Adv ML for Gen-AI

Gaussian Mixture Model

https://ml-graph.github.io/spring-2026/

Yu Wang, Ph.D.
Assistant Professor
Department of Computer Science
University of Oregon
Personal: https://yuwang0103.github.io/
Lab: https://kindlab-fly.github.io/

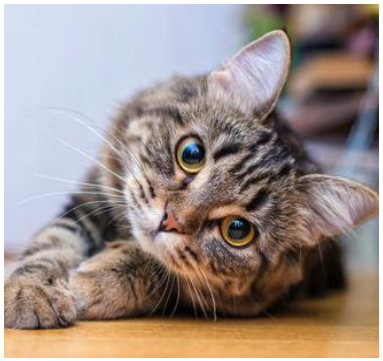
Data Distribution

Dog





Cat


Data Distribution

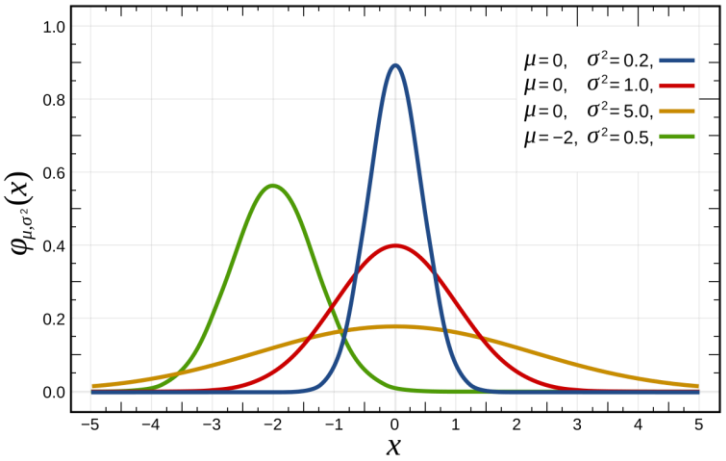
Dog – P(Dog)


Cat – P(Cat)


1. There is no concrete image/shape of the dog, everyone can come up with one of your own choice

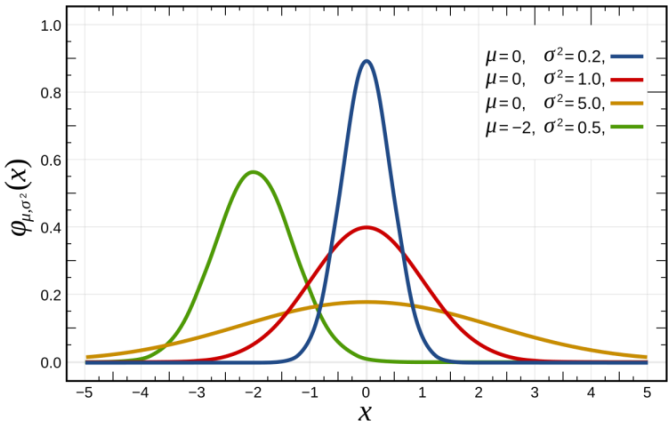
2. But somehow dog and cat image distributions are different

When you draw an image, you are actually sampling from a probability distribution!



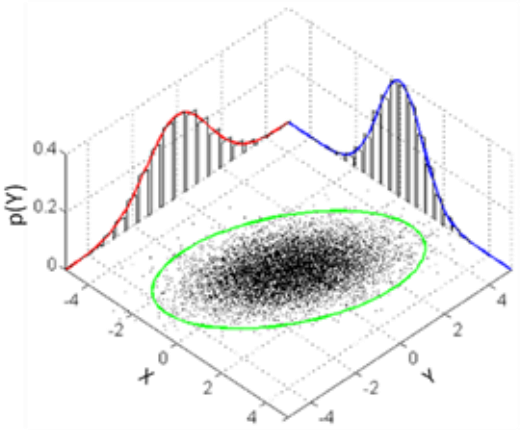
Data Distribution

1D Gaussian Distribution



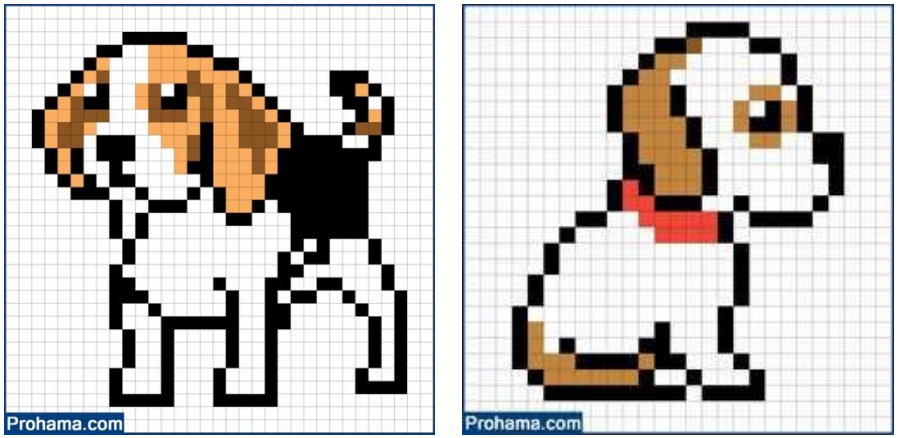
$\mathbb{R}$

2D Gaussian Distribution



$\mathbb{R}^2$

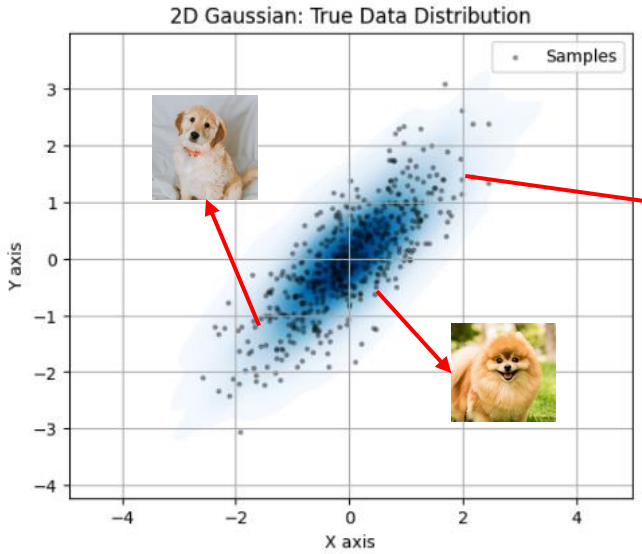
$\mathbb{R}^{256\times256}$



$\mathbb{R}^{256\times256}$

Estimating Data Distribution and Sampling from it

Estimate Data Distribution



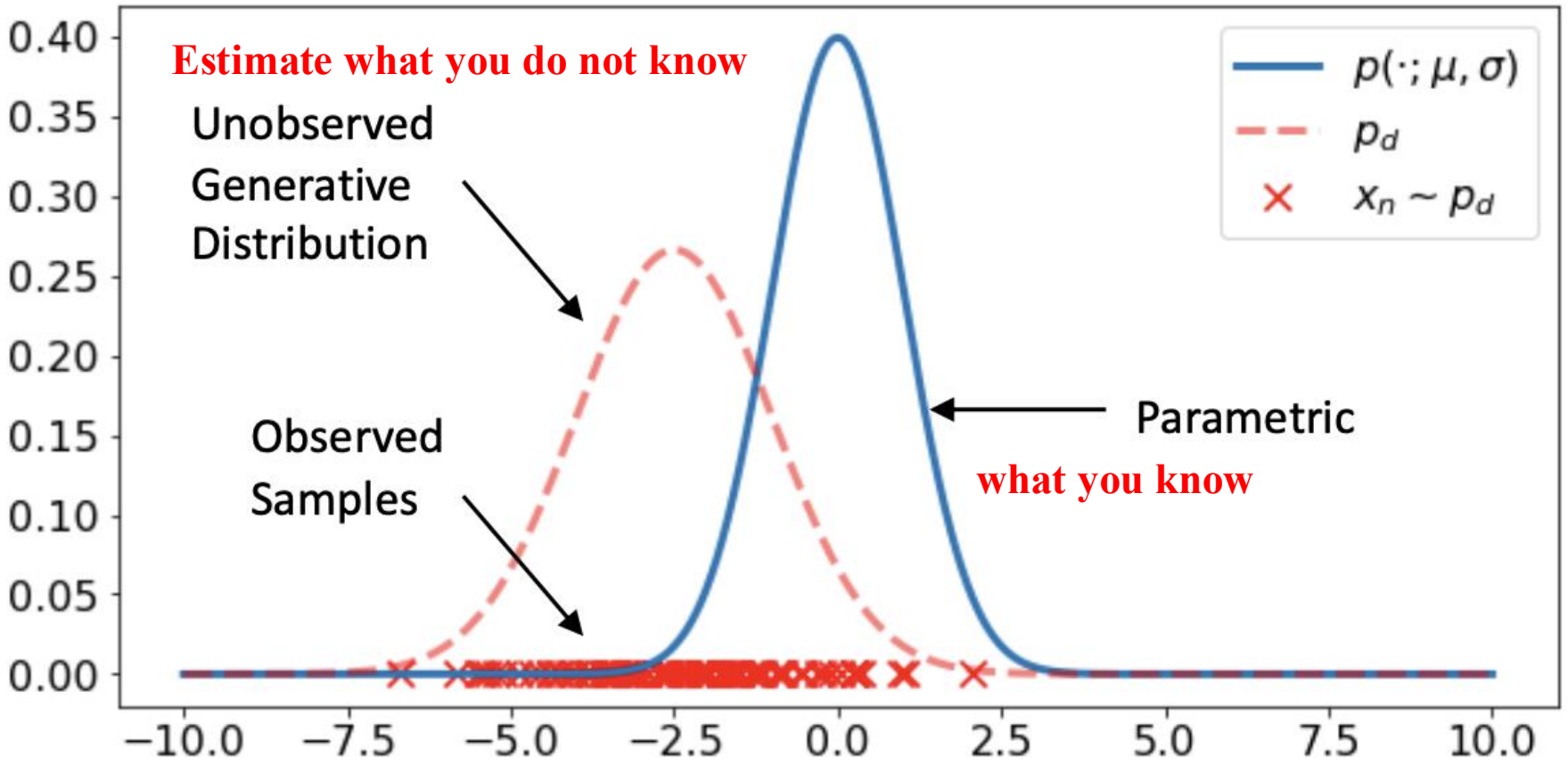
Sampling from it

Outline

- **Foundation of Gen-ML**
  - Distribution
  - Estimation and Sampling
- **1 Dimension Gaussian Generation**
  - Gaussian Distribution
  - Likelihood Function
  - Maximum Likelihood Optimization (Theoretical/Computational Solution)
- **Gaussian Kernal Density Estimation**
  - Parametric vs Nonparametric Distribution
  - Dirac to Gaussian
  - Maximum Likelihood Optimization on Gaussian KDE
  - Leave-one-out Estimation on Gaussian KDE
- **2D Dimension Gaussian Generation**
  - PDF and Sampling
  - Maximum Likelihood Optimization
  - Application to Digit Image Generation

1D Gaussian Estimation

Using existing function to estimate what you do not know that can best fit your observation



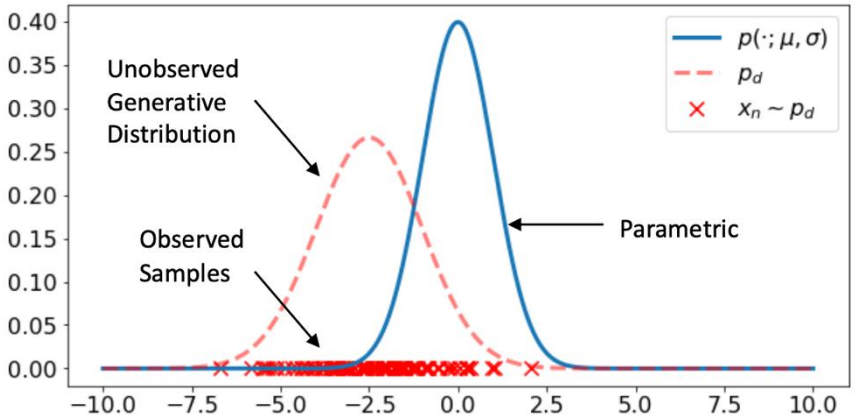
1D Gaussian Estimation

Using existing function to estimate what you do not know that can best fit your observation

$$\{x_i | x_i \sim P_d\}_{i=1}^{N}$$

$$p(x;\mu,\sigma) = N(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

What is μ, σ ?



1- Maximum Likelihood:

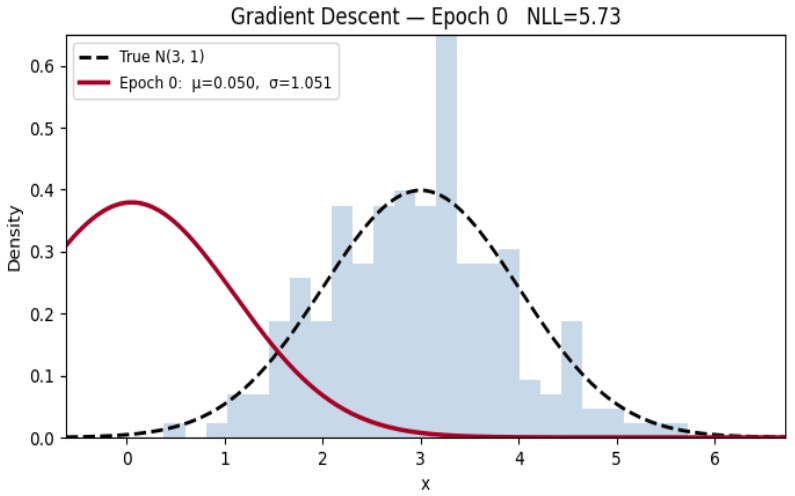
$$\underset{\mu,\sigma}{\operatorname{argmax}}\, p(x_1, \ldots, x_N; \mu, \sigma) = \prod_{n=1}^{N} p(x_n; \mu, \sigma)$$

1D Gaussian Estimation

```python
import torch
import numpy as np

x = torch.tensor(data, dtype=torch.float32)

# ── Initialization ──────────────────────────────────────────────
mu        = torch.tensor(0.0, requires_grad=True)   # bad starting guess
log_sigma = torch.tensor(0.0, requires_grad=True)  # optimise log σ for stability

optimizer = torch.optim.Adam([mu, log_sigma], lr=0.05)

n_epochs     = 300
record_every = 30
history      = []   # (epoch, mu, sigma, nll)

# ── Training loop ───────────────────────────────────────────────
for epoch in range(n_epochs):
    sigma = torch.exp(log_sigma)
    nll   = (
        0.5 * torch.log(2 * torch.tensor(np.pi))
        + log_sigma
        + 0.5 * ((x - mu) ** 2 / sigma ** 2).mean()
    )
    optimizer.zero_grad()
    nll.backward()
    optimizer.step()

    if epoch % record_every == 0 or epoch == n_epochs - 1:
        history.append((epoch, mu.item(), torch.exp(log_sigma).item(), nll.item()))

# ── Training log ────────────────────────────────────────────────
print(f"{'Epoch':>6}  {'μ (mu)':>10}  {'σ (sigma)':>10}  {'NLL':>10}")
print("-" * 44)
for epoch, mu_val, sigma_val, nll_val in history:
    print(f"{epoch:>6}  {mu_val:>10.4f}  {sigma_val:>10.4f}  {nll_val:>10.4f}")

print(f"\nLearned  μ = {history[-1][1]:.4f},  σ = {history[-1][2]:.4f}")
print(f"True     μ = 3.0000,  σ = 1.0000")
```

Epoch	μ (mu)	σ (sigma)	NLL
0	0.0500	1.0513	5.7287
30	1.1211	2.4646	2.1749
60	1.6300	2.1697	1.9816
90	2.1959	1.5625	1.6727
120	2.8080	1.0914	1.3831
150	2.9932	0.9118	1.3460
180	2.9524	0.9273	1.3450
210	2.9605	0.9297	1.3449
240	2.9590	0.9284	1.3449
270	2.9593	0.9287	1.3449
299	2.9592	0.9287	1.3449

Learned μ = 2.9592, σ = 0.9287
True μ = 3.0000, σ = 1.0000

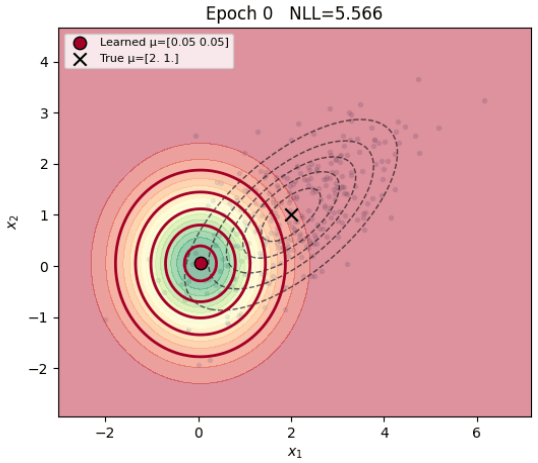


1D Gaussian Estimation

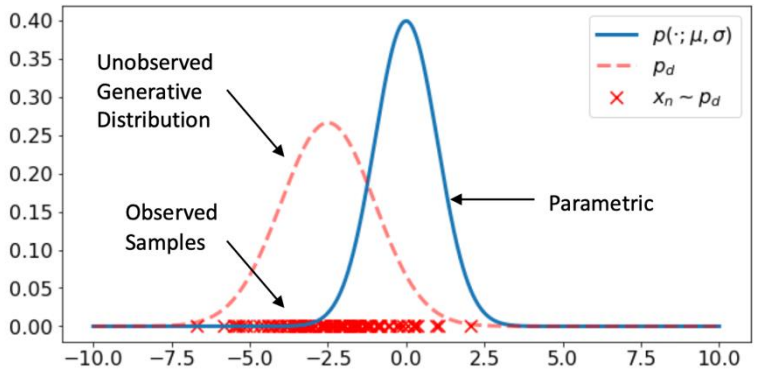
```python
import torch

X2d = torch.tensor(data_2d, dtype=torch.float32)   # (N, 2)

# ── Initialization – bad spherical guess ──────────────────────────
mu2d = torch.tensor([0.0, 0.0], requires_grad=True)

# Cholesky factor L: lower-triangular with positive diagonal (via log)
L_diag = torch.tensor([0.0, 0.0], requires_grad=True)   # log of diagonal entries
L_off  = torch.tensor([0.0],      requires_grad=True)   # single off-diagonal entry

def build_L():
    """Reconstruct lower-triangular Cholesky factor."""
    L = torch.zeros(2, 2)
    L[0, 0] = torch.exp(L_diag[0])
    L[1, 1] = torch.exp(L_diag[1])
    L[1, 0] = L_off[0]
    return L

optimizer2d = torch.optim.Adam([mu2d, L_diag, L_off], lr=0.05)

n_epochs2d     = 400
record_every2d = 40
history2d      = []   # (epoch, mu_np, cov_np, nll)

# ── Training loop ─────────────────────────────────────────────────
for epoch in range(n_epochs2d):
    L   = build_L()
    Sig = L @ L.t()                      # Σ = L Lᵀ  (always PSD)
    Sig_inv = torch.linalg.inv(Sig)
    log_det = 2 * L_diag.sum()           # log|Σ| = 2 * sum(log diag L)

    diff = X2d - mu2d                    # (N, 2)
    mahal = (diff @ Sig_inv * diff).sum(dim=1)  # (N,)
    nll2d = 0.5 * (2 * torch.log(torch.tensor(2 * torch.pi)) + log_det + mahal).mean()

    optimizer2d.zero_grad()
    nll2d.backward()
    optimizer2d.step()

    if epoch % record_every2d == 0 or epoch == n_epochs2d - 1:
        history2d.append((
            epoch,
            mu2d.detach().numpy().copy(),
            Sig.detach().numpy().copy(),
            nll2d.item()
        ))
```

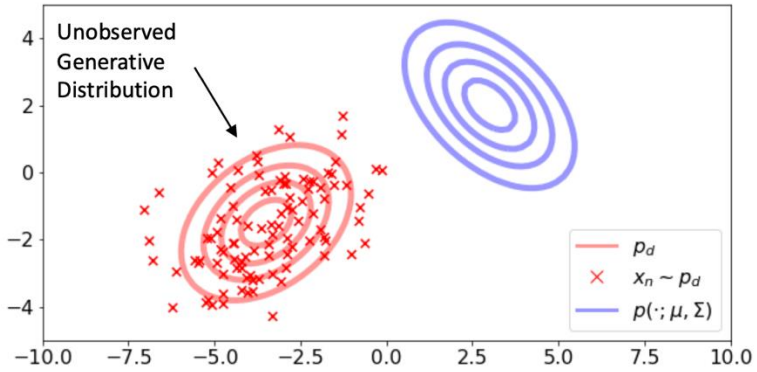
Epoch	μ	Σ diag	NLL
0	[0.05 0.05]	[1. 1.]	5.5661
40	[1.344 0.587]	[3.043 1.454]	2.8954
80	[1.939 0.958]	[1.604 0.971]	2.7095
120	[2.031 1.008]	[1.435 0.931]	2.7043
160	[2.017 1.]	[1.441 0.938]	2.7042
200	[2.019 1.001]	[1.442 0.938]	2.7042
240	[2.019 1.001]	[1.441 0.938]	2.7042
280	[2.019 1.001]	[1.442 0.938]	2.7042
320	[2.019 1.001]	[1.442 0.938]	2.7042
360	[2.019 1.001]	[1.442 0.938]	2.7042
399	[2.019 1.001]	[1.442 0.938]	2.7042

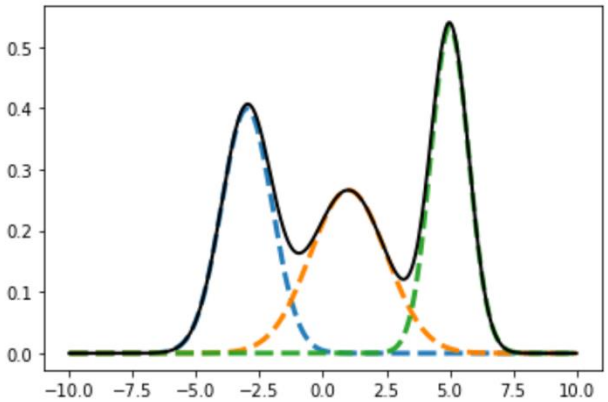
```
Learned  μ = [2.019 1.001]
True     μ = [2. 1.]

Learned  Σ =
[[1.442 0.766]
 [0.766 0.938]]
True     Σ =
[[1.5 0.8]
 [0.8 1. ]]
```

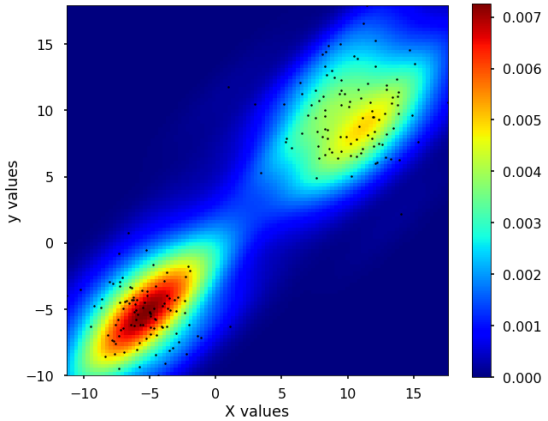


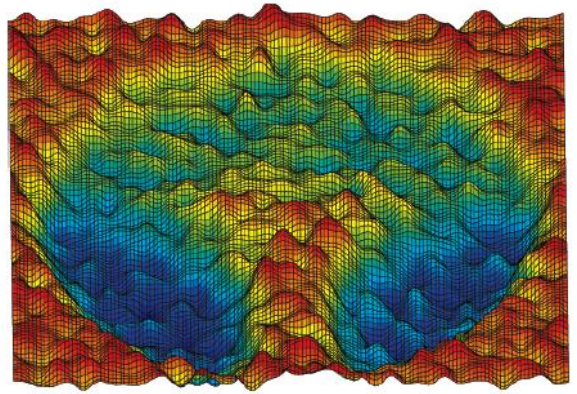
Problem with only using 1D 1 Gaussian









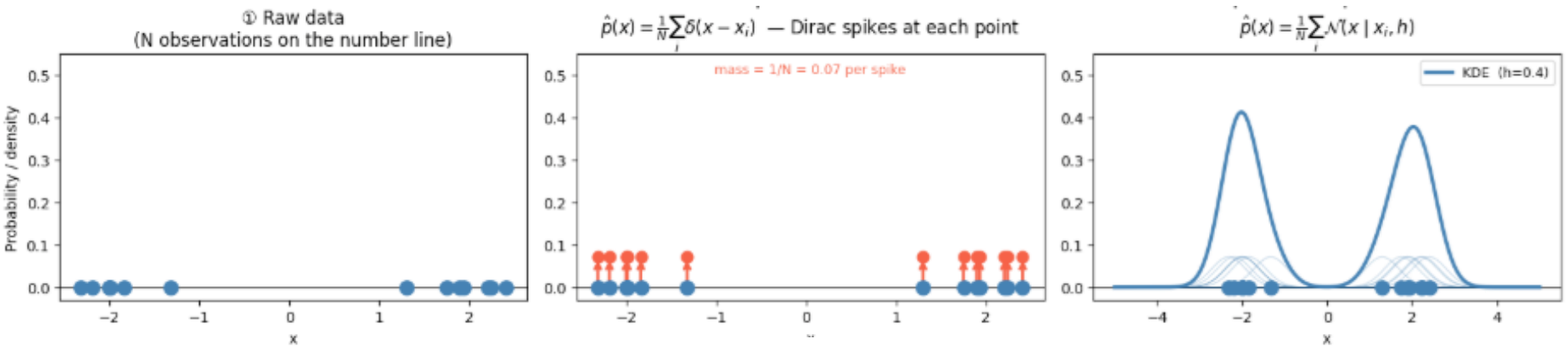


If using one existing function to estimate is not enough, then let's try more!

Outline

- **Foundation of Gen-ML**
  - Distribution
  - Estimation and Sampling
- **1 Dimension Gaussian Generation**
  - Gaussian Distribution
  - Likelihood Function
  - Maximum Likelihood Optimization (Theoretical/Computational Solution)
- **Gaussian Kernal Density Estimation**
  - Parametric vs Nonparametric Distribution
  - Dirac to Gaussian
  - Maximum Likelihood Optimization on Gaussian KDE
  - Leave-one-out Estimation on Gaussian KDE
- **Gaussian Mixture Model (GMM)**
  - PDF and Format
  - Sampling from GMM
  - Gradient Descent and E-M Algorithm

Gaussian KDE



$$\hat{p}_{\text{empirical}}(x) = \frac{1}{N}\sum_{i=1}^{N}\delta(x - x_i)$$

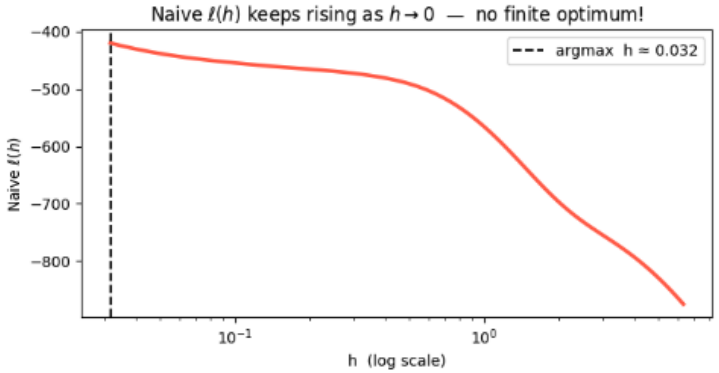
$$\hat{p}(x) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{N}(x \mid x_i,\, h)$$

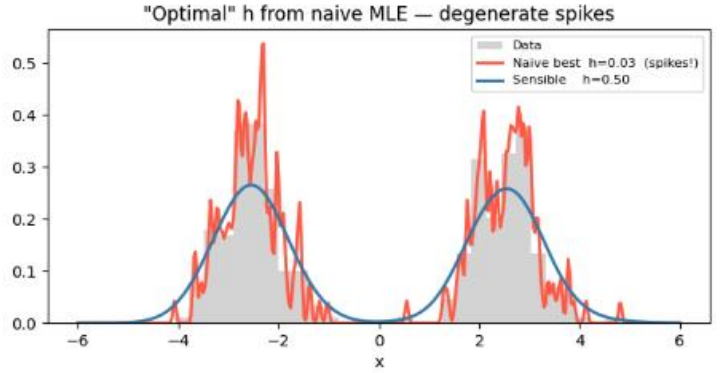
$$x = x_i, \delta = 1$$
$$x \neq x_i, \delta = 0$$

$$\hat{p}(x) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{h\sqrt{2\pi}}\exp\left(-\frac{(x - x_i)^2}{2h^2}\right)$$

Limit	Kernel shape	KDE result
$h \to 0$	$\mathcal{N}(x \mid x_i, h) \to \delta(x - x_i)$	Recovers the spiky empirical distribution
$h \to \infty$	Kernel flattens to near-zero everywhere	Density smears out, all structure lost

Gaussian KDE – Maximum Likelihood





$$\hat{p}(x) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{N}(x \mid x_i,\, h)$$

$$\hat{p}(x) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{h\sqrt{2\pi}} \exp\left(-\frac{(x - x_i)^2}{2h^2}\right)$$

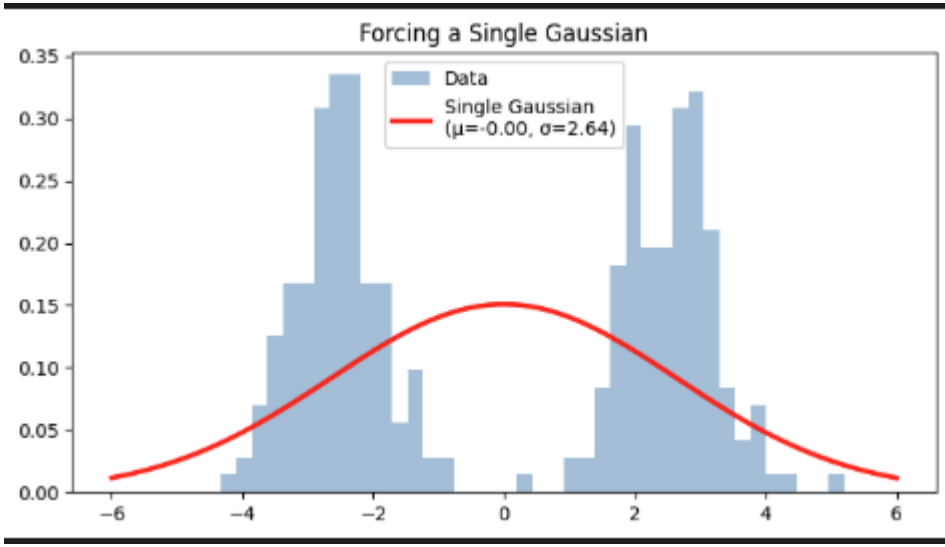
$$\ell_{\text{naive}}(h) = \sum_{i=1}^{N} \log \hat{p}(x_i) = \sum_{i=1}^{N} \log \left[\frac{1}{N} \sum_{j=1}^{N} \mathcal{N}(x_i \mid x_j,\, h) \right]$$

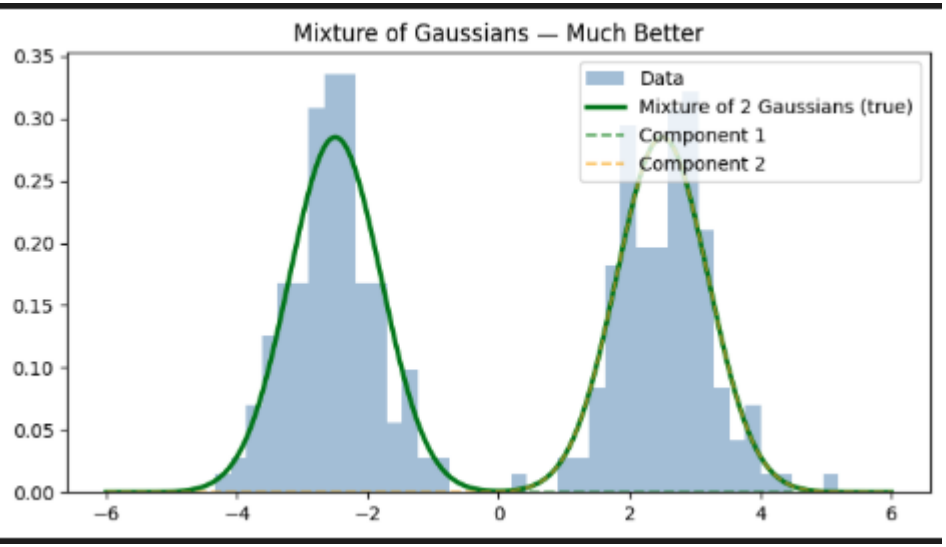
Gaussian KDE – Parametric vs Non-parametric

	Parametric	Non-parametric
Core idea	Assume a fixed functional form for $p(x)$	Make no assumption about the shape of $p(x)$
Parameters	Fixed, finite number (e.g. μ, σ)	Grows with the data (e.g. N kernels in KDE)
Fitting	MLE, gradient descent	Data-driven rules (e.g. Silverman's rule)
Risk	Wrong if assumed shape doesn't match reality	Needs more data; harder to interpret
Conventional examples	Gaussian, GMM, Linear Regression, Logistic Regression, Neural Networks	KDE, k-NN, Empirical Distribution
In Generative AI	VAE, GAN, Diffusion models	KDE — the simplest non-parametric generative model

Parametric vs Non-Parametric

Gaussian Mixture Models - PDF





$$p(x) = \sum_{k=1}^{K} \pi_k \, \mathcal{N}(x \mid \mu_k, \sigma_k)$$

Symbol	Name	Meaning
π_k	**mixing weight**	How often this component is chosen. Must satisfy $\pi_k \geq 0$ and $\sum_k \pi_k = 1$.
μ_k	**mean**	Where the k-th Gaussian is centered.
σ_k	**standard deviation**	How wide the k-th Gaussian is.

Gaussian Mixture Models – Sampling/Generation

$$p(x) = \sum_{k=1}^{K} \pi_k \, \mathcal{N}(x \mid \mu_k, \sigma_k) \qquad \sum_k \pi_k = 1$$

Symbol	Name	Meaning
π_k	**mixing weight**	How often this component is chosen. Must satisfy $\pi_k \geq 0$ and $\sum_k \pi_k = 1$.
μ_k	**mean**	Where the k-th Gaussian is centered.
σ_k	**standard deviation**	How wide the k-th Gaussian is.

Because the GMM has a clean generative story, sampling is easy:

1. Draw a component index $z \in \{1, \ldots, K\}$ with probabilities $[\pi_1, \ldots, \pi_K]$.
2. Draw $x \sim \mathcal{N}(\mu_z, \sigma_z)$.

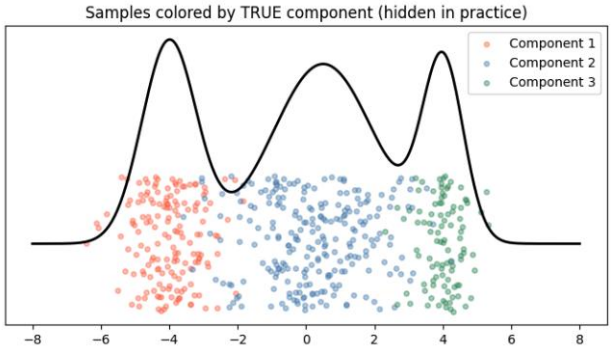
Gaussian Mixture Models – Sampling/Generation

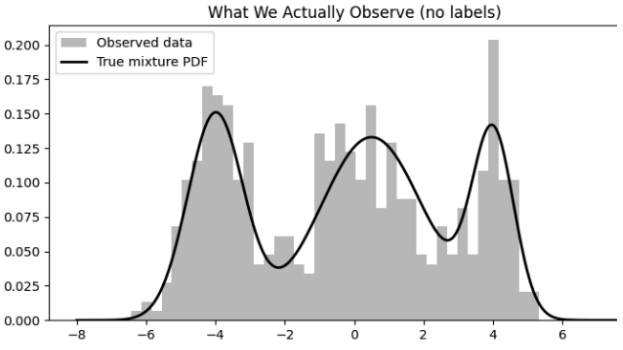
$$p(x) = \sum_{k=1}^{K} \pi_k \, \mathcal{N}(x \mid \mu_k, \sigma_k)$$

Because the GMM has a clean generative story, sampling is easy:

1. Draw a component index $z \in \{1, \ldots, K\}$ with probabilities $[\pi_1, \ldots, \pi_K]$.
2. Draw $x \sim \mathcal{N}(\mu_z, \sigma_z)$.

```python
np.random.seed(0)

# True GMM parameters
pis_true    = np.array([0.3, 0.5, 0.2])
mus_true    = np.array([-4.0, 0.5, 4.0])
sigmas_true = np.array([0.8, 1.5, 0.6])
K = len(pis_true)
N = 500

# Step 1: draw component labels
z = np.random.choice(K, size=N, p=pis_true)      # shape (N,)

# Step 2: draw samples from assigned components
x_samples = np.array([np.random.normal(mus_true[zi], sigmas_true[zi]) for zi in z])

x_plot = np.linspace(-8, 8, 400)
mixture_pdf = sum(pi * norm.pdf(x_plot, mu, sigma)
                  for pi, mu, sigma in zip(pis_true, mus_true, sigmas_true))

fig, axes = plt.subplots(1, 2, figsize=(14, 4))

# Left: colored by true component (normally hidden!)
for k in range(K):
    axes[0].scatter(x_samples[z == k], np.zeros(np.sum(z == k)) + np.random.uniform(-0.05, 0.05, np.sum(z == k)),
                    alpha=0.4, s=15, color=colors[k], label=f'Component {k+1}')
axes[0].plot(x_plot, mixture_pdf, 'k-', lw=2)
axes[0].set_yticks([])
axes[0].set_title('Samples colored by TRUE component (hidden in practice)')
axes[0].legend()

# Right: what we actually observe — unlabeled
axes[1].hist(x_samples, bins=40, density=True, alpha=0.5, color='gray', label='Observed data')
axes[1].plot(x_plot, mixture_pdf, 'k-', lw=2, label='True mixture PDF')
axes[1].set_title('What We Actually Observe (no labels)')
axes[1].legend()

plt.tight_layout()
plt.show()

print(f'Samples per component (true): {[(z==k).sum() for k in range(K)]}')
```



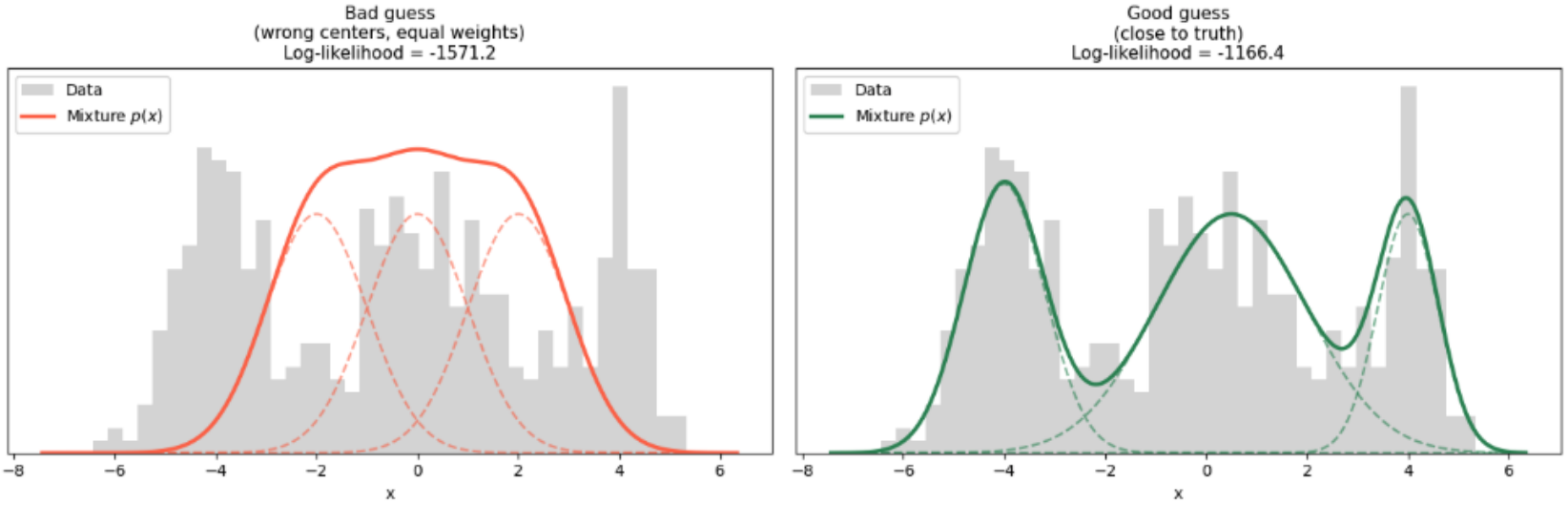


Gaussian Mixture Models – Estimation

$$\log \mathcal{L}(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\sigma}) = \sum_{i=1}^{N} \log \left(\sum_{k=1}^{K} \pi_k \, \mathcal{N}(x_i \mid \mu_k, \sigma_k) \right)$$

```
guesses = {
    'Bad guess\n(wrong centers, equal weights)': {
        'pis':    [1/3, 1/3, 1/3],
        'mus':    [-2.0, 0.0, 2.0],
        'sigmas': [1.0,  1.0, 1.0],
        'color':  'tomato',
    },
    'Good guess\n(close to truth)': {
        'pis':    pis_true,
        'mus':    mus_true,
        'sigmas': sigmas_true,
        'color':  'seagreen',
    },
}
```

Same data, different parameter guesses → very different log-likelihoods




```
Bad guess  LL: -1571.23
Good guess LL: -1166.43
Higher log-likelihood = better fit.
```

Estimating GMM using Maximum Likelihood with Gradient descent

$$\mathcal{L}(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\sigma}) = -\sum_{i=1}^{N} \log \underbrace{\left(\sum_{k=1}^{K} \pi_k \, \mathcal{N}(x_i \mid \mu_k, \sigma_k) \right)}_{p(x_i)}$$

$$\pi_k = \frac{e^{\alpha_k}}{\sum_j e^{\alpha_j}} \quad \text{(softmax of free logits } \boldsymbol{\alpha}\text{)},$$

Symbol	Name	Meaning
π_k	**mixing weight**	How often this component is chosen. Must satisfy $\pi_k \geq 0$ and $\sum_k \pi_k = 1$.
μ_k	**mean**	Where the k-th Gaussian is centered.
σ_k	**standard deviation**	How wide the k-th Gaussian is.

We can use gradient descent

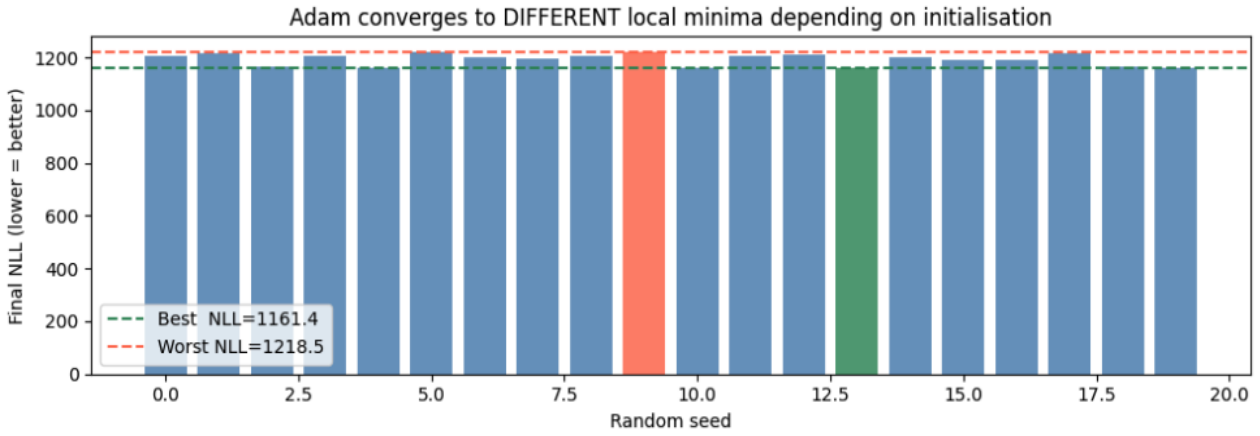
Estimating GMM using Maximum Likelihood with Gradient descent

$$\mathcal{L}(\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\sigma}) = -\sum_{i=1}^{N} \log \underbrace{\left(\sum_{k=1}^{K} \pi_k \, \mathcal{N}(x_i \mid \mu_k, \sigma_k) \right)}_{p(x_i)}$$

$$\pi_k = \frac{e^{\alpha_k}}{\sum_j e^{\alpha_j}} \quad \text{(softmax of free logits } \boldsymbol{\alpha}\text{)},$$

Symbol	Name	Meaning
π_k	**mixing weight**	How often this component is chosen. Must satisfy $\pi_k \geq 0$ and $\sum_k \pi_k = 1$.
μ_k	**mean**	Where the k-th Gaussian is centered.
σ_k	**standard deviation**	How wide the k-th Gaussian is.

```python
x_torch = torch.tensor(x_samples, dtype=torch.float32)
K = 3

def gmm_nll_torch(x, pi_logits, mus, log_sigmas):
    """Negative log-likelihood of a GMM (PyTorch, differentiable)."""
    pis    = F.softmax(pi_logits, dim=0)
    sigmas = torch.exp(log_sigmas)
    log_components = (
        torch.log(pis)[None, :]
        - log_sigmas[None, :]
        - 0.5 * np.log(2 * np.pi)
        - 0.5 * ((x[:, None] - mus[None, :]) / sigmas[None, :]) ** 2
    )
    return -torch.logsumexp(log_components, dim=1).sum()

def run_adam(seed, n_iters=500, lr=0.05):
    torch.manual_seed(seed)
    pi_logits  = torch.randn(K, requires_grad=True)
    mus_t      = torch.randn(K, requires_grad=True)
    log_sigmas = torch.zeros(K, requires_grad=True)
    opt = torch.optim.Adam([pi_logits, mus_t, log_sigmas], lr=lr)
    for _ in range(n_iters):
        opt.zero_grad()
        gmm_nll_torch(x_torch, pi_logits, mus_t, log_sigmas).backward()
        opt.step()
    with torch.no_grad():
        return (F.softmax(pi_logits, dim=0).numpy().copy(),
                mus_t.numpy().copy(),
                torch.exp(log_sigmas).numpy().copy(),
                gmm_nll_torch(x_torch, pi_logits, mus_t, log_sigmas).item())
```



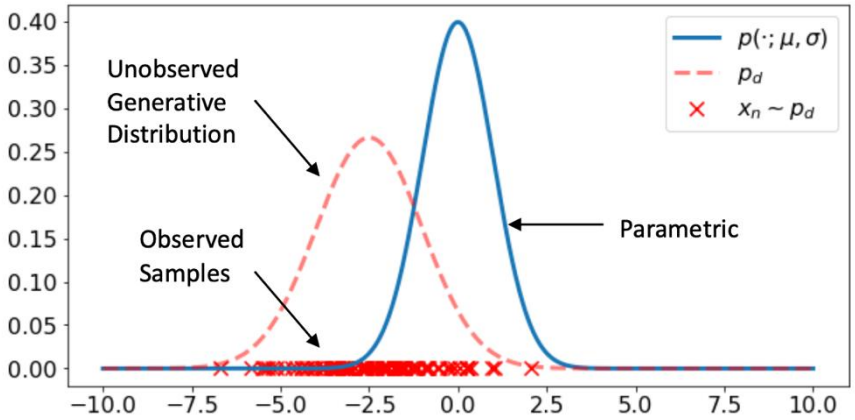
Estimating GMM using Maximum Likelihood – Theoretical Sol?

Gaussian Kernel Density Estimation

$$\{x_i | x_i \sim P_d\}_{i=1}^{N}$$

$$p(x;\mu,\sigma) = N(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

What is μ, σ ?



3- Minimizing Negative Log-Likelihood:

$$\underset{\mu,\sigma}{\operatorname{argmin}} \underbrace{\sum_{n=1}^{N} \frac{\log(2\pi\sigma^2)}{2} + \frac{(x_n-\mu)^2}{2\sigma^2}}_{L} \longrightarrow \begin{cases} \frac{\partial L}{\partial \mu} = 0 \Rightarrow \mu_* = \frac{1}{N}\sum_{n=1}^{N} x_n \\ \frac{\partial L}{\partial \sigma} = 0 \Rightarrow \sigma_*^2 = \frac{1}{N}\sum_{n=1}^{N} (x_n - \mu^*)^2 \end{cases}$$

Estimating GMM using Maximum Likelihood – Theoretical Sol?

1D Gaussian

$$\ell(\mu, \sigma) = \log \prod_{i=1}^{N} \mathcal{N}(x_i \mid \mu, \sigma) = \sum_{i=1}^{N} \log \mathcal{N}(x_i \mid \mu, \sigma)$$

$$= \sum_{i=1}^{N} \left[-\log \sigma - \tfrac{1}{2} \log 2\pi - \frac{(x_i - \mu)^2}{2\sigma^2} \right]$$

1D GMM

$$p(x_i \mid \theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x_i \mid \mu_k, \sigma_k)$$

$$\ell(\theta) = \sum_{i=1}^{N} \log \underbrace{\left[\sum_{k=1}^{K} \pi_k \mathcal{N}(x_i \mid \mu_k, \sigma_k) \right]}_{\text{sum inside the log}}$$

$$\frac{\partial \ell}{\partial \mu_k} = \sum_{i=1}^{N} \frac{\pi_k \mathcal{N}(x_i \mid \mu_k, \sigma_k)}{\sum_j \pi_j \mathcal{N}(x_i \mid \mu_j, \sigma_j)} \cdot \frac{x_i - \mu_k}{\sigma_k^2} = 0$$

Estimating GMM using Maximum Likelihood – Theoretical Sol?

1D Gaussian

$$\ell(\mu, \sigma) = \log \prod_{i=1}^{N} \mathcal{N}(x_i \mid \mu, \sigma) = \sum_{i=1}^{N} \log \mathcal{N}(x_i \mid \mu, \sigma)$$

$$= \sum_{i=1}^{N} \left[-\log \sigma - \tfrac{1}{2} \log 2\pi - \frac{(x_i - \mu)^2}{2\sigma^2} \right]$$

1D GMM

$$p(x_i \mid \theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x_i \mid \mu_k, \sigma_k)$$

$$\ell(\theta) = \sum_{i=1}^{N} \log \underbrace{\left[\sum_{k=1}^{K} \pi_k \mathcal{N}(x_i \mid \mu_k, \sigma_k) \right]}_{\text{sum inside the log}}$$

Different parameters they are coupled together

$$\frac{\partial \ell}{\partial \mu_k} = \sum_{i=1}^{N} \frac{\pi_k \mathcal{N}(x_i \mid \mu_k, \sigma_k)}{\sum_j \pi_j \mathcal{N}(x_i \mid \mu_j, \sigma_j)} \cdot \frac{x_i - \mu_k}{\sigma_k^2} = 0$$

Estimating GMM using Maximum Likelihood – Theoretical Sol?

$$p(x_i \mid \theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x_i \mid \mu_k, \sigma_k)$$

$$z_{ik} = \begin{cases} 1 & \text{if point } i \text{ was generated by component } k \\ 0 & \text{otherwise} \end{cases}$$

$$\ell(\theta) = \sum_{i=1}^{N} \log \underbrace{\left[\sum_{k=1}^{K} \pi_k \mathcal{N}(x_i \mid \mu_k, \sigma_k) \right]}_{\text{sum inside the log}}$$

$$\ell_{\text{complete}}(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} z_{ik} \underbrace{[\log \pi_k + \log \mathcal{N}(x_i \mid \mu_k, \sigma_k)]}_{\text{log is now INSIDE — easy!}}$$

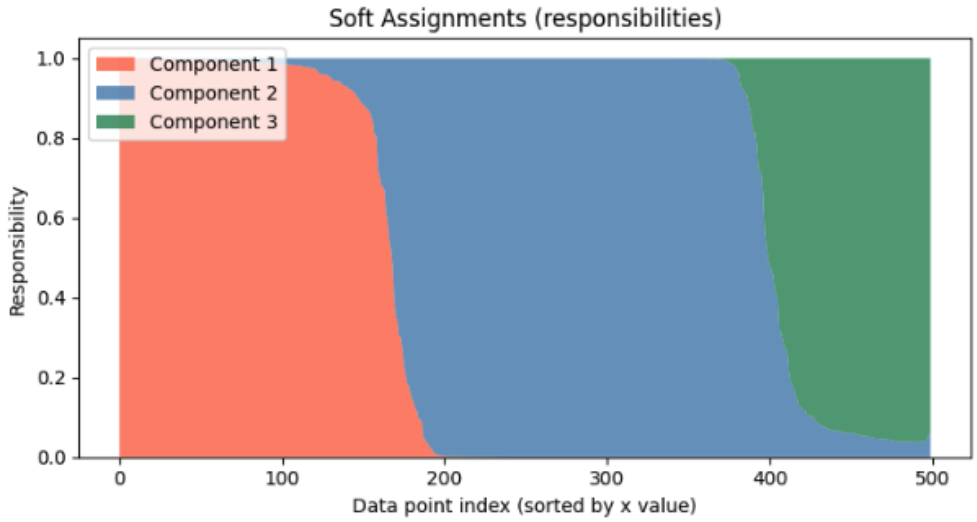
$$\ell_{\text{complete}}(\theta) = \underbrace{\sum_{k=1}^{K} \left(\sum_{i=1}^{N} z_{ik} \right) \log \pi_k}_{\text{depends only on } \pi} + \underbrace{\sum_{k=1}^{K} \sum_{i=1}^{N} z_{ik} \left[-\log \sigma_k - \tfrac{1}{2} \log 2\pi - \frac{(x_i - \mu_k)^2}{2\sigma_k^2} \right]}_{\text{depends only on } \mu_k, \sigma_k \text{ — independent across } k}$$

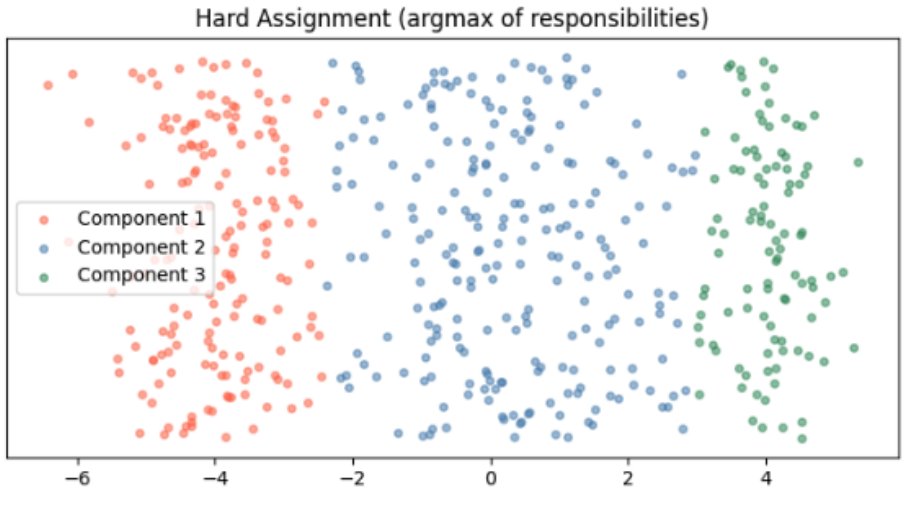
Once we introduce z, different parameters are decoupled

Estimating GMM using Maximum Likelihood – Theoretical Sol?

$$\ell_{\text{complete}}(\theta) = \underbrace{\sum_{k=1}^{K}\left(\sum_{i=1}^{N} z_{ik}\right)\log \pi_k}_{\text{depends only on } \pi} + \underbrace{\sum_{k=1}^{K}\sum_{i=1}^{N} z_{ik}\left[-\log \sigma_k - \tfrac{1}{2}\log 2\pi - \frac{(x_i-\mu_k)^2}{2\sigma_k^2}\right]}_{\text{depends only on } \mu_k, \sigma_k \text{ — independent across } k}$$

Once we introduce z, different parameters are decoupled





Estimating GMM using Maximum Likelihood – EM Algorithm

E - Step

$$p(x_i \mid \theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x_i \mid \mu_k, \sigma_k)$$

$$z_{ik} = \begin{cases} 1 & \text{if point } i \text{ was generated by component } k \\ 0 & \text{otherwise} \end{cases}$$

$$\ell(\theta) = \sum_{i=1}^{N} \log \underbrace{\left[\sum_{k=1}^{K} \pi_k \mathcal{N}(x_i \mid \mu_k, \sigma_k) \right]}_{\text{sum inside the log}}$$

$$\ell_{\text{complete}}(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} z_{ik} \underbrace{[\log \pi_k + \log \mathcal{N}(x_i \mid \mu_k, \sigma_k)]}_{\text{log is now INSIDE — easy!}}$$

z_{ik}: how much contribution does k^{th} gaussian represent to i^{th} data?

$$r_{ik} = \mathbb{E}[z_{ik} \mid x_i, \theta^{\text{old}}] = P(z_{ik} = 1 \mid x_i, \theta^{\text{old}})$$

$$r_{ik} = \frac{\pi_k^{\text{old}} \mathcal{N}(x_i \mid \mu_k^{\text{old}}, \sigma_k^{\text{old}})}{\sum_{j=1}^{K} \pi_j^{\text{old}} \mathcal{N}(x_i \mid \mu_j^{\text{old}}, \sigma_j^{\text{old}})}$$

Estimating GMM using Maximum Likelihood – EM Algorithm

M - Step

$$\ell_{\text{complete}}(\theta) = \sum_{i=1}^{N}\sum_{k=1}^{K} \underbrace{z_{ik}}_{\text{unknown}} \left[\log \pi_k + \log \mathcal{N}(x_i \mid \mu_k, \sigma_k)\right] \longrightarrow \sum_{i=1}^{N}\sum_{k=1}^{K} \underbrace{r_{ik}}_{\text{known}} \left[\log \pi_k + \log \mathcal{N}(x_i \mid \mu_k, \sigma_k)\right]$$

z_{ik}: how much contribution does k^{th} gaussian represent to i^{th} data?

5a — Update π_k $\quad \sum_k \pi_k = 1,\ \pi_k \geq 0$ $\quad \frac{\partial}{\partial \pi_k}\left[\ell_{\text{complete}} - \lambda\left(\sum_k \pi_k - 1\right)\right] = \frac{N_k}{\pi_k} - \lambda = 0 \implies \pi_k = \frac{N_k}{\lambda}$

$\sum_k \pi_k = 1 \Rightarrow \lambda = \sum_k N_k = N.$

$$\pi_k^{\text{new}} = \frac{N_k}{N}, \qquad N_k = \sum_{i=1}^{N} r_{ik}$$

M - Step

$$\ell_{\text{complete}}(\theta) = \sum_{i=1}^{N}\sum_{k=1}^{K} \underbrace{z_{ik}}_{\text{unknown}} [\log \pi_k + \log \mathcal{N}(x_i \mid \mu_k, \sigma_k)] \longrightarrow \sum_{i=1}^{N}\sum_{k=1}^{K} \underbrace{r_{ik}}_{\text{known}} [\log \pi_k + \log \mathcal{N}(x_i \mid \mu_k, \sigma_k)]$$

z_{ik}: how much contribution does k^{th} gaussian represent to i^{th} data?

5b — Update μ_k

5c — Update σ_k

$$\sum_{i=1}^{N} r_{ik} \log \mathcal{N}(x_i \mid \mu_k, \sigma_k) = -\frac{1}{2\sigma_k^2}\sum_{i=1}^{N} r_{ik}(x_i - \mu_k)^2 + \text{const}$$

$$\frac{1}{\sigma_k^2}\sum_{i=1}^{N} r_{ik}(x_i - \mu_k) = 0 \implies \sum_i r_{ik}x_i = \mu_k N_k$$

$$-\frac{N_k}{\sigma_k} + \frac{1}{\sigma_k^3}\sum_{i=1}^{N} r_{ik}(x_i - \mu_k^{\text{new}})^2 = 0$$

$$\boxed{\mu_k^{\text{new}} = \frac{\sum_{i=1}^{N} r_{ik}\, x_i}{N_k}}$$

$$\boxed{\sigma_k^{\text{new}} = \sqrt{\frac{\sum_{i=1}^{N} r_{ik}(x_i - \mu_k^{\text{new}})^2}{N_k}}}$$

Estimating GMM using Maximum Likelihood – EM Algorithm

```
initialise  π, μ, σ  randomly

repeat:
    E-step:  r_ik  ←  Bayes' theorem  (Step 4)
    M-step:  N_k   ←  Σ_i r_ik
             π_k   ←  N_k / N
             μ_k   ←  Σ_i(r_ik · x_i) / N_k
             σ_k   ←  sqrt(Σ_i(r_ik · (x_i − μ_k)²) / N_k)
until  |ΔLL| < ε
```

```python
# ── Gaussian PDF (manual, no scipy wrapper) ──────────────────
def gaussian_pdf(x, mu, sigma):
    """N(x | mu, sigma) computed directly from the formula."""
    coeff  = 1.0 / (sigma * np.sqrt(2 * np.pi))
    kernel = np.exp(-0.5 * ((x - mu) / sigma) ** 2)
    return coeff * kernel
```

```python
# ── E-step: Bayes' theorem → responsibilities ────────────────
def e_step(x, pis, mus, sigmas):
    """
    Compute r[i

    From Bayes:

    Returns r: array of shape (N, K).
    """
    N, K = len(x), len(pis)

    # Numerator: pi_k * N(x_i | mu_k, sigma_k)  shape (N, K)
    numer = np.column_stack([
        pis[k] * gaussian_pdf(x, mus[k], sigmas[k])
        for k in range(K)
    ])

    # Denominator: p(x_i) = sum over k of numerator row   shape (N, 1)
    denom = numer.sum(axis=1, keepdims=True)

    r = numer / denom          # soft assignments, each row sums to 1
    return r
```

$$r_{ik} = \frac{\pi_k^{\text{old}}\,\mathcal{N}(x_i \mid \mu_k^{\text{old}}, \sigma_k^{\text{old}})}{\sum_{j=1}^{K} \pi_j^{\text{old}}\,\mathcal{N}(x_i \mid \mu_j^{\text{old}}, \sigma_j^{\text{old}})}$$

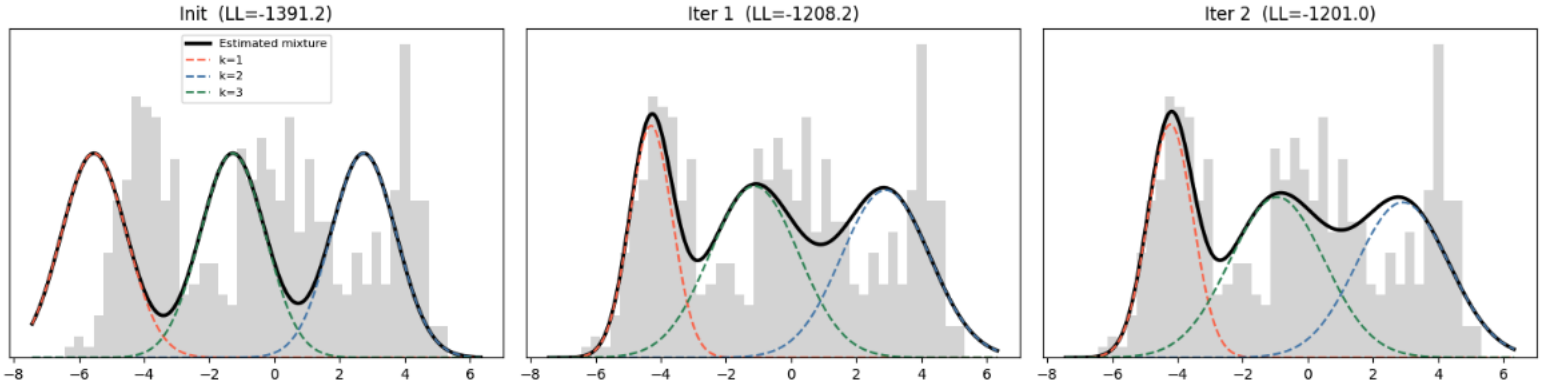
```python
# ── M-step: closed-form updates from the Q-function derivatives ──
def m_step(x, r):
    """
    Update parameters by maximising the expected complete-data log-likelihood.

    Derived formulas:
        N_k     = sum_i r_ik                          (effective count)
        pi_k    = N_k / N                             (from Lagrange multiplier)
        mu_k    = sum_i(r_ik * x_i) / N_k             (weighted mean)
        sigma_k = sqrt(sum_i(r_ik*(x_i-mu_k)^2) / N_k)   (weighted std)

    Returns new_pis, new_mus, new_sigmas.
    """
    N, K = r.shape

    # Effective count per component
    Nk = r.sum(axis=0)

    # pi update — from d/d(pi_k)[sum N_k log pi_k − lambda(sum pi_k
    new_pis = Nk / N

    # mu update — from d/d(mu_k)[sum r_ik * log N(x_i|mu_k,sigma_k)]
    new_mus = (r * x[:, None]).sum(axis=0) / Nk   # shape (K,)

    # sigma update — from d/d(sigma_k)[...] = 0
    residuals_sq = (x[:, None] - new_mus[None, :]) ** 2   # (N, K
    new_sigmas   = np.sqrt((r * residuals_sq).sum(axis=0) / Nk)

    return new_pis, new_mus, new_sigmas
```

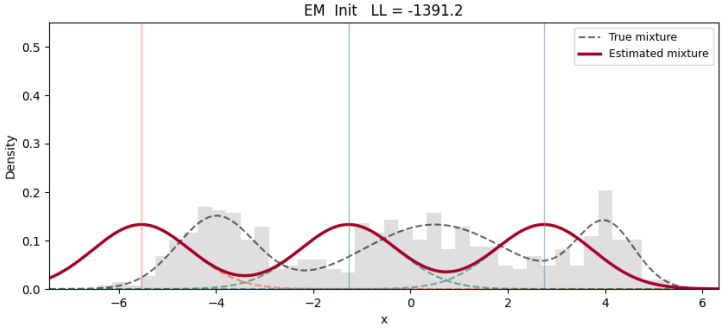
$$\pi_k^{\text{new}} = \frac{N_k}{N}, \qquad N_k = \sum_{i=1}^{N} r_{ik}$$

$$\mu_k^{\text{new}} = \frac{\sum_{i=1}^{N} r_{ik}\, x_i}{N_k}$$

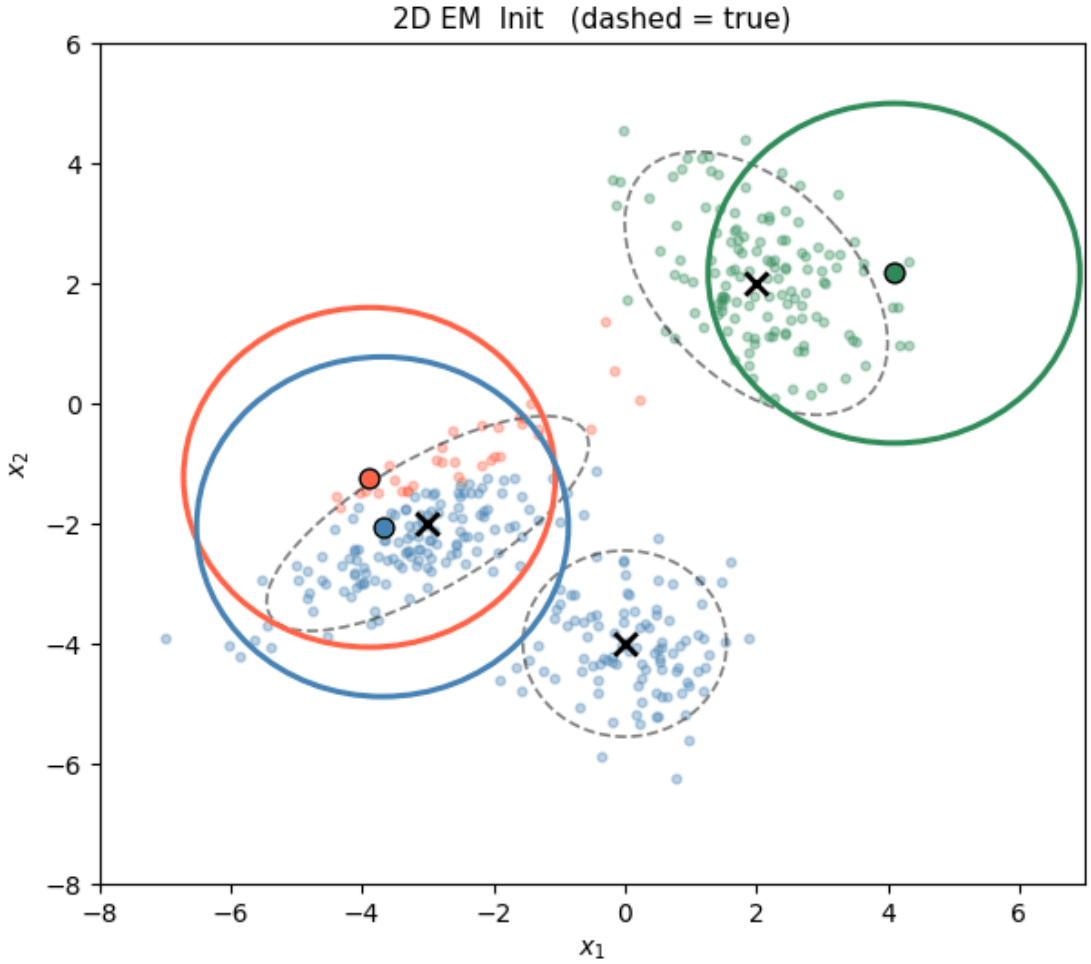
$$\sigma_k^{\text{new}} = \sqrt{\frac{\sum_{i=1}^{N} r_{ik}(x_i - \mu_k^{\text{new}})^2}{N_k}}$$

Estimating GMM using Maximum Likelihood – EM Algorithm





Estimating GMM using Maximum Likelihood – EM Algorithm



QA