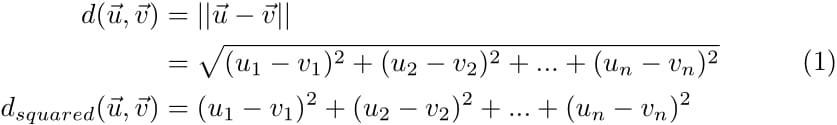
2- **Learning Vector Quantization**

LVQ is a so-called prototype-based learning method. One or more prototypes are used to represent each class in the dataset, each prototype is described as a point in the feature space. New (unknown) datapoints are then assigned the class of the prototype that is nearest to them. In order for “nearest” to make sense, a distance measure has to be defined. You are free to pick any distance metric, usually, Euclidean distance is the distance measure of choice.

**Recall that the Euclidean distance between 2 vectors in N dimensions is given by:**

****

**The code below is an implementation of the LVQ algorithm in Python.**

**import numpy as np**

|  | **# train\_lvq: trains an lvq system using the given training data and** |
| --- | --- |
|  | **# corresponding labels. Run the desired number of epochs using the** |
|  | **# given learning rate. Optional validation set to monitor performance.** |
|  | **def train\_lvq(data, labels, num\_epochs, learning\_rate, validation\_data=None, validation\_labels=None):** |
|  | **# Get unique class labels.** |
|  | **num\_dims = data.shape[1]** |
|  | **labels = labels.astype(int)** |
|  | **unique\_labels = list(set(labels))** |
|  |  |
|  | **num\_protos = len(unique\_labels)** |
|  | **prototypes = np.empty((num\_protos, num\_dims))** |
|  | **proto\_labels = []** |
|  |  |
|  | **# Initialize prototypes using class means.** |
|  | **for i in unique\_labels:** |
|  | **class\_data = data[labels == i, :]** |
|  |  |
|  | **# Compute class mean.** |
|  | **mean = np.mean(class\_data, axis=0)** |
|  |  |
|  | **prototypes[i] = mean** |
|  | **proto\_labels.append(i)** |
|  |  |
|  | **# Loop through data set.** |
|  | **for epoch in range(0, num\_epochs):** |
|  | **for fvec, lbl in zip(data, labels):** |
|  | **# Compute distance from each prototype to this point** |
|  | **distances = list(np.sum(np.subtract(fvec, p)\*\*2) for p in prototypes)** |
|  | **min\_dist\_index = distances.index(min(distances))** |
|  |  |
|  | **# Determine winner prototype.** |
|  | **winner = prototypes[min\_dist\_index]** |
|  | **winner\_label = proto\_labels[min\_dist\_index]** |
|  |  |
|  | **# Push or repel the prototype based on the label.** |
|  | **if winner\_label == lbl:** |
|  | **sign = 1** |
|  | **else:** |
|  | **sign = -1** |
|  |  |
|  | **# Update winner prototype** |
|  | **prototypes[min\_dist\_index] = np.add(prototypes[min\_dist\_index], np.subtract(fvec, winner) \* learning\_rate \* sign)** |
|  |  |
|  | **# Use validation set to test performance.** |
|  | **val\_err = 0** |
|  | **if validation\_labels is not None:** |
|  | **for fvec, lbl in zip(validation\_data, validation\_labels):** |
|  | **distances = list(np.sum(np.subtract(fvec, p) \*\* 2) for p in prototypes)** |
|  | **min\_dist\_index = distances.index(min(distances))** |
|  |  |
|  | **# Determine winner prototype label** |
|  | **winner\_label = proto\_labels[min\_dist\_index]** |
|  |  |
|  | **# Check if labels match** |
|  | **if not winner\_label == lbl:** |
|  | **val\_err = val\_err + 1** |
|  |  |
|  | **val\_err = val\_err / len(validation\_labels)** |
|  | **print("Epoch " + str(epoch) + ". Validation error: " + str(val\_err))** |
|  | **else:** |
|  | **print("Epoch " + str(epoch))**  **return (prototypes, proto\_labels)** |
|  |  |
|  |  |