part_5_clustering

February 21, 2022

[1]:
```python
#import numpy as np
import torch as t
from torch.distributions import Normal, Categorical, Bernoulli
from torch.distributions import MultivariateNormal as MvNormal
import matplotlib as mpl
import matplotlib.pyplot as plt
#%matplotlib widget
from ipywidgets import FloatSlider, IntSlider, interact, interact_manual
mpl.rcParams['figure.max_open_warning'] = False
```

Part 5: Unsupervised learning and clustering

Clustering is not classification…

…and unsupervised learning is not supervised learning

Clustering is a type of unsupervised learning, which is very, very different from the supervised learning you have seen in my slides so far.

In supervised learning, we have a list of input, `x`, output, `y`, pairs as data, and the idea is to learn a function that maps from a new input test point, to a distribution over the corresonding `Y`

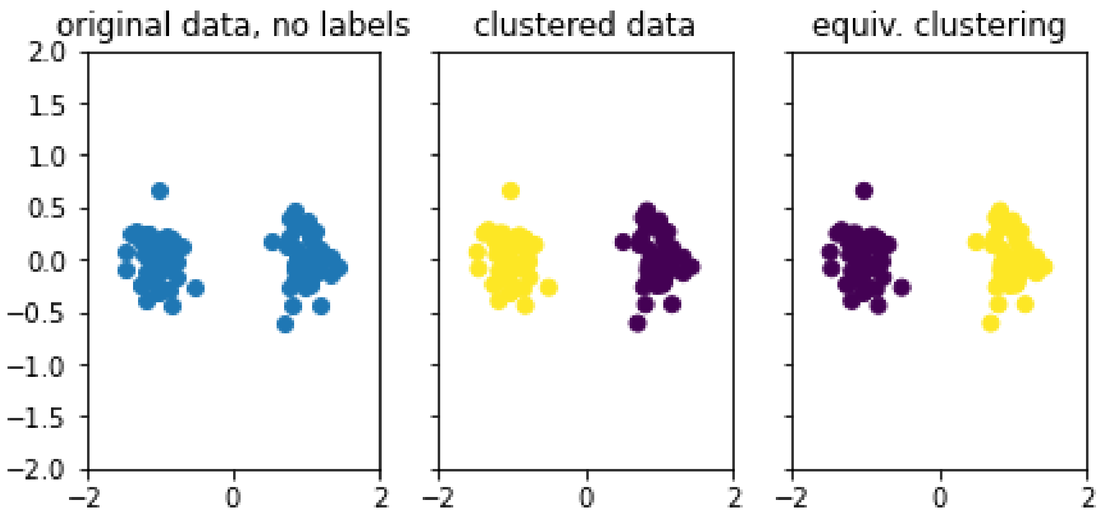
```
#### Supervised learning
x : X # Input
y : Y # Output
    [(X, Y)] -> (X -> Y)
```

In unsupervised learning/clustering, we only have a list of inputs, and the goal is to compute a (distribution over) a latent variable that somehow summarises the structure of the inputs.

```
#### Unsupervised learning
x : X # Input
z : Z # Latent
    [X] -> [Z]
```

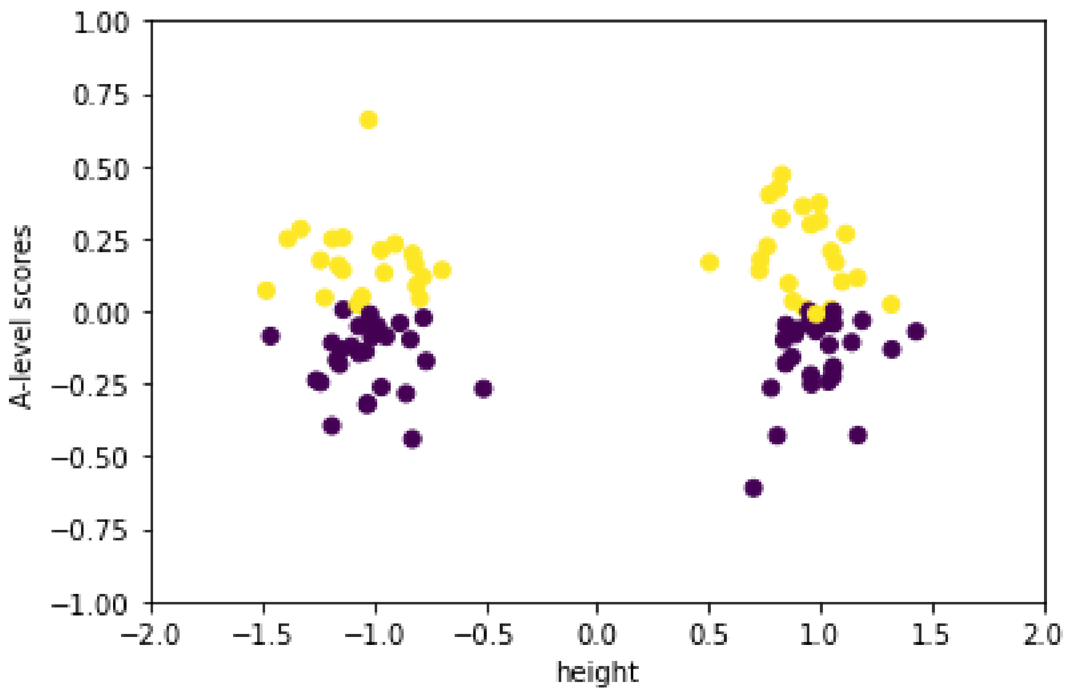
First example

[2]:
```python
X = t.cat([
    t.randn(50, 2)/5 + t.tensor([1., 0.]),
    t.randn(50, 2)/5 + t.tensor([-1., 0.])
])

Z = t.cat([
    t.zeros(50).int(),
    t.ones(50).int()
])

#fig = plot.figure()

fig, axs = plt.subplots(ncols=3, figsize=(7,3), sharey=True)
axs[0].set_xlim(-2, 2)
axs[1].set_xlim(-2, 2)
axs[2].set_xlim(-2, 2)
axs[0].set_ylim(-2, 2)
axs[0].scatter(X[:, 0], X[:, 1])
axs[1].scatter(X[:, 0], X[:, 1], c=Z);
axs[2].scatter(X[:, 0], X[:, 1], c=-Z);
axs[0].set_title("original data, no labels")
axs[1].set_title("clustered data")
axs[2].set_title("equiv. clustering");
```



A classification problem where we need to ignore clusters

[3]:
```
Y = Bernoulli(t.sigmoid(100*X[:, 1])).sample()

fig, ax = plt.subplots()
ax.set_xlim(-2, 2)
ax.set_ylim(-1, 1)
ax.set_xlabel("height")
ax.set_ylabel("A-level scores")
ax.scatter(X[:, 0], X[:, 1], c=Y);
```



Simplest clustering algorithm: K-means

We start by introducing cluster centers,

$$\boldsymbol{\mu} = \text{cluster center, for cluster indexed } z \tag{1}$$

and integer-valued cluster assignments, z_i, describing the cluster associated with the ith datapoint,

$$z_i = \text{integer cluster index for each datapoint, indexed } i \tag{2}$$

The objective is find a bunch of cluster centers such that all datapoints are close to a cluster center. Formally, we minimise the squared distance between each datapoint and its assigned cluster center,

$$\mathcal{L}(\mathbf{z}, \boldsymbol{\mu}) = \sum_i \left\| \mathbf{x}_i - \boldsymbol{\mu}_{z_i} \right\|^2 \tag{3}$$

To optimize this algorithm, we use coordinate descent (i.e. we alternate between optimizing $\mathbf{z}$ and $\boldsymbol{\mu}$).

First, we assign each datapoint to the nearest cluster,

$$z_i \leftarrow \text{argmin}_{z\in\{1\ldots Z\}} \left\| \mathbf{x}_i - \boldsymbol{\mu}_z \right\|^2 \tag{4}$$

Then, we put the cluster centers at the mean of the assigned datapoints,

$$\boldsymbol{\mu}_z \leftarrow \frac{1}{\sum_i \delta_{z,z_i}} \sum_i \delta_{z,z_i} \mathbf{x}_i \tag{5}$$

remember that z is an index, whereas z_i is the actual cluster assignment for datapoint i.

$$\delta_{z,z_i} = \begin{cases} 1 & \text{if } z = z_i\text{, i.e. datapoint } i \text{ is currently assigned to the } z\text{th cluster} \\ 0 & \text{if } z \neq z_i\text{, i.e. datapoint } i \text{ is not in the } z\text{th cluster} \end{cases} \tag{6}$$

so,

$$\sum_i \delta_{z,z_i} = \text{number of datapoints in cluster } z \sum_i \mathbf{x}_i \delta_{z,z_i} = \text{sum of all the datapoints in cluster } z \tag{7}$$

[4]:
```
# N == number of datapoints
# K == number of clusters
# D == dimension of the data vectors (usually 2 in our examples)
#X.shape  == (N, 1, D) # Data
#mu.shape ==    (K, D) # Cluster-centers
#q.shape  == (N, K, 1) # One-hot representation of the cluster-assignments for␣
 ↪each datapoint
#z.shape  == (N,)      # Cluster index for each datapoint

t.manual_seed(0)

X = t.cat([
    t.randn(50, 2)/5 + t.tensor([1., 0.]),
    t.randn(50, 2)/5,
    t.randn(50, 2)/5 + t.tensor([-1., 0.])
])

def kmeans(X, K):
    N, D = X.shape
    X = X[:, None, :]
    mu = t.randn(K, D)
    while True:
        sd = ((X - mu)**2).sum(-1) # sd.shape = (N, K)
        z = t.argmin(sd, 1)        # z.shape  = (N)
        q = t.zeros(N, K, 1)
        q[t.arange(N), z, 0] = 1.
        print(f"loss = {loss(X, z, mu).item()}")
        plot(X, z, mu)
        yield None
        mu = (q * X).sum(0) / q.sum(0)
        print(f"loss = {loss(X, z, mu).item()}")
        plot(X, z, mu)
        yield None

def loss(X, z, mu):
    return ((X - mu[z, :][:, None, :])**2).mean()

def plot(X, z, mu):
    fig, ax = plt.subplots()
    ax.set_xlim(-2, 2)
    ax.set_ylim(-2, 2)
    ax.scatter(X[:, 0, 0], X[:, 0, 1], c=z);
    ax.scatter(mu[:, 0], mu[:, 1], s=100, c='r', label="cluster centers")
    ax.legend()

kmeans_iter = iter(kmeans(X, 3))
def kmeans_call():
    next(kmeans_iter)
interact_manual(kmeans_call);

#Change marker type
```

```
interactive(children=(Button(description='Run Interact', style=ButtonStyle()),␣
↪Output()), _dom_classes=('widge…
```

However, one strange property of this algorithm is that it assumes “hard” cluster assignments.

Soft clustering and the Gaussian Mixture Model (GMM)

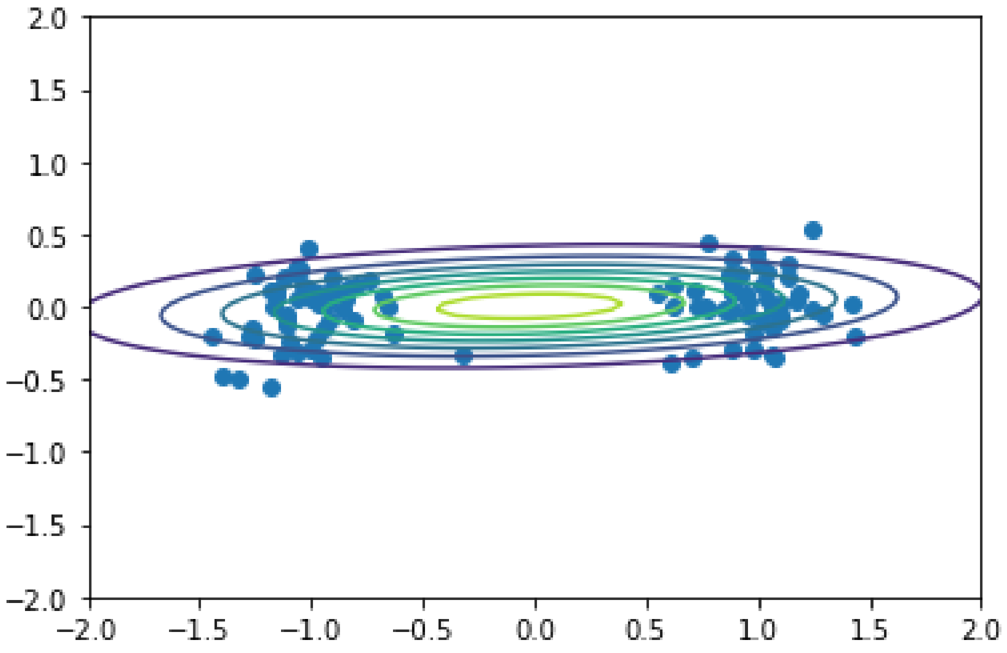
The goal of the GMM, is really just to model the density of the data we use the latent variables to give us a better model of the data.

For instance, consider fitting a MvNormal to the following data,

[5]:
```
X = t.cat([
    t.randn(50, 2)/5 + t.tensor([1., 0.]),
    t.randn(50, 2)/5 + t.tensor([-1., 0.])
])

N = X.shape[0]
mu = X.sum(0)/N
cov = ((X - mu).T @ (X-mu))/N
mvn = MvNormal(mu, cov)

logPx = mvn.log_prob(X).sum()
print(f"logPx = {logPx}")

P = 100
```

```
x0s = t.linspace(-2, 2, P)[:, None].expand(P, P)
x1s = x0s.T
xs  = t.stack([x0s.reshape(-1), x1s.reshape(-1)], 1)
ps = mvn.log_prob(xs).exp().view(P, P)

fig, ax = plt.subplots()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.scatter(X[:, 0], X[:, 1]);
ax.contour(x0s, x1s, ps);
```

```
logPx = -127.18228912353516
```



How do we get a better model for this kind of data? We use a GMM.

The simplest form of a GMM is a density, with K different Gaussians, in different locations, $\boldsymbol{\mu}_k$, and with different covariance matrices, $_k$,

$$\mathrm{P}(\mathbf{x}_i) = \sum_{z=1}^{Z} p_z \mathcal{N}\left(\mathbf{x}_i; \boldsymbol{\mu}_z, {}_z\right). \tag{8}$$

However, this doesn't give us a way to extract any notion of assignment of a datapoint to a mixture component.

We therefore write this distribution in terms of a latent variable,

$$\mathrm{P}(\mathbf{x}_i) = \sum_{z_i=1}^{Z} \mathrm{P}(\mathbf{x}_i|z_i)\,\mathrm{P}(z_i) \tag{9}$$

where, $\mathrm{P}(z_i)$ is a Categorical (i.e. a distribution over integers from 1 to Z), and represents the mixture component for the ith data point, and $\mathrm{P}(x_i|z_i)$ is a *single* Gaussian, (corresponding to the z_ith mixture component),

$$\begin{aligned} \mathrm{P}(z_i) &= \mathrm{Categorical}\left(z_i; \mathbf{p}\right) = p_{z_i} && (10) \\ \mathrm{P}(\mathbf{x}_i|z_i) &= \mathcal{N}\left(\mathbf{x}_i; \boldsymbol{\mu}_{z_i},\ {}_{z_i}\right). && (11) \end{aligned}$$

The substituting in the values of $\mathrm{P}(z_i)$ and $\mathrm{P}(x_i|z_i)$, we get the same expression as above, except use z_i as the summation index, instead of z

$$\mathrm{P}(\mathbf{x}_i) = \sum_{z_i=1}^{Z} \mathrm{P}(\mathbf{x}_i|z_i)\,\mathrm{P}(z_i) = \sum_{z_i=1}^{Z} p_{z_i} \mathcal{N}\left(\mathbf{x}_i; \boldsymbol{\mu}_{z_i},\ {}_{z_i}\right). \tag{12}$$

Now, we can use Bayes theorem to give a posterior distribution over z_i,

$$\mathrm{P}(z_i|\mathbf{x}_i) \propto \mathrm{P}(\mathbf{x}_i|z_i)\,\mathrm{P}(z_i) \tag{13}$$

Suggests an algorithm similar to K-means: first compute the posterior over z_i (E-step), then update the parameters of the Gaussians, $\boldsymbol{\mu}_z$, using a mean weighted by the posterior (M-step).

In particular, the **E-step** updates the posterior over z_i for a given x_i, and "saves" the posterior into $\mathbf{q}$. The denominator normalizes the distribution so that it sums to 1.

$$q_{i;z} \leftarrow \mathrm{P}(z_i = z|x_i) = \frac{\mathrm{P}(\mathbf{x}_i|z_i = z)\,\mathrm{P}(z_i = z)}{\sum_{z'} \mathrm{P}(\mathbf{x}_i|z_i = z')\,\mathrm{P}(z_i = z')} \tag{14}$$

and the **M-step**, is a weighted average of $\mathbf{x}_i$'s, with the weights corresponding to the degree to which that datapoint is assigned to that cluster,

$$\boldsymbol{\mu}_z \leftarrow \frac{\sum_i \mathbf{x}_i q_{i,z}}{\sum_i q_{i,z}} \tag{15}$$

In the case where we are very sure about the cluster assignments, the posteriors become equal to the hard-assignment case, because $\mathrm{P}(z_i = z|x_i)$ is 1 for the "right" assignment and 0 otherwise,

$$q_{i;z} = \mathrm{P}(z_i = z|x_i) = \delta_{z_i,z} \tag{16}$$

[6]:
```
t.manual_seed(0)

X = t.cat([
    t.randn(50, 2)/3 + t.tensor([1., 0.]),
    t.randn(50, 2)/3 + t.tensor([-1., 0.])
])

def gmm_em(X, K):
    N, D = X.shape
    X = X[:, None, :]
    mu  = t.randn(K, D)
    cov = 0.5*t.eye(D, D).expand(K, -1, -1)
    while True:
        # unnormalized posterior probability
        q = MvNormal(mu, cov).log_prob(X).exp()
        # normalized posterior probability
        q = q / q.sum(1, keepdim=True)
        # expand out last such that q.shape = (N, K, 1)
        q = q[:, :,  None]
        plot_gmm(X, q, mu, cov)
        yield None

        #weighted mean
        mu = (q * X).sum(0) / q.sum(0)
        plot_gmm(X, q, mu, cov)
        yield None


def plot_gmm(X, q, mu, cov):
    fig, ax = plt.subplots()
    ax.set_xlim(-2, 2)
    ax.set_ylim(-2, 2)
    ax.scatter(X[:, 0, 0], X[:, 0, 1], c=q[:, 0, 0]);
    ax.scatter(mu[:, 0], mu[:, 1], s=100, c='r', label="cluster centers")
    ax.legend()


gmm_em_iter = iter(gmm_em(X, 2))
def gmm_em_call():
    next(gmm_em_iter)
interact_manual(gmm_em_call);

#Change marker type
```

```
interactive(children=(Button(description='Run Interact', style=ButtonStyle()),␣
↪Output()), _dom_classes=('widge…
```

What is the objective function for EM? [Non-examinable]

Turns out that this is a very subtle question.

We need to define an "approximate posterior", corresponding to the current cluster assignments,

$$\mathrm{Q}_{\mathbf{q}}(z_i) = \text{Categorical}\left(z_i; \mathbf{q}_i\right) = q_{i,z_i} \tag{17}$$

We can write down the "evidence lower-bound objective" (ELBO). Note that the "model evidence" is $\log \mathrm{P}(\mathbf{x})$, and the ELBO can be defined as,

$$\log \mathrm{P}_{\boldsymbol{\mu}}(\mathbf{x}) \geq \mathcal{L}\left(\boldsymbol{\mu}, \mathbf{q}\right) = \log \mathrm{P}_{\boldsymbol{\mu}}(\mathbf{x}) - D_{\mathrm{KL}}\left(\mathrm{Q}_{\mathbf{q}}(\mathbf{z}) \| \mathrm{P}_{\boldsymbol{\mu}}(\mathbf{z}|\mathbf{x})\right) \tag{18}$$

Where the ELBO is a lower-bound because the KL-divergence is non-negative,

$$0 \leq D_{\mathrm{KL}}\left(\mathrm{Q}_{q}(\mathbf{z}) \| \mathrm{P}_{\boldsymbol{\mu}}(\mathbf{z}|\mathbf{x})\right) = \mathbb{E}_{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}\left[\log \frac{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}{\mathrm{P}_{\boldsymbol{\mu}}(\mathbf{z}|\mathbf{x})}\right] \tag{19}$$

And the KL-divergence equals zero when the approximate posterior equals the true posterior, $\mathrm{Q}(\mathbf{z}) = \mathrm{P}(\mathbf{z}|\mathbf{x})$.

Expectation (E) step

Thus, the E-step consists of maximizing the ELBO wrt to the parameters of the approximate posterior, $\mathbf{q}$.

$$\mathbf{q} \leftarrow \mathrm{argmax}_{\mathbf{q}} \mathcal{L}\left(\boldsymbol{\mu}, \mathbf{q}\right) \tag{20}$$

as $\log \mathrm{P}_{\boldsymbol{\mu}}(\mathbf{x})$ does not depend on $\mathbf{q}$ this is equivalent to,

$$\mathbf{q} \leftarrow \mathrm{argmax}_{\mathbf{q}} D_{\mathrm{KL}}\left(\mathrm{Q}_{\mathbf{q}}(\mathbf{z}) \| \mathrm{P}_{\boldsymbol{\mu}}(\mathbf{z}|\mathbf{x})\right) \tag{21}$$

This KL-divergence is minimized by setting $\mathbf{q}$ to the true posterior (i.e. the E-step that we saw above), concretely, we can achieve this by setting,

$$q_{i,z} \leftarrow \mathrm{P}\left(z_i = z|\mathbf{x}_i\right). \tag{22}$$

Maximization (M) step

The goal is,

$$\boldsymbol{\mu} \rightarrow \mathrm{argmax}_{\boldsymbol{\mu}} \mathcal{L}\left(\boldsymbol{\mu}, \mathbf{q}\right) \tag{23}$$

Both terms in the previous form for the ELBO depend on $\boldsymbol{\mu}$,

$$\mathcal{L}\left(\boldsymbol{\mu}, \mathbf{q}\right) = \log \mathrm{P}_{\boldsymbol{\mu}}(\mathbf{x}) + D_{\mathrm{KL}}\left(\mathrm{Q}_{\mathbf{q}}(\mathbf{z}) \| \mathrm{P}_{\boldsymbol{\mu}}(\mathbf{z}|\mathbf{x})\right) \tag{24}$$

so we need to rearrange to get a good update for $\boldsymbol{\mu}$.

In particular,

$$\begin{align}
\mathcal{L}\left(\boldsymbol{\mu}, \mathbf{q}\right) &= \log \mathrm{P}_{\boldsymbol{\mu}}(\mathbf{x}) - \mathbb{E}_{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}\left[\log \frac{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}{\mathrm{P}_{\boldsymbol{\mu}}(\mathbf{z}|\mathbf{x})}\right] \tag{25}\\
\mathcal{L}\left(\boldsymbol{\mu}, \mathbf{q}\right) &= \log \mathrm{P}_{\boldsymbol{\mu}}(\mathbf{x}) + \mathbb{E}_{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}\left[\log \frac{\mathrm{P}_{\boldsymbol{\mu}}(\mathbf{z}|\mathbf{x})}{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}\right] \tag{26}\\
\mathcal{L}\left(\boldsymbol{\mu}, \mathbf{q}\right) &= \mathbb{E}_{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}\left[\log \mathrm{P}_{\boldsymbol{\mu}}(\mathbf{x}) + \log \frac{\mathrm{P}_{\boldsymbol{\mu}}(\mathbf{z}|\mathbf{x})}{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}\right] \tag{27}\\
\mathcal{L}\left(\boldsymbol{\mu}, \mathbf{q}\right) &= \mathbb{E}_{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}\left[\log \frac{\mathrm{P}_{\boldsymbol{\mu}}(\mathbf{z}|\mathbf{x})\, \mathrm{P}_{\boldsymbol{\mu}}(\mathbf{x})}{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}\right] \tag{28}\\
\mathcal{L}\left(\boldsymbol{\mu}, \mathbf{q}\right) &= \mathbb{E}_{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}\left[\log \frac{\mathrm{P}_{\boldsymbol{\mu}}(\mathbf{x}, \mathbf{z})}{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}\right] \tag{29}\\
\mathcal{L}\left(\boldsymbol{\mu}, \mathbf{q}\right) &= \mathbb{E}_{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}[\log \mathrm{P}_{\boldsymbol{\mu}}(\mathbf{x}, \mathbf{z})] - \mathbb{E}_{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}[\mathrm{Q}_{\mathbf{q}}(\mathbf{z})] \tag{30}\\
& \tag{31}
\end{align}$$

Note that the second term doesn't depend on $\boldsymbol{\mu}$, so,

$$\boldsymbol{\mu} \to \mathrm{argmax}_{\boldsymbol{\mu}} \mathbb{E}_{\mathrm{Q}_{\mathbf{q}}(\mathbf{z})}[\log \mathrm{P}_{\boldsymbol{\mu}}(\mathbf{x}, \mathbf{z})] \tag{32}$$

And this is just maximum-likelihood fitting of the parameters of the mixtures (here, the mixture-means), with datapoints weighted by $\mathrm{Q}(\mathbf{z})$.