Prototype-based learning. Part I: GMLVQ from scratch

Diego Hernández Jiménez

17/8/2024

Description

In my journey to bridge psychology and data science, I discovered Learning Vector Quantization (LVQ) and immediately saw its potential connection to human category learning. This realization led me to dive deeper, implementing LVQ from scratch, initially by coding all the operations using PyTorch, and then by abstracting the optimization and learning processes to fully leverage PyTorch's features.

Problem

Human category learning has been a subject of extensive study, with two predominant approaches: exemplar-based learning and prototype-based learning. Prototype learning, in particular, posits "that a category of things in the world (objects, animals, shapes, etc.) can be represented in the mind by a prototype. A prototype is a cognitive representation that captures the regularities and commonalities among category members and can help a perceiver distinguish category members from non-members" (Minda & Smith, 2011)

And what about LVQ? Well, this model learns prototypes and classifies new examples based on comparisons with these prototypes, which makes it a very promising model for human learning.

What is different and interesting?

One of the compelling aspects of LVQ is its flexibility in learning prototypes. Unlike traditional prototype models, which often rely on a single, average prototype per category, LVQ allows for multiple prototypes per category. These prototypes are not merely averages but can be more complex summaries of the data.

Furthermore, the similarity function in LVQ can be learned and adapted. Besides, this function doesn't have to be unique; each prototype can have its own similarity function. This flexibility enables some LVQ models to handle not only linearly separable categories but also those that are non-linearly separable —a significant improvement over standard prototype models, which struggle with non-linear separations.

Objective

While LVQ shows promise as a cognitive model, validating it against human experimental data is beyond my current means due to a lack of access to such data. Therefore, my focus will be on evaluating LVQ's effectiveness as a machine learning classifier. In this first part, I aim to replicate the results of a foundational paper in the LVQ field—the very paper that introduced me to this family of models (Schneider et al., 2009).

Clarifying Nomenclature

Through my exploration of LVQ models, I've encountered a variety of articles that use complex nomenclature, especially when explaining aspects like updating rules. To make the concepts more accessible, I've made a few adjustments in how the model is defined. These changes aim to simplify the understanding of LVQ, both for myself and for others who might find the traditional terminology challenging. Apologies if I just made it worse..., after all, I'm not a mathematician.

- By default, vectors are row vectors. This is due to the fact that, commonly, datasets are shaped $n \times p$ (number of observations $\times$ number of features)
- $\mathbf{x}$ = data point: $\begin{bmatrix} x_1 & \dots & x_p \end{bmatrix}$
- $\mathbf{w}$ = prototype: $\begin{bmatrix} w_1 & \dots & w_p \end{bmatrix}$
- $\mathbf{R}_{p \times p}$ = relevance matrix. In GMLVQ, it's a dense symmetric matrix of relevance/attention weights given to features where $tr(\mathbf{R}) = 1$. If diagonal, the model will be Generalized Relevance LVQ (GRLVQ) and won't account for correlations between features. If identity matrix, we have the Generalized LVQ (GLVQ).
- $\mathbf{R} = \mathbf{Q}^\intercal\mathbf{Q} = \mathbf{Q}\mathbf{Q}^\intercal$ (because of symmetry)

How does the model work? A trained lvq model works by comparing an unseen exemplar $\mathbf{x}$ to each of the learned prototypes stored in memory $\mathbf{w}_1, ..., \mathbf{w}_C$. There may be one for each of the C classes, as here, or more than one. The comparison is done using a dissimilarity function d (sometimes called Mahalanobis distance, although I don't think it's exactly the same) defined as:

$$d(\mathbf{x}, \mathbf{w}_k, \mathbf{Q}) = (\mathbf{x} - \mathbf{w}_k)\mathbf{R}(\mathbf{x} - \mathbf{w}_k)^\intercal$$

In terms of $\mathbf{Q}$:

$$\begin{aligned} d(\mathbf{x}, \mathbf{w}_k, \mathbf{Q}) &= (\mathbf{x} - \mathbf{w}_k)\mathbf{Q}^\intercal\mathbf{Q}(\mathbf{x} - \mathbf{w}_k)^\intercal \\ &= \Big((\mathbf{x} - \mathbf{w}_k)\mathbf{Q}^\intercal\Big)\Big(\mathbf{Q}(\mathbf{x} - \mathbf{w}_k)^\intercal\Big) \\ &= \Big(\mathbf{Q}(\mathbf{x} - \mathbf{w}_k)^\intercal\Big)^\intercal\Big(\mathbf{Q}(\mathbf{x} - \mathbf{w}_k)^\intercal\Big) \\ &= \|\mathbf{Q}\,(\mathbf{x} - \mathbf{w}_k)^\intercal\|_2^2 \end{aligned}$$

In the case of LGMLVQ we'd just add a subscript to $\mathbf{Q}$ indicating which matrix are we referring to, because there is one per prototype.

```
def dist_to_prototypes(x:torch.tensor, W:list, Q:torch.tensor, lvq_mode:str) -> torch.tensor:
  """
    Computes the distances between a given sample `x` and the prototypes in `W`,
    using a distance function that depends on the specified `lvq_mode`. The distances
    are calculated with the help of a matrix or matrices `Q`, depending on the mode.

    Args:
        x (torch.Tensor): A tensor representing the input sample. It should have the same
        dimensionality as the prototypes in `W`.

        W (list of torch.Tensor): A list containing the prototypes, where each element
        is a tensor representing a prototype. Each prototype has the same dimensionality
        as `x`.

        Q (torch.Tensor or list of torch.Tensor): A transformation matrix or a list of
```

```
        matrices used to compute the distance.
        - If `lvq_mode` is 'gmlvq', `Q` should be a single matrix.
        - If `lvq_mode` is 'lgmlvq', `Q` should be a list of matrices, one for each prototype.

        lvq_mode (str): Specifies the type of Learning Vector Quantization (LVQ) model to use.
        - 'gmlvq': Generalized Matrix Learning Vector Quantization, where a single matrix `Q` is used
          for all prototypes.
        - 'lgmlvq': Local Generalized Matrix Learning Vector Quantization, where each prototype
          has its own matrix `Q`.

    Returns:
        torch.Tensor: A tensor containing the computed distances between `x` and each prototype
        in `W`. The length of the output tensor is equal to the number of prototypes.

    Raises:
        ValueError: If an invalid `lvq_mode` is provided, or if the dimensions of `Q` do not match
        the requirements of the specified `lvq_mode`.

    Example:
        dists = dist_to_prototypes(x, W, Q, lvq_mode='gmlvq')
    """

  p = len(W)
  dists = torch.zeros(p)
  if lvq_mode == 'gmlvq':
    for j in range(p):
      raw_diff = x - W[j]
      dists[j] = torch.linalg.multi_dot([raw_diff, Q.t(), Q, raw_diff.t()])
      # dists[j] = torch.linalg.vector_norm(raw_diff @ Q, ord=2)**2

  elif lvq_mode == 'lgmlvq':
    for j in range(p):
      raw_diff = x - W[j]
      dists[j] = torch.linalg.multi_dot([raw_diff, Q[j].t(), Q[j], raw_diff.t()])
      # dists[j] = torch.linalg.vector_norm(raw_diff @ Q[j], ord=2)**2

  else:
    print('choose appropiate lvq model')
    return None

  return dists
```

This version is the one we are actually going to use for training purposes.

The decision rule is to assign to the exemplar the class corresponding to the closest prototype:

$$\hat{c} = \underset{k \in C}{\operatorname{argmin}}\, d(\mathbf{x}, \mathbf{w}_k, \mathbf{R})$$

```
def predict_class(X:torch.tensor, Wclasses:list, W:list, Q:torch.tensor, lvq_mode:str) -> torch.tensor:
  """
    Predicts the class labels for a set of input samples based on their
    distances to prototypes.
    The function uses a specified Learning Vector Quantization (LVQ) mode
```

```
    to calculate these distances.

    Args:
        X (torch.Tensor): A tensor of shape (n_samples, n_features) containing
        the input samples for which the class labels are to be predicted.

        Wclasses (torch.Tensor): A tensor containing the class labels associated
        with each prototype. The length of `Wclasses` should match the number of prototypes.

        W (list of torch.Tensor): A list of tensors where each tensor
        represents a prototype. The prototypes have the same dimensionality
        as the input samples in `X`.

        Q (torch.Tensor or list of torch.Tensor): The transformation matrix
        (or matrices) used to compute the distance between the samples and prototypes.
        - If `lvq_mode` is 'gmlvq', `Q` should be a single matrix.
        - If `lvq_mode` is 'lgmlvq', `Q` should be a list of matrices, one for each prototype.

        lvq_mode (str): Specifies the type of LVQ model to use for distance computation.
        - 'gmlvq': Generalized Matrix Learning Vector Quantization,
        where a single matrix `Q` is used for all prototypes.
        - 'lgmlvq': Local Generalized Matrix Learning Vector Quantization,
        where each prototype has its own matrix `Q`.

    Returns:
        torch.Tensor: A tensor of predicted class labels for each input sample in `X`.
        The length of the output tensor is equal to the number of input samples.

    Example:
        preds = predict_class(X, Wclasses, W, Q, lvq_mode='gmlvq')

    Details:
        - For each input sample in `X`, the function computes the distances
        to all prototypes using the `dist_to_prototypes` function.
        - The class label associated with the prototype that has the minimum
        distance to the sample is selected as the predicted class.
        - The function iterates over all input samples to generate predictions for each one.
    """

  preds = torch.zeros(X.shape[0])
  for i in range(X.shape[0]):
    x = X[i]
    d = dist_to_prototypes(x, W, Q, lvq_mode)
    preds[i] = Wclasses[torch.argmin(d)]

  return preds
```

Now let's get back for a moment to the similarity function and examine how the assumptions on $\mathbf{R}$ change the final model.

- If it's dense and unique, we have the explained GMLVQ.
- If it's diagonal, then $d(\mathbf{x}, \mathbf{w}_k, \mathbf{R}) = \sum_{i=1}^{p} r_i(x_i - w_{ki})^2$, which is the squared euclidean metric used for GRLVQ.

- If $\mathbf{R} = \mathbf{I}$, then $d(\mathbf{x}, \mathbf{w}_k, \mathbf{R}) = (\mathbf{x} - \mathbf{w}_k)(\mathbf{x} - \mathbf{w}_k)^\intercal = \|(\mathbf{x} - \mathbf{w}_k)\|_2^2$, the basic squared euclidean distance of GLVQ.
- Finally, if our relevance matrix is not unique and have one for each prototype, then we have a localized version of the model (LGMLQ or LGRLVQ). For this first part of the project we've just implemented GMLVQ and LGMLVQ.

To see more clearly how the relevance matrix affects the distance calculations, we can simulate an extremely basic example where we have bidimensional instances:

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$$

$$\mathbf{w}_k = \begin{bmatrix} w_1 & w_2 \end{bmatrix}$$

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} \\ r_{12} & r_{22} \end{bmatrix}$$

Notice how we only have r_{12} and not r_{21}. That is because, as mentioned, the relevance matrix is symmetric, so $r_{12} = r_{21}$. The distance between the instance and the prototype would be:

$$\begin{aligned} d(\mathbf{x}, \mathbf{w}_k, \mathbf{R}) &= (\mathbf{x} - \mathbf{w}_k)\mathbf{R}(\mathbf{x} - \mathbf{w}_k)^\intercal \\ &= \begin{bmatrix} \delta_1 & \delta_2 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} \\ r_{12} & r_{22} \end{bmatrix} \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix} \\ &= \begin{bmatrix} (\delta_1 r_{11} + \delta_2 r_{12}) & (\delta_1 r_{12} + \delta_2 r_{22}) \end{bmatrix} \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix} \\ &= \delta_1^2 \cdot r_{11} + \delta_2^2 \cdot r_{22} + \delta_1 \cdot \delta_2 r_{12} \end{aligned}$$

where $\delta_i = x_i - w_i$.

Now it's easier to see how the choice of the model/relevance matrix impacts distance:

GLVQ $(r_{12} = 0\ ;\ r_{ii} = 1) : d(\mathbf{x}, \mathbf{w}_k, \mathbf{R}) = \delta_1^2 + \delta_2^2$

GRLVQ $(r_{12} = 0) :\ d(\mathbf{x}, \mathbf{w}_k, \mathbf{R}) = \delta_1^2 \cdot r_{11} + \delta_2^2 \cdot r_{22}$

GMLVQ $: d(\mathbf{x}, \mathbf{w}_k, \mathbf{R}) = \delta_1^2 \cdot r_{11} + \delta_2^2 \cdot r_{22} + \delta_1 \cdot \delta_2 r_{12}$

LGRLVQ $: d(\mathbf{x}, \mathbf{w}_k, \mathbf{R}) = \delta_1^2 \cdot r_{11}^k + \delta_2^2 \cdot r_{22}^k$

LGMLVQ $: d(\mathbf{x}, \mathbf{w}_k, \mathbf{R}) = \delta_1^2 \cdot r_{11}^k + \delta_2^2 \cdot r_{22}^k + \delta_1 \cdot \delta_2 r_{12}^k$

Notice how in the localized versions the relevance weights depend on the prototype used in the comparison. That's exactly what makes LVQ so powerful, to the point to achieve good classification rates of categories with nonlinear boundaries.

How does the model learn?

One of the things that also attracted me from these lvq models is the fact that parameters can be optimized using gradient descent. This is specially attractive for me because is the optimization method that i understand the best. The loss function is the relative distance loss, which was first used to develop the GLVQ model, and is defined as:

$$\mathcal{L}(d^+, d^-) = \phi\Big(\frac{d^+ - d^-}{d^+ + d^-}\Big)$$

```python
def relative_dist_loss(d_pos:torch.tensor, d_neg:torch.tensor, activ:str='linear') -> torch.tensor:
  """
    Computes the relative distance loss between positive and negative distances
    using a specified activation function.This loss is often used in
    Learning Vector Quantization (LVQ) models to measure the relative similarity
    between a sample and prototypes.

    Args:
        d_pos (torch.Tensor): The "positive distance", representing the distance
        between the sample andthe closest correct prototype.

        d_neg (torch.Tensor): The "negative distance", representing the distance
        between the sample and the closest incorrect prototype.

        activ (str, optional): The activation function to apply to the relative distance.
        Default is 'linear'. Supported values are:
        - 'linear': No activation function (identity).
        - 'sigmoid': Applies a sigmoid activation to the result
        (careful,this is not supported in practice).

    Returns:
        torch.Tensor: The result of applying the activation function to the relative distance.
        The output is a tensor of the same shape as the input distances.

    Raises:
        KeyError: If an unsupported activation function is provided.

    Example:
        loss = relative_dist_loss(d_pos, d_neg, activ='sigmoid')

    Details:
        - The function first computes the relative difference between the positive
        and negative distances, normalized by their sum: `(d_pos - d_neg) / (d_pos + d_neg)`.
        - It then applies the specified activation function (`linear` or `sigmoid`) to this value.
    """

  dict_activations = {
      'linear': torch.nn.Identity(),
      'sigmoid': torch.nn.Sigmoid()
  }
  phi = dict_activations[activ]

  return phi( (d_pos - d_neg) / (d_pos + d_neg) )
```

where ϕ is a function that can add an additional transformation. Here we are going to assume, as in the paper I'm using as reference, that $\phi(x) = x$, but in the second part we'll try some other functions such as the sigmoid or ReLU. Notice that we have d^+ and d^-. Those distances are the distance of the data point from the closest prototype with the *correct* label ($\mathbf{w}^+$) and the distance of the data point from the closest prototype with the *incorrect* label ($\mathbf{w}^-$), respectively (from the energy-based model learning point of view, the latter would correspond to the "most offending answer", a name that I found quite funny). In the "localized" model scenario, we would also have $\mathbf{R}^+$ and $\mathbf{R}^-$

With stochastic gradient descent (in the second part we'll work with batch gradient descent):

$$\mathbf{w}_{t+1}^{+} = \mathbf{w}_{t}^{+} - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{w}_{t}^{+}}$$
$$\mathbf{w}_{t+1}^{-} = \mathbf{w}_{t}^{-} - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{w}_{t}^{-}}$$

For the GRLVQ and GMLVQ we also need to update de pseudo-relevance matrix:

$$\mathbf{Q}_{t+1} = \mathbf{Q}_{t} - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{Q}_{t}}$$

And in the case of the localized models, we instead update separately $\mathbf{Q}^{+}$ and $\mathbf{Q}^{-}$:

$$\mathbf{Q}_{t+1}^{+} = \mathbf{Q}_{t}^{+} - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{Q}_{t}^{+}}$$
$$\mathbf{Q}_{t+1}^{-} = \mathbf{Q}_{t}^{-} - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{Q}_{t}^{-}}$$

After that, we would also normalize the values of the matrix/matrices to enforce the $tr(\mathbf{R}) = 1$ constraint.

How to get the derivatives for every parameter? For that it's convenient to first decompose the output of the loss in the "forward pass" into a series of sequential transformations:

$$\mathcal{L}(\mu) = \mu$$

$$\mu(d^{+}, d^{-}) = \frac{d^{+} - d^{-}}{d^{+} + d^{-}}$$

$$d(\mathbf{x}, \mathbf{w}, \mathbf{Q}) = (\mathbf{x} - \mathbf{w})\ \mathbf{Q}^{\intercal}\mathbf{Q}\ (\mathbf{x} - \mathbf{w})^{\intercal} = \|\mathbf{Q}(\mathbf{x} - \mathbf{w})^{\intercal}\|_{2}^{2}$$

(remember that $\mathbf{R}$ is not learned directly)

Then the derivatives of each step are:

$$\frac{\partial \mathcal{L}}{\partial \mu} = 1$$

$$\frac{\partial \mu}{\partial d^{+}} = \frac{(d^{+} + d^{-}) - (d^{+} - d^{-})}{(d^{+} + d^{-})^{2}} = \frac{2d^{-}}{(d^{+} + d^{-})^{2}}$$

$$\frac{\partial \mu}{\partial d^{-}} = \frac{-(d^{+} + d^{-}) - (d^{+} - d^{-})}{(d^{+} + d^{-})^{2}} = \frac{-2d^{-}}{(d^{+} + d^{-})^{2}}$$

$$\frac{\partial d}{\partial \mathbf{w}} = -2\mathbf{Q}^{\intercal}\mathbf{Q}(\mathbf{x} - \mathbf{w})^{\intercal}$$

$$\frac{\partial d}{\partial \mathbf{Q}} = 2\mathbf{Q}(\mathbf{x} - \mathbf{w})^{\intercal}(\mathbf{x} - \mathbf{w})$$

And by chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}^+} = 1 \cdot \frac{2d^-}{(d^+ + d^-)^2} \cdot \left(-2\mathbf{Q}^\intercal \mathbf{Q}(\mathbf{x} - \mathbf{w}^+)^\intercal\right)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}^-} = 1 \cdot \frac{-2d^-}{(d^+ + d^-)^2} \cdot \left(-2\mathbf{Q}^\intercal \mathbf{Q}(\mathbf{x} - \mathbf{w}^-)^\intercal\right)$$

$$\frac{\partial L}{\partial \mathbf{Q}^+} = 1 \cdot \frac{2d^-}{(d^+ + d^-)^2} \cdot 2\mathbf{Q}\ (\mathbf{x} - \mathbf{w}^+)^\intercal(\mathbf{x} - \mathbf{w}^+)$$

$$\frac{\partial L}{\partial \mathbf{Q}^-} = 1 \cdot \frac{-2d^-}{(d^+ + d^-)^2} \cdot 2\mathbf{Q}(\mathbf{x} - \mathbf{w}^-)^\intercal(\mathbf{x} - \mathbf{w}^-)$$

If GMLVQ (only one relevance matrix):

$$\frac{\partial L}{\partial \mathbf{Q}} = \frac{\partial L}{\partial \mathbf{Q}^+} + \frac{\partial L}{\partial \mathbf{Q}^-}$$

Now we have everything prepared to implement gradient descent. We can make some tweaks such as including a decaying learning rate, but that's extra (I included it anyway, to be consistent with the paper).

In code:

```
def train_lvq(data:torch.utils.data.TensorDataset,
              Wclasses:list,
              W:list,
              Q:torch.tensor,
              lvq_mode:str,
              eta0:dict,
              tau:float,
              epochs:int) -> tuple:
  """
    Trains a Learning Vector Quantization (LVQ) model by updating prototypes
    and relevance matrices using gradient descent. The training process iteratively
    adjusts the prototypes and relevance matrices to minimize the classification error.

    Args:
        data (torch.utils.data.TensorDataset): The training dataset, containing feature vectors
        and corresponding class labels.

        W (list of torch.Tensor): A list of tensors representing the prototypes.
        Each tensor corresponds to a prototype.

        Q (torch.Tensor or list of torch.Tensor): The relevance matrix (or matrices)
        used to compute distances.
        - If `lvq_mode` is 'gmlvq', `Q` is a single matrix shared by all prototypes.
        - If `lvq_mode` is 'lgmlvq', `Q` is a list of matrices, one for each prototype.

        lvq_mode (str): Specifies the LVQ model type.
        - 'gmlvq': Generalized Matrix Learning Vector Quantization, where a single relevance matrix is
        - 'lgmlvq': Local Generalized Matrix Learning Vector Quantization,
        where each prototype has its own relevance matrix.
```

```
        eta0 (dict): A dictionary with initial learning rates for
        updating prototypes (`'W'`) and relevance matrices (`'Q'`).

        tau (float): A hyperparameter used to adjust the learning rate over time.

        epochs (int): The number of epochs to run the training process.

    Returns:
        tuple: A tuple containing the updated prototypes `W` and relevance matrices `Q`.

    Example:
        W, Q = train_lvq(data, W, Q, lvq_mode='lgmlvq', eta0={'W': 0.01, 'Q': 0.001}, tau=0.5, epochs=1

    Details:
        - The function iterates through the dataset for the specified number of epochs.
        - For each sample, it computes the distances to all prototypes using `dist_to_prototypes`.
        - The prototype closest to the correct class and the prototype closest to
        the incorrect class are identified.
        - Gradients are computed with respect to the prototypes and relevance matrices
        to minimize the loss, which is calculated using `relative_dist_loss`.
        - The prototypes and relevance matrices are updated using gradient descent,
        with learning rates controlled by `eta0` and `tau`.
        - In the case of 'lgmlvq', separate updates are performed for each prototype's
        relevance matrix, while in 'gmlvq', a single relevance matrix is updated.
        - Relevance matrices are normalized after each update to maintain constraints.

    Raises:
        ValueError: If an invalid `lvq_mode` is provided or if other input parameters
        are not compatible with the training process.

    Notes:
        - The learning rate for the prototypes remains constant, while the
        learning rate for the relevance matrices decreases over time based on the epoch number.
    """

    n = len(data)
    for epoch in range(epochs):
      for i in torch.randperm(n):
        x, lab = data[i]
        d = dist_to_prototypes(x, W, Q, lvq_mode)
        d_pos= torch.min(d[Wclasses == lab])
        id_winner = torch.argwhere(d == d_pos)
        w_pos = W[id_winner]
        Q_pos = Q[id_winner]

        d_neg = torch.min(d[Wclasses != lab])
        id_loser = torch.argwhere(d == d_neg)
        w_neg = W[id_loser]
        Q_neg = Q[id_loser]

        loss = relative_dist_loss(d_pos, d_neg)

        # DERIVATIVES
```

```
    dloss_dmu = 1
    dmu_ddpos = (2*d_neg) / torch.pow(d_pos + d_neg, 2)
    dmu_ddneg = (-2*d_pos) / torch.pow(d_pos + d_neg, 2)

    if lvq_mode == 'lgmlvq':
      ddpos_dwpos = (-2) * torch.linalg.multi_dot([Q_pos.T, Q_pos, (x - w_pos)])
      ddneg_dwneg = (-2) * torch.linalg.multi_dot([Q_neg.T, Q_neg, (x - w_neg)])

      ddpos_dQpos = 2 * torch.linalg.matmul(Q_pos, torch.outer(x - w_pos, x - w_pos))
      ddneg_dQneg = 2 * torch.linalg.matmul(Q_neg, torch.outer(x - w_neg, x - w_neg))

    else:
      ddpos_dwpos = (-2) * torch.linalg.multi_dot([Q.T, Q, (x - w_pos)])
      ddneg_dwneg = (-2) * torch.linalg.multi_dot([Q.T, Q, (x - w_neg)])

      ddpos_dQpos = 2 * torch.linalg.matmul(Q, torch.outer(x - w_pos, x - w_pos))
      ddneg_dQneg = 2 * torch.linalg.matmul(Q, torch.outer(x - w_neg, x - w_neg))

    dloss_dwpos = dloss_dmu * dmu_ddpos * ddpos_dwpos
    dloss_dwneg = dloss_dmu * dmu_ddneg * ddneg_dwneg

    dloss_dQpos = dloss_dmu * dmu_ddpos * ddpos_dQpos
    dloss_dQneg = dloss_dmu * dmu_ddneg * ddneg_dQneg

    if lvq_mode == 'gmlvq':
      dloss_dQ = dloss_dQpos + dloss_dQneg
    else:
      pass


    # PARAMETERS UPDATE
    # uncomment for dynamic prototype learning rate
    # eta = (4*eta0['W'])/(1 + tau*(epoch - 1))
    eta = eta0['W']
    W[id_winner] = w_pos - eta * dloss_dwpos
    W[id_loser] = w_neg - eta * dloss_dwneg

    eta = (4*eta0['Q'])/(1 + tau*(epoch - 1))
    # uncomment for fixed pseudo-relevance learning rate
    # eta = eta0['Q']

    if lvq_mode == 'gmlvq':
      Q -= eta * dloss_dQ
      Q = Q / torch.sqrt( (Q.T @ Q).diag().sum() )
    else:
      Q_pos -= eta * dloss_dQpos

      # transform Q_pos and Q_neg to satisfy constraints
      Q_pos = Q_pos / torch.sqrt( (Q_pos.T @ Q_pos).diag().sum() )
      Q[id_winner] = Q_pos

      Q_neg -= eta * dloss_dQneg
      Q_neg = Q_neg / torch.sqrt( (Q_neg.T @ Q_neg).diag().sum() )
```

```
        Q[id_loser] = Q_neg


  return W, Q
```

Other architectural details

How we initialize parameters? That's a key question and, in my a case, I followed the suggestions of PAPER. For the prototypes, each vector associated with class k consists of the average of a random sample of z observations of class k, with z being equal to: 1/number of prototypes for class k $\times$ total number of observations in class k. For the simple case of one prototype per class this means z = total number of observations in class k, so prototypes are initialized as class conditional means.

```
def init_prototypes(data:torch.utils.data.TensorDataset, n_prototypes_per_class:list) -> list:
  """
    Initializes prototypes for each class based on the input dataset. For each class, a
    specified number of prototypes is generated by averaging random subsets of samples
    from that class.

    Args:
        data (torch.utils.data.TensorDataset): A dataset containing features and labels.
        The dataset is expected to have two tensors:
        - `X`: A tensor of shape (n_samples, n_features) representing the feature vectors.
        - `y`: A tensor of shape (n_samples,) representing the class labels corresponding
          to each feature vector.

        n_prototypes_per_class (list of int): A list where each element specifies the
        number of prototypes to generate for the corresponding class. The index of the
        element represents the class ID.

    Returns:
        list of torch.Tensor: A list of tensors, where each tensor is a prototype. The length
        of the list is the total number of prototypes, and each prototype has the same
        dimensionality as the input features.

    Details:
        - For each class, the function identifies all samples belonging to that class.
        - It then creates the specified number of prototypes by randomly selecting subsets
          of these samples and averaging their feature vectors.
        - The size of each subset is determined by dividing the total number of samples
          in the class by the number of prototypes to be generated for that class.

    Example:
        If `n_prototypes_per_class = [2, 3]` and the dataset has two classes (0 and 1),
        the function will generate 2 prototypes for class 0 and 3 prototypes for class 1.

        W = init_prototypes_2(data, n_prototypes_per_class=[2, 3])
    """
  X, y = data.tensors
  W = []
  for class_id, times_each_proto in enumerate(n_prototypes_per_class):
    ids = torch.nonzero(y == class_id).squeeze()
```

```
    subset_size = (1/times_each_proto) * len(ids)
    subset_ids = ids[torch.randperm(len(ids))[:int(subset_size)]]
    w = X[subset_ids].mean(dim=0)
    W.append(w)

  return W
```

In the case of $\mathbf{Q}$, in the first iteration is just a diagonal matrix with all its diagonal elements set to: $\sqrt{\frac{1}{p}}$. Given that $\mathbf{R} = \mathbf{Q}^\intercal\mathbf{Q}$, this ensures that the relevance matrix satisfies the constraints of symmetry and $tr(\mathbf{R}) = 1$.

```
def init_pseudorelevance_matrix(n_prototypes_per_class:list, n_features:int, lvq_mode:str) -> torch.tens
  """
    Initializes the pseudorelevance matrix or matrices used in Learning Vector Quantization (LVQ) model
    The initialization depends on the specified LVQ mode.

    Args:
        n_prototypes_per_class (list of int): A list where each element specifies the number
        of prototypes to generate for each class. The sum of this list gives the total number
        of prototypes.

        n_features (int): The number of features in the input data, which determines the size
        of the pseudorelevance matrix (or matrices).

        lvq_mode (str): Specifies the LVQ model type.
        - 'gmlvq': Generalized Matrix Learning Vector Quantization, where a single pseudorelevance
          matrix `Q` is used for all prototypes.
        - 'lgmlvq': Local Generalized Matrix Learning Vector Quantization, where each prototype
          has its own pseudorelevance matrix `Q`.

    Returns:
        torch.Tensor or list of torch.Tensor:
        - If `lvq_mode` is 'gmlvq', returns a single matrix `Q` of shape (n_features, n_features).
        - If `lvq_mode` is 'lgmlvq', returns a list of matrices `Q`, where each matrix is of
          shape (n_features, n_features) and corresponds to a prototype.

    Raises:
        ValueError: If an invalid `lvq_mode` is provided.

    Details:
        - In 'gmlvq' mode, a single matrix `Q` is initialized as the square root of the identity
          matrix divided by the number of features.
        - In 'lgmlvq' mode, a list of such matrices is created, with one matrix for each prototype.

    Example:
        Q = init_pseudorelevance_matrix(n_prototypes_per_class=[2, 3], n_features=5, lvq_mode='gmlvq')
    """

  n_prototypes_total = sum(n_prototypes_per_class)
  if lvq_mode == 'gmlvq':
    Q = torch.sqrt(torch.eye(n_features)/n_features)

  elif lvq_mode == 'lgmlvq':
```

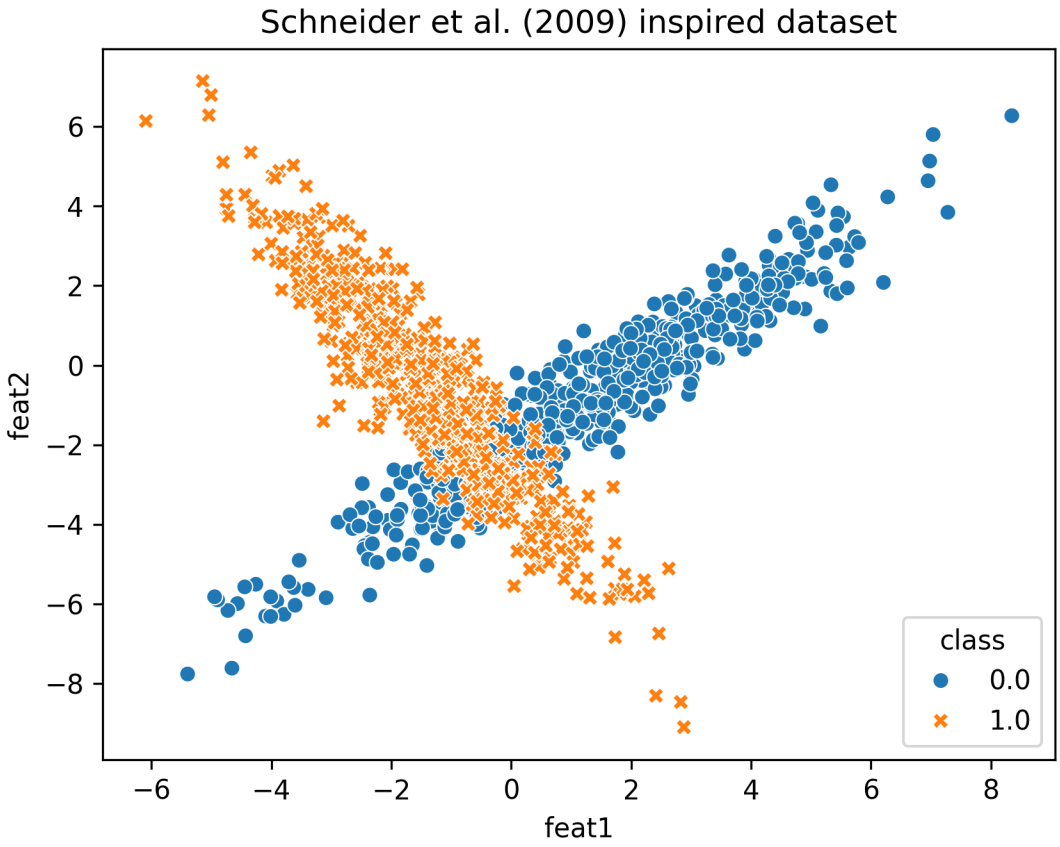
```
    Q = [torch.sqrt(torch.eye(n_features)/n_features) for _ in range(n_prototypes_total)]

  else:
    print('choose appropiate lvq model')
    return None

  return Q
```

Validation

Let's now try the model with the artificial data generated in Schneider et al. (2009)

```
def create_artificial_data(n:int, seed:int) -> tuple:
  """
  The dataset consists of two clusters of data points,
    each generated from a different multivariate normal distribution with specific
    means, covariance matrices, and rotations (inspired by Schneider et al., 2009).

    Args:
        n (int): Number of samples to generate for each cluster.
        seed (int): Random seed for reproducibility.

    Returns:
        tuple: A tuple containing:
            - data (torch.utils.data.TensorDataset): A PyTorch TensorDataset containing
              the features and labels.
            - df (pandas.DataFrame): A pandas DataFrame containing the features and labels,
              where 'feat1' and 'feat2' are the feature columns and 'class' is the label column.

    Details:
        - Two clusters are generated from multivariate normal distributions with means
          [1.5, 0.0] and [-1.5, 0.0], and covariance matrices with diagonal values
          [0.5^2, 3.0^2].
        - The first cluster is rotated by pi/4 radians, and the second cluster by -pi/6 radians.
        - The final dataset is a combination of both clusters, with labels 0 for the first
          cluster and 1 for the second.

    Example:
        data, df = create_artificial_data(n=100, seed=42)
  """

  torch.manual_seed(seed)
  MV0 = torch.distributions.MultivariateNormal(
      loc=torch.tensor([1.5, 0.0]),
      covariance_matrix=torch.diag(torch.tensor([0.5, 3.0])).pow(2) # [0.1,3.0] [0.5,3.0]
  )
  MV1 = torch.distributions.MultivariateNormal(
      loc=torch.tensor([-1.5, 0.0]), # [-0.8,0.0]
      covariance_matrix=torch.diag(torch.tensor([0.5, 3.0])).pow(2) # [0.1,3.0] [0.5,3.0]
  )
  cluster0 = MV0.sample((1, n)).squeeze(0)
  cluster1 = MV1.sample((1, n)).squeeze(0)
```

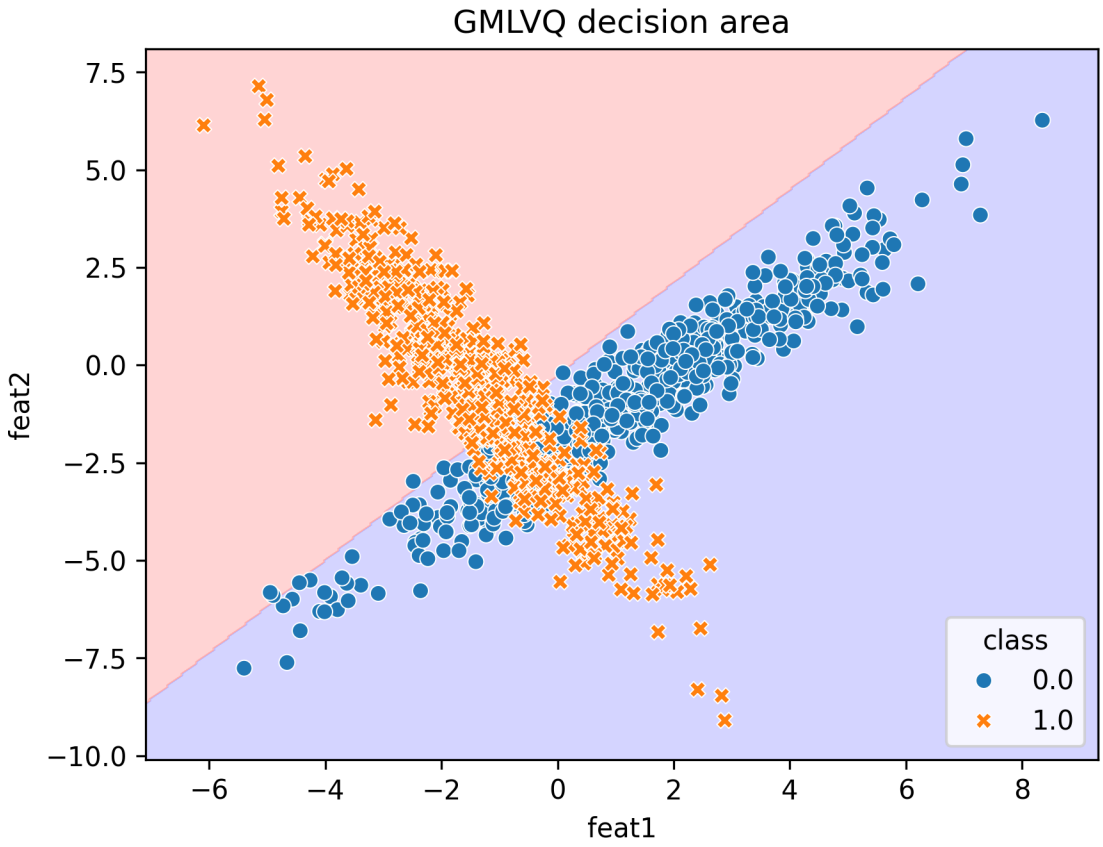
```
  phi = torch.tensor([torch.pi/4, -torch.pi/6])
  rot_matrix0 = torch.tensor(
      [[torch.cos(phi[0]), -torch.sin(phi[0])],
      [torch.sin(phi[0]), torch.cos(phi[0])]]
  )
  rot_matrix1 = torch.tensor(
      [[torch.cos(phi[1]), -torch.sin(phi[1])],
      [torch.sin(phi[1]), torch.cos(phi[1])]]
  )
  X0 = torch.matmul(cluster0, rot_matrix0)
  X1 = torch.matmul(cluster1, rot_matrix1)
  X = torch.cat([X0, X1], dim=0)
  y = torch.cat([torch.zeros(n), torch.ones(n)])

  # tensor dataset
  data = torch.utils.data.TensorDataset(X, y)

  # pandas dataframe
  df = pd.DataFrame(X, columns=['feat1', 'feat2'])
  df['class'] = y

  return data, df


n = 600
data, df = create_artificial_data(n, seed=42)
p = sns.scatterplot(data=df, x="feat1", y="feat2", hue="class", style="class")
p.set_title('Schneider et al. (2009) inspired dataset')
plt.savefig('SchneiderGMLVQ_data.png', dpi=300)
```



If we instantiate a GMLVQ model and train it for 25 epochs (way less that in the original paper, but there was no need of more epochs) we achieve around 0.8 accuracy:

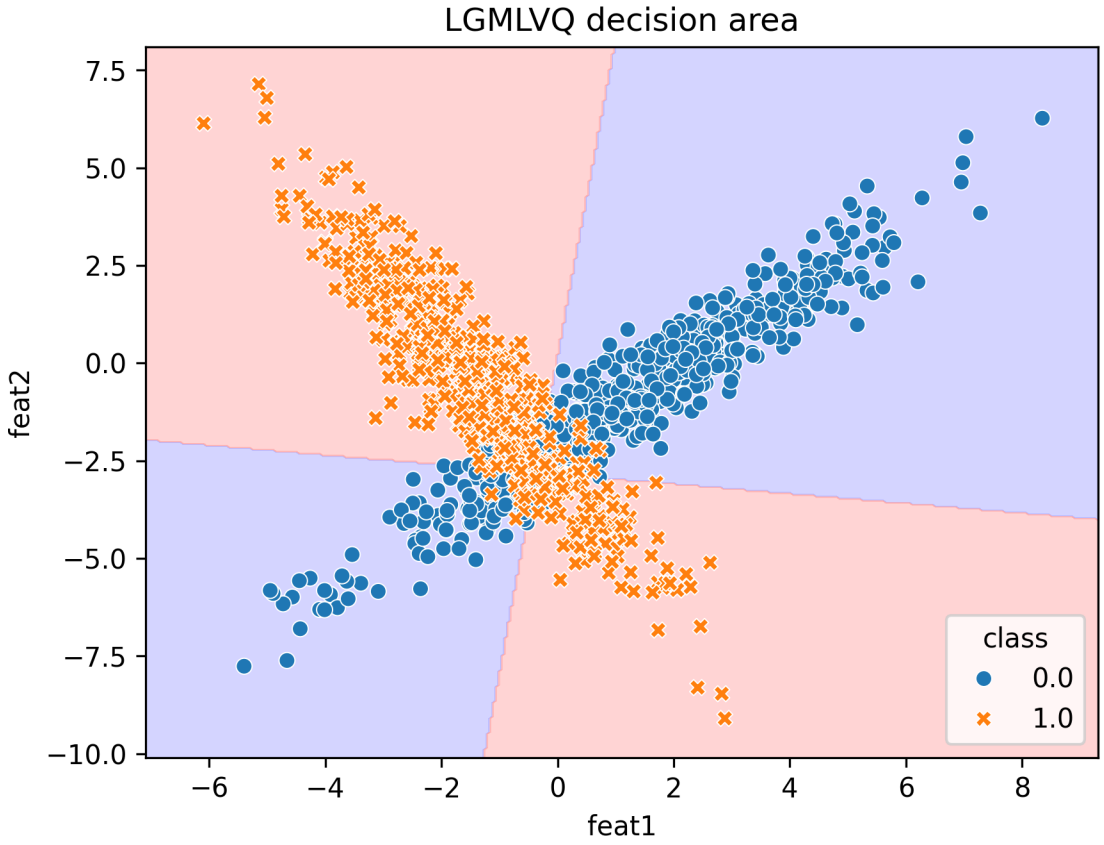
```
n_features = 2 # features
n_categories = 2 # classes

n_prototypes_per_class = [1, 1]
n_prototypes = sum(n_prototypes_per_class)
W = init_prototypes(data, n_prototypes_per_class)
Wclasses = get_wclasses(n_prototypes_per_class)

# init pseudo-relevance matrix
lvq_mode = 'gmlvq' # gmlvq: one matrix, lgmlvq: one matrix per prototype
Q = init_pseudorelevance_matrix(n_prototypes_per_class, n_features, lvq_mode)

eta0 = {'W': 0.005, 'Q': 0.001}
tau = 0.001

epochs = 25

W, Q = train_lvq(data, Wclasses, W, Q, lvq_mode, eta0, tau, epochs)
```

which is pretty close to the one obtained in the paper. If we visualize the results we can see why tho model fails achieve higher accuracy.



As said before (and much better explained in the paper, GMLVQ is at the end a linear classifier)

However, when we allow multiple relavance matrices with LGMLVQ, we obtain 0.91 accuracy and this is what we get.



With these results at hand it seems like we've successfully implemented (L)GMLVQ!

Despite the success (I was very proud when I finally made it work), this is only the beginning, the code in this proof of concept is very rigid and doesn't allow many changes. To properly explore LVQ, we'd need to be able to:

- Use GLVQ or GRLVQ, apart from (L)GMLVQ.
- modify the activation function in the relative distance loss without having to alter the code for derivatives every time.
- modify learning parameters more easily, e.g. choose among different gradient based optimization algorithms.

Fortunately, all that can be done very easily, as we'll see in the next part (click here).

References

Minda, J.P. & Smith, D. (2011). Prototype models of categorization: basic formulation, predictions, and limitations. In E.M. Pothos, N. Chater & P. Hines, *Formal Approaches in Categorization* (pp. 40-65)

Schneider, P., Biehl, M. & Hammer, B. (2009). Adaptive relevance matrices in Learning Vector Quantization, *Neural computation*, 21(12), 3532-3561.