Introduction to Machine Learning
Assignment 5

Group 31
Stijn Kammer (s4986296) & Ramon Kits (s5440769)

October 20, 2022

Introduction

Learning Vector Quantization (LVQ) is a prototype-based supervised classification algorithm. LVQ can be trained on a dataset with labels and then used to classify new data without labels. It can be compared to Vector Quantization (VQ), which is an unsupervised algorithm that is used for data compression and finding clusters in data. LVQ works by defining a set of prototypes and then moving them to better represent the data. The prototypes are moved in the direction of the input vector if the input vector belongs to the class of the prototype, and in the opposite direction if it does not. After several iterations, the prototypes should have moved to a position where they represent the data well. The prototypes are then used to classify new data. There are many different variations of LVQ, but the most common and easy to understand is LVQ1.

Methods

LVQ1

For this assignment, the LVQ algorithm has been implemented in Python. For the majority of the assignment, the standard 2-dimensional LVQ dataset has been used which was delivered by the assignment.
The LVQ1 algorithm has been implemented in the following steps:

1. The dataset is loaded and split into 2 classes. The first 50 samples are class 0 and the last 50 samples are class 1.

Listing 1: *dividing classes*

```python
def divideIntoClasses(data, numClasses, dim):
    """Divide data into classes by adding column with class number.
    """
    data = np.insert(data, dim, 0, axis=1)
    for i in range(numClasses):
        data[i*(data.shape[0]//numClasses):(i+1)*(data.shape[0]//
    numClasses), dim] = i
    return data
```

2. The prototypes are initialized. The prototypes are initialized randomly within the range of the dataset.

Listing 2: *initializing prototypes*

```python
def initPrototypes(data, numExamples, numPrototypes, dim, random=True, initAtClassMean=False):
    """
    Initialize prototypes.
    Requires labeled data.
    """
    # sort data by class
    dataSorted = data[data[:,dim].argsort()]
    # get random data point(s) for each class
    prototypes = np.zeros((numPrototypes, dim+1))
    if initAtClassMean:
        for i in range(numPrototypes):
            prototypes[i] = np.append(np.mean(dataSorted[i*(numExamples//numPrototypes):(i+1)*(numExamples//numPrototypes), :-1], axis=0), dataSorted[i*(numExamples//numPrototypes)][-1])
    else:
        if random:
            for i in range(numPrototypes):
                prototypes[i] = dataSorted[np.random.randint(i*(numExamples//numPrototypes), (i+1)*(numExamples//numPrototypes)), :dim+1]
        else:
            for i in range(numPrototypes):
                prototypes[i] = dataSorted[i*(numExamples//numPrototypes), :dim+1]
    return prototypes
```

First the data is sorted by class if it not already is. Then the prototypes are initialized by randomly selecting a sample from a subset of a class. Subsets are equally divided over the prototypes of that class. This makes it less likely that the prototypes are initialized in the same area. This code also supports prototypes to be assigned to the mean vector of a class.

3. Start the first epoch, at the start of each epoch the prototypes are shuffled.

4. Loop through all samples in the dataset.

5. Get the closest prototype to the sample. In this case, the list of prototypes is sorted by distance to the sample.

Listing 3: *getting the closest prototype*

```python
    prototypesSorted = sorted(prototypes, key=lambda x: eaclidianDistance(point, x))
```

The first prototype in the list is the closest so it is returned.

6. If the sample belongs to the class of the prototype, move the prototype in the direction of the sample. If the sample does not belong to the class of the prototype, move the prototype in the opposite direction of the sample.

Listing 4: *moving the prototype*

```
            # update closest prototype
            changeBy = LR * (data[i][:-1] - closestPrototype[:-1])
            if data[i][-1] == closestPrototype[-1]:
                closestPrototype[:-1] += changeBy
            else:
                closestPrototype[:-1] -= changeBy
```

7. If the sample is the last in the dataset, start the next epoch.

8. When the maximum number of epochs is reached or the Quantization Error is roughly stable, stop the algorithm.

Listing 5: *stopping the algorithm*

```
    # if error moving average seems stable, stop learning
    if stopWhenStable and len(trainingErrors) >= stableMovingAverage*2:
        # get mean of last stableMovingAverage moving averages
        movingAverageMean = np.mean([trainingErrors[i][2] for i in range(len(trainingErrors)-stableMovingAverage, len(trainingErrors))])
        # check if all moving averages are within 0.01 of the mean
        differences = []
        for movingAverage in trainingErrors[-stableMovingAverage:]:
            differences.append(abs(movingAverage[2] - movingAverageMean))
        if max(differences) < 0.001:
            # stop learning
            return prototypes, trainingErrors, prototypePositionHistory
```

The above code shows the stopping criteria. The Quantization Error is calculated by summing the wrong classifications and dividing that by the total number of samples. The Quantization Error is calculated at the end of each epoch. From errors, a moving average is calculated. The moving average is used to determine if the Quantization Error is roughly stable. When the moving average has not changed more than 1% in the last x epochs, the algorithm is stopped.

GLVQ

For the GLVQ algorithm, the same steps as for the LVQ1 algorithm have been taken. The only difference is that when moving the prototypes, not only the closest prototype is used, but also the second closest prototype from a different class. The prototypes are moved in the direction of the sample if the sample belongs to the class of the closest prototype, and in the opposite direction if it does not.

Listing 6: *moving the prototype*

```
            if method.lower() == 'glvq':
                # get closest prototype and closest prototype of different class
                closestSamePrototype, closestDifferentClass, closestPrototype = getClosestPrototype(data[i], prototypes, findClosestSameAndDifferent=True)
                # update closest prototype of same class
                closestSamePrototype[:-1] += LR * (data[i][:-1] - closestSamePrototype[:-1])
                # update closest prototype of different class
                closestDifferentClass[:-1] -= LR * (data[i][:-1] - closestDifferentClass[:-1])
```

Above code shows this behavior. The amount the prototypes are moved is determined by the learning rate and the distance between the sample and the prototype. When the GLVQ algorithm is used, the learinging rate decreases over time. Every epoch, the learning rate is lowered by 10% as shown in the code below.

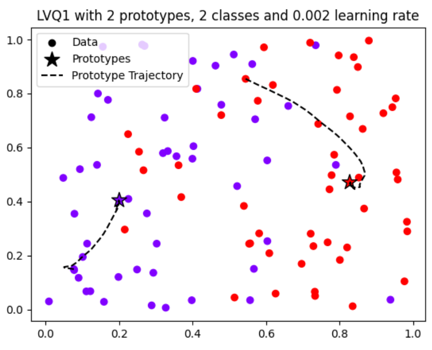
Listing 7: *lowering the learning rate*

```
        # if method is glvq, decrease LR 10%
        if method.lower() == 'glvq':
            LR *= 0.9
```

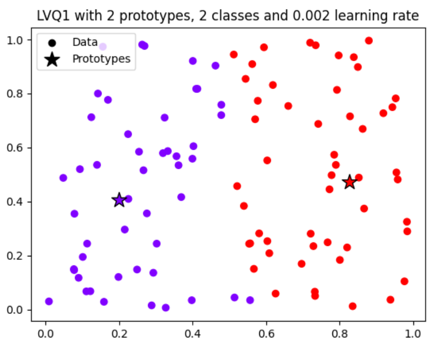
Iris dataset

The code for LVQ1 has been written with multidimensional datasets in mind. The Iris dataset has 4-dimensions, so the code can be used for this dataset as well. Only the dataset had to be changed. The dataset had labeled samples, the labels which were strings had to be converted to integers.

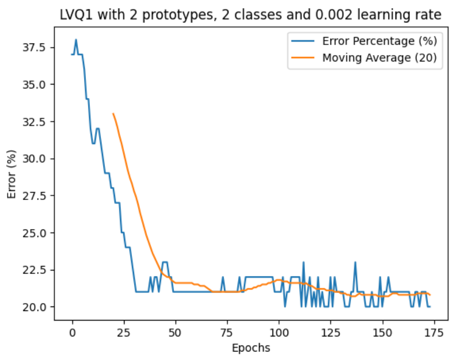
Results



(a) *Original classes with prototypes*

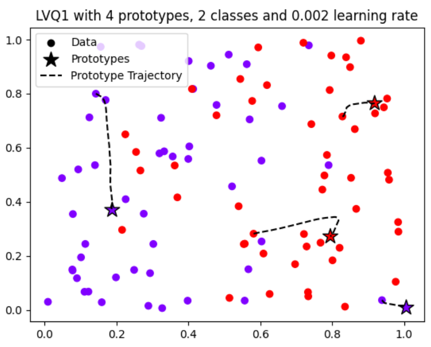


(b) *New classes with prototypes*

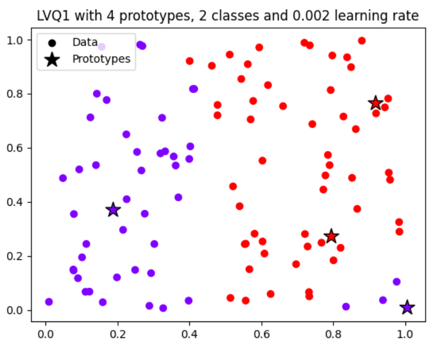


(c) *Error percentage per epoch*

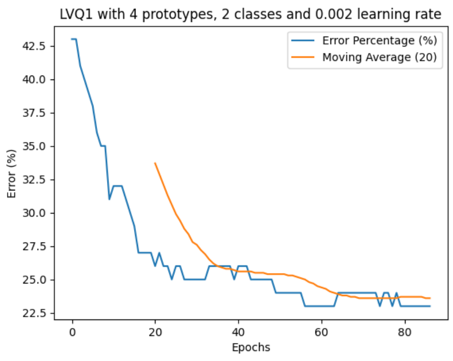
Figure 1: *These two plots represent LVQ1 with 1 prototype per class, 2 in total, with a learning rate of 0.002. The prototypes are shown at their location after the last epoch. The prototypes have been randomly initialized from one of the points in the corresponding class.* ***a*** *shows the original classes and the trajectories of the prototypes.* ***b*** *shows the new classes and the same prototypes the new classification is based on.* ***c*** *shows the classification error percentage per epoch and a moving average of 20 epochs.*



(a) *Original classes with prototypes*

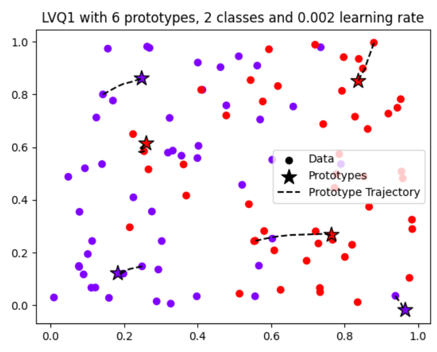


(b) *New classes with prototypes*

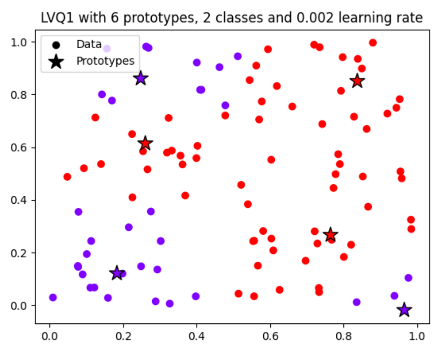


(c) *Error percentage per epoch*

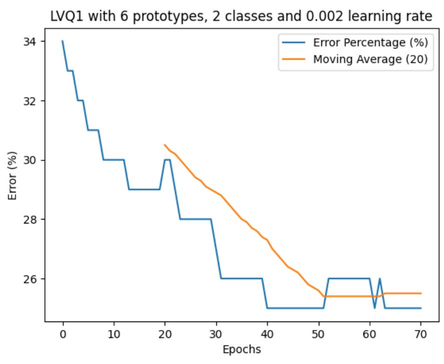
Figure 2: *These two plots represent LVQ1 with 2 prototypes per class, 4 in total, with a learning rate of 0.002. The prototypes are shown at their location after the last epoch. The prototypes have been randomly initialized from one of the points in the corresponding class.* ***a*** *shows the original classes and the trajectories of the prototypes.* ***b*** *shows the new classes and the same prototypes the new classification is based on.* ***c*** *shows the classification error percentage per epoch and a moving average of 20 epochs.*



(a) *Original classes with prototypes*

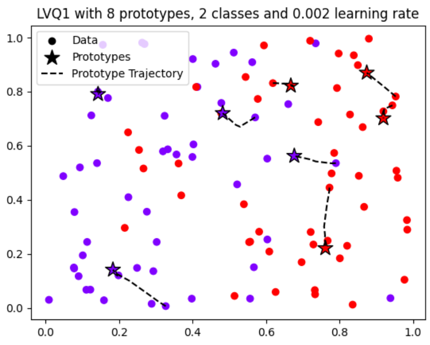


(b) *New classes with prototypes*

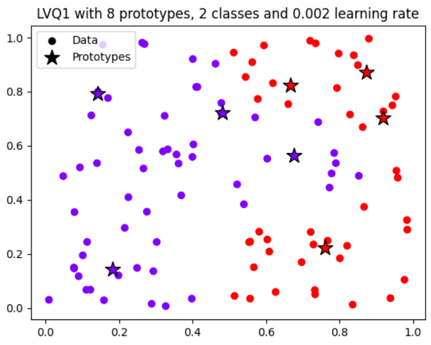


(c) *Error percentage per epoch*

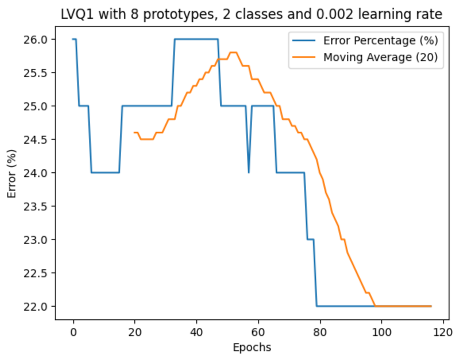
Figure 3: *These two plots represent LVQ1 with 3 prototypes per class, 6 in total, with a learning rate of 0.002. The prototypes are shown at their location after the last epoch. The prototypes have been randomly initialized from one of the points in the corresponding class.* **a** *shows the original classes and the trajectories of the prototypes.* **b** *shows the new classes and the same prototypes the new classification is based on.* **c** *shows the classification error percentage per epoch and a moving average of 20 epochs.*



(a) *Original classes with prototypes*

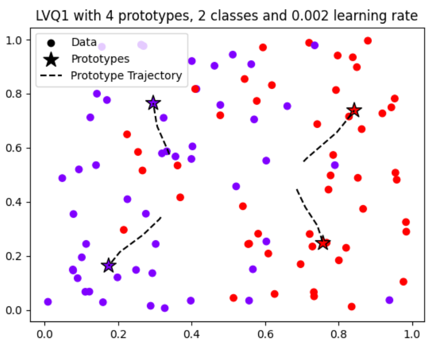


(b) *New classes with prototypes*

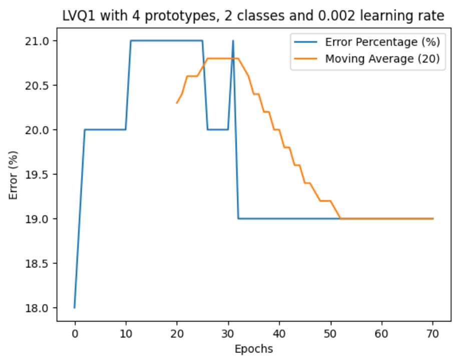


(c) *Error percentage per epoch*

Figure 4: *These two plots represent LVQ1 with 4 prototypes per class, 8 in total, with a learning rate of 0.002. The prototypes are shown at their location after the last epoch. The prototypes have been randomly initialized from one of the points in the corresponding class.* **a** *shows the original classes and the trajectories of the prototypes.* **b** *shows the new classes and the same prototypes the new classification is based on.* **c** *shows the classification error percentage per epoch and a moving average of 20 epochs.*



(a) *Original classes with prototypes*



(b) *Error percentage per epoch*

Figure 5: *These two plots represent LVQ1 with 2 prototypes per class, 4 in total, with a learning rate of 0.002. The prototypes are shown at their location after the last epoch. The prototypes have been initialized on the mean of a subset of the corresponding class. The subset is chosen by the amount of prototypes per class.* **a** *shows the original classes and the trajectories of the prototypes.* **b** *shows the classification error percentage per epoch and a moving average of 20 epochs.*

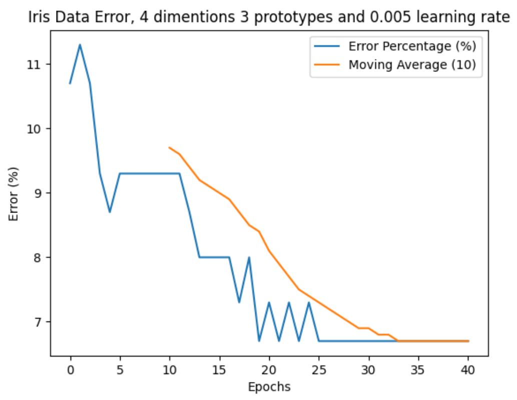
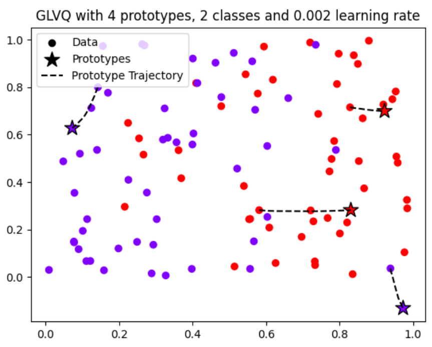
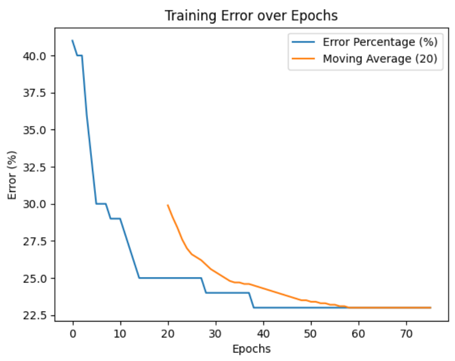


Figure 6: *This chart shows the classification error percentage per epoch and a moving average of 20 epochs for the Iris dataset. This is a 4-dimensional dataset with 3 classes. 3 prototypes have been used, which have been initialized randomly on one datapoint from a corresponding class.*



(a) *Original classes with prototypes*



(b) *New classes with prototypes*

Figure 7: *These two plots represent GLVQ with 2 prototypes per class, 4 in total, with a learning rate of 0.002. The prototypes are shown at their location after the last epoch. The prototypes have been initialized on the mean of a subset of the corresponding class. The subset is chosen by the amount of prototypes per class.* ***a*** *shows the original classes and the trajectories of the prototypes.* ***b*** *shows the classification error percentage per epoch and a moving average of 20 epochs.*

Discussion

LVQ1

In Figure 1a we see that the prototypes can find their way to their corresponding class. Figure 1c shows that the error percentage is decreasing rapidly in the first 30 epochs from about 38% to 21%. From that moment the error percentage is getting stable. The algorithm stopped after 175 epochs, when it decided that the error percentage was stable enough. This means that around 80% of the data is classified correctly. This is a good result, taking in mind only 2 prototypes were used and the data was very noisy.

In Figure 2a we see 2 prototypes per class. In this case, also the datapoint in the bottom right corner is classified correctly since it got initialized in that corner. Figure 2c shows that the error percentage is decreasing from 43% to 23% in the first 60 epochs. The algorithm stopped after about 70 epochs. It stopped at a higher error percentage than in the previous case with 1 prototype per class. This can be explained by the fact that the one purple prototype is initialized in the corner with one purple datapoint. Because of that, it causes more quantization errors in that particular corner. On top of that, the other purple prototype is the only one that is able to classify the purple datapoints on the left side.

Both scenarios with 3 and 4 prototypes per class do not show better results either, see Figure 3c and Figure 4c. This led to believe that the amount of prototypes with this specific dataset is 1 per class. Testing very many amounts of prototypes is not needed, this will lead to overfitting the dataset. This way it is easy to get an error percentage of 0% but will make classification of new datapoint very untrustworthy.

When testing 2 prototypes per class and the prototypes are initialized on the mean of the corresponding class as shown in Figure 5a, the results look very promising. The chart in Figure 5b confirms this. The error percentage is decreasing from 21% to 19% in the first 60 epochs. What also is interesting is that the error percentage starts at an even lower value than when the error percentage becomes stable. As you can see, the prototypes all start near the middle of the plot.

Apparently, this starting position gave a lower error percentage than when the error stabilizes. This is probably because the two classes are roughly separated in the middle of the plot. Due to the prototypes being repelled by the points of the other class, they are pushed away from the middle of the plot even though for the final classification this is not needed in this case.

Iris dataset

Figure **??** shows a starting error percentage of over 11% and a final error percentage of a little below 7%. Compared to the other dataset, this is a very good result. This is probably because the Iris dataset is very clean. It probably has well-defined clusters and the prototypes can find their way to the correct class. The prototypes starting at a low error percentage already can be an indication that points in the dataset are most likely already inside a well-defined cluster. Having used the Euclidean distance measure has been a reasonable choice for this dataset since all the features have about the same scale. When this was not the case, when one feature had a much higher scale than the others, it would have been better to use another distance measure.

GLVQ

Figure 7a shows that the prototypes can find their way to their corresponding class. Because of the rapid decline in learning rate, the prototypes find a stable position after 40 epochs. Since the learning rate at this point has decreased by a factor of 100, the prototypes are not able to move much anymore. so the error percentage will certainly be stable from this point as can be seen in Figure 7b. The stable error percentage is very similar to the one of LVQ1 with 2, 3 and 4 prototypes per class. Since the very simple implementation of this algorithm, there is not yet clear evidence that GLVQ is better than LVQ1.

Work distribution

- **Ramon**:
  - Mainly worked on the code for the LVQ1 algorithm and the bonus parts.
- **Stijn**:
  - Focussed on the analysis of the algorithms and the report.

When both parts were in the final stages, both team members worked together on the report to make sure everything was correct, clear and understandable. This strategy facilitated the sharing of knowledge and the work was divided in a way that both the team members could learn from each other.