1.]
 [ 31.   1. 631.]]
Accuracy: 0.8291
```

9. **Give a final discussion and conclusion where you explain the differences between the performances of the different classifiers. Pros and cons etc.**

Generally, we saw good results for the k-NN algorithm on all the datasets. It is a very simple model to understand and explain. It can support both linear and non-linear separable classes for classification and only has one hyperparameter that needs tuning. It does, however, have some drawbacks. For one, as a non-parametric model, having to always have the training data available makes it a poor choice for situations where predictions need to be made continuously or when the data is very large. It also tends to create messy decision boundaries, especially when k is low, that can be troublesome for border case decisions.

The single layer neural network is fast to train and can be somewhat understood mathematically from looking at the weight matrix. It does, however, only support linear decision boundaries which makes it unsuitable for many applications.

The multilayer neural network supports many kinds of decision boundaries and is generally a well performing model. It is parametric so it can be used in situations where prediction needs to be done continuously. It does, however, take some time to train and has quite a few hyperparameters that need tuning. In addition, the initialization of the weights makes a big difference for the results and the vanishing gradient problem makes having very deep network problematic.

In conclusion, the k-NN model is very adaptive and can solve many kinds of problems with only limited hyperparameter tuning. It is an impressive model that should not be underestimated. Our implementation that checks for ties makes it computationally heavy when data is large. Perhaps if the code was optimized better or was run in a lower-level language it would be faster than the multilayer neural network. Neural networks are on the other hand parametric, which gives some versatility, and our multilayer implementation is faster than k-NN implementation.

10. **Do you think there is something that can improve the results? Pre-processing, algorithm-wise etc.**

When we talk about implementing k-NN model, firstly it is desirable to have all the data points normalized which definitely helps in calculating distance in the same units and gives better result but we already have normalized data, so we have got good results from k-NN. Secondly, as implemented, we can have tie breakers as explained above which allows a logical process rather than random tir break, that ultimately provides good performance.

While, talking about single layer, since we do not have a choice of non linear activation function, we can still choose by trying different learning rates that may play a decent role in getting good accuracy. Also, weight initialization is important in order to get good accuracy from the model since there is a possibility of gradient becoming zero if weights are irregularly chosen. We can use methods like Adaptive weight initialization, Xavier weight initialization etc. to initialize weights.

For multi-layer model, there are several parameters that can be tweaked to improve the results. We can have various different non-linear activation functions to enhance model's capability to capture more complex pattern in the data. We can initialize weights differently like Xavier's weight initialization etc. That significantly affects the results. Also, we can use different optimizers like Stochastic gradient descent, mini-batch SGD, Adam, RMSPROP, etc. They may come at a cost of computation or parameter tuning but are very efficient in reaching a good accuracy.

11. **Optional task (but very recommended). Simple gradient decent like what you have implemented can work well, but in most cases we can improve the weight update by using more sophisticated algorithms. Some of the most common optimization algorithms are summarized nicely here: https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6 Implement one or a few different optimizers and compare the speed at which the training converges. A good starting point is to implement momentum gradient decent.**

It was a fun activity. We have implemented mini-batch stochastic gradient descent and momentum gradient descent. By comparing different optimizers, we see that momentum gradient descent was able to converge the fastest and gave good results. The only challenge we faced was with tuning hyperparameters since now even the slightest changes were producing erratic results. But, of course, faster training outweighs prarameter tuning, any day.