Adv ML for Gen-AI

Gaussian KDE

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/

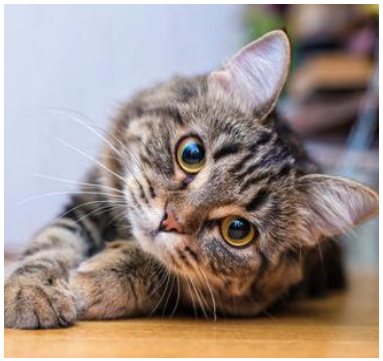
Data Distribution

Dog





Cat


Data Distribution

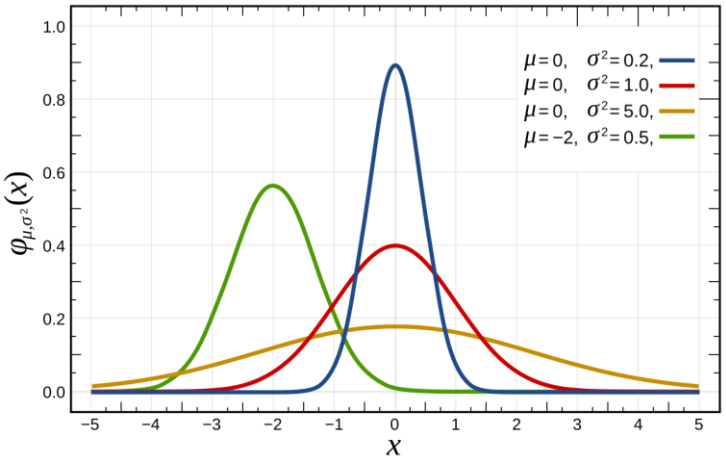
Dog – P(Dog)


Cat – P(Cat)


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

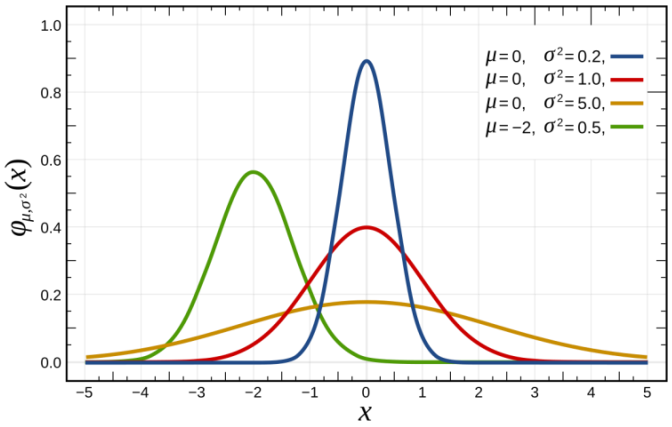
2. But somehow dog and cat image distributions are different

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



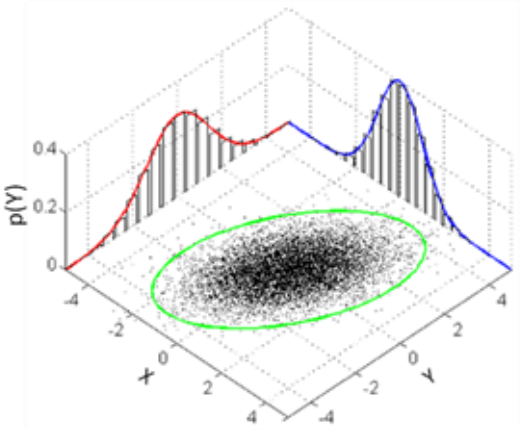
Data Distribution

1D Gaussian Distribution



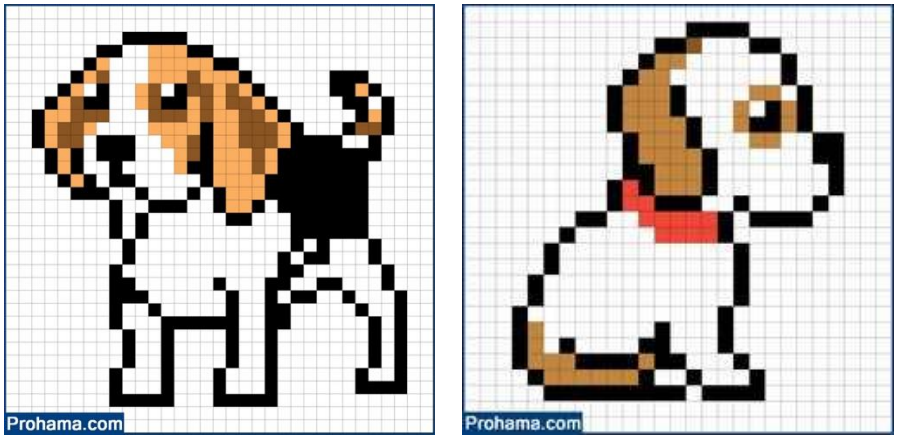
$\mathbb{R}$

2D Gaussian Distribution



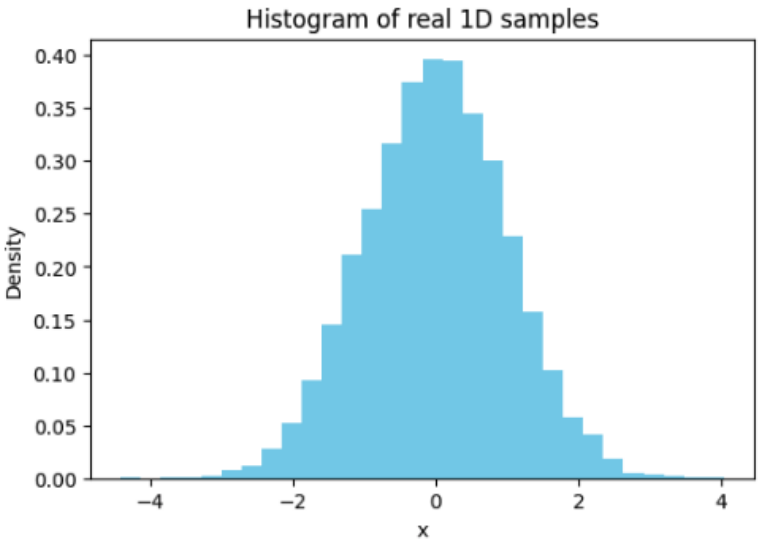
$\mathbb{R}^2$

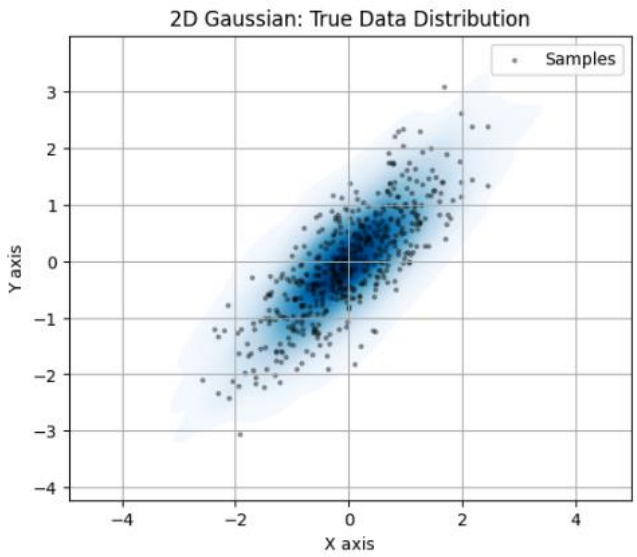
$\mathbb{R}^{256\times256}$



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

Gaussian Distribution





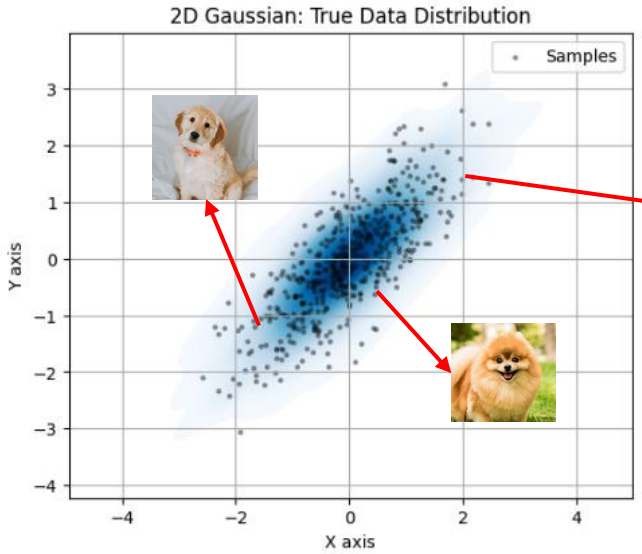

```python
import numpy as np
import matplotlib.pyplot as plt

# Generate real 1D data samples from a Gaussian distribution N(mean, std^2)
mean1d, std1d = 0.0, 1.0
real_samples_1d = np.random.normal(mean1d, std1d, size=10000)

# Plot histogram of real samples
plt.figure(figsize=(6,4))
plt.hist(real_samples_1d, bins=30, density=True, color='skyblue')
plt.title("Histogram of real 1D samples")
plt.xlabel("x"); plt.ylabel("Density")
plt.show()
```

```python
# Generate real 2D data (Gaussian with correlation)
mean2d = [0.0, 0.0]
cov2d = [[1.0, 0.8],
         [0.8, 1.0]]  # covariance matrix with correlation 0.8
real_samples_2d = np.random.multivariate_normal(mean2d, cov2d, size=5000)
```

```python
import seaborn as sns

# Unpack real samples into x and y
x_real = real_samples_2d[:, 0]
y_real = real_samples_2d[:, 1]

# Plot: scatter with density contour
plt.figure(figsize=(6, 5))
sns.kdeplot(x=x_real, y=y_real, fill=True, cmap="Blues", thresh=0.01, levels=100)
plt.scatter(x_real[:500], y_real[:500], s=5, color="black", alpha=0.3, label="Samples")
plt.title("2D Gaussian: True Data Distribution")
plt.xlabel("X axis"); plt.ylabel("Y axis")
plt.legend()
plt.grid(True)
plt.axis("equal")
plt.show()
```

Estimating Data Distribution and Sampling from it

Estimate Data Distribution



Sampling from it

Goal of Generative Learning

Probability distribution of the **objective** based on the **observed data**

$$\{x_i\}_{i=1}^{N} \xrightarrow{\text{Good Model}} P(x) \xrightarrow{\text{Good Data}} x$$

- **Machine Learning Methods**
  - Gaussian Kernel Density Estimation
  - Gaussian Mixture Models

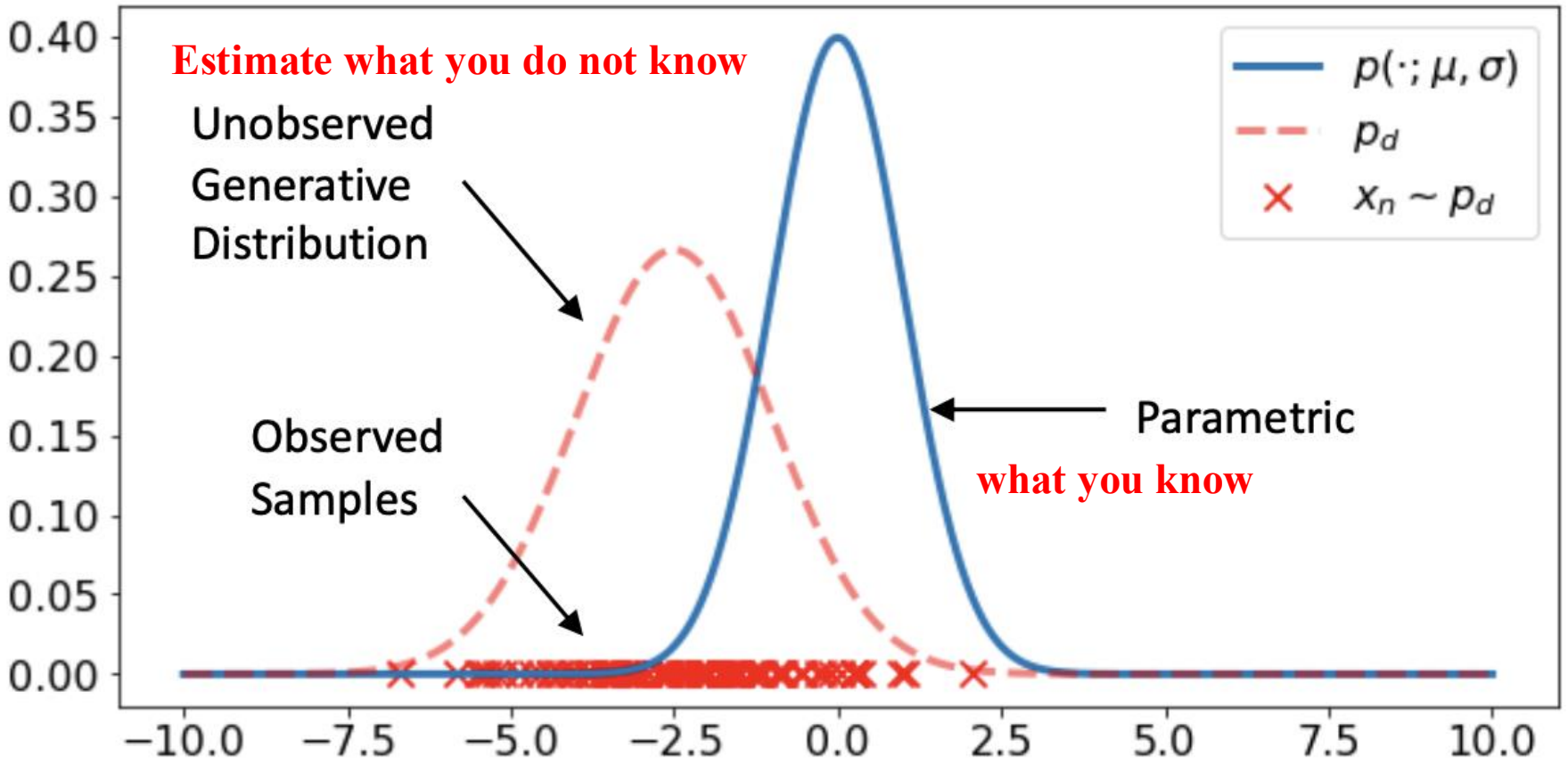
Using existing function to estimate what you do not know that can best fit your observation

- **Deep Learning Methods**
  - Auto-Encoder (AE)
  - Variational AE (VAE)
  - Generative Adversarial Network
  - Diffusion Model

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

Gaussian Kernel Density Estimation

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

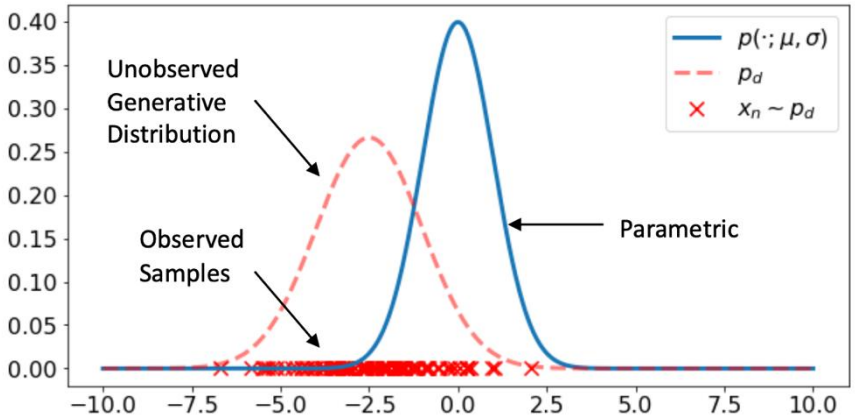


Gaussian Kernel Density Estimation

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

What you know is Gaussian

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



What you observe is a set of i.i.d samples from a Gaussian

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

How can we estimate the gaussian distribution?

What is μ, σ ?

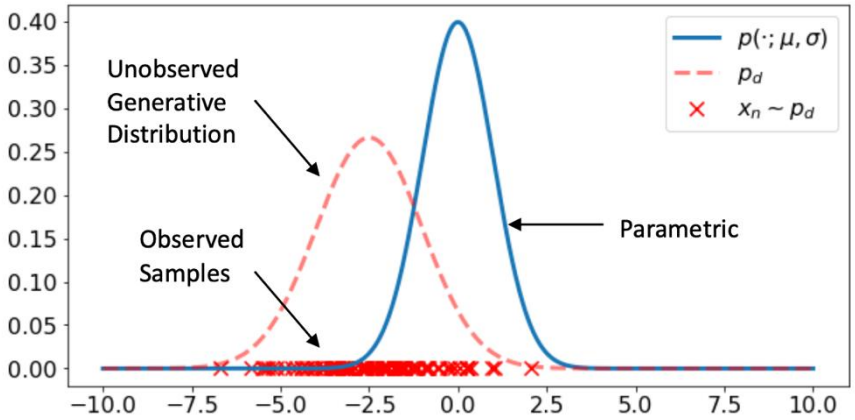
Gaussian Kernel Density Estimation

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:

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

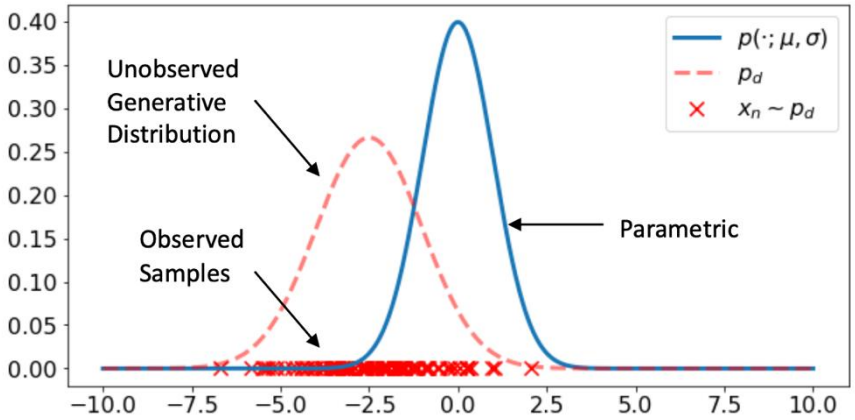
Gaussian Kernel Density Estimation

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 ?



2 - Maximum Log-Likelihood:

$$\underset{\mu,\sigma}{\operatorname{argmax}} \log\left(\prod_{n=1}^{N} p(x_n;\mu,\sigma)\right) = \sum_{n=1}^{N} \log(p(x_n;\mu,\sigma))$$

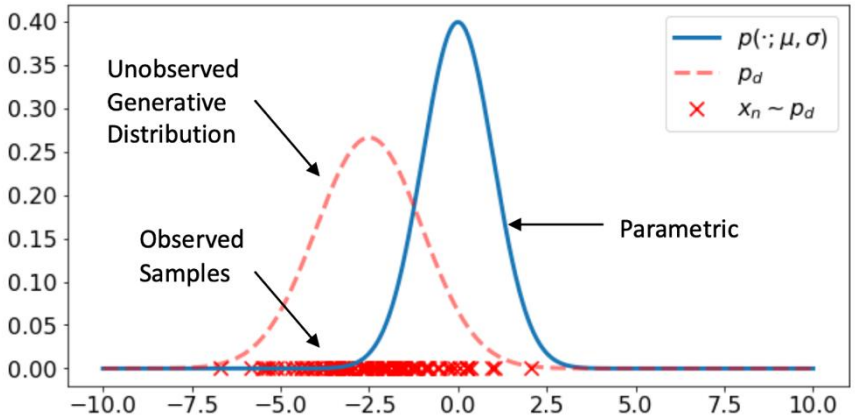
Gaussian Kernel Density Estimation

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:

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

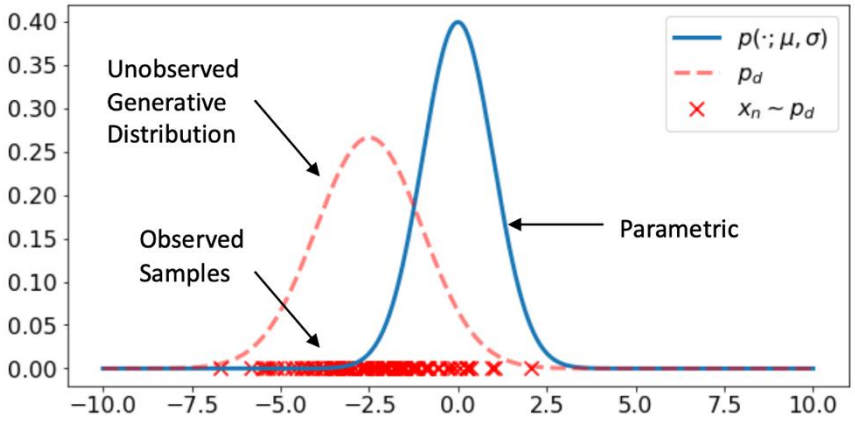
Gaussian Kernel Density Estimation – Why it works?

2 - Maximum Log-Likelihood:

$$\underset{\mu,\sigma}{\operatorname{argmax}} \log\left(\prod_{n=1}^{N} p(x_n;\mu,\sigma)\right) = \sum_{n=1}^{N} \log(p(x_n;\mu,\sigma))$$

Monte-Carlo Approximation

$$\int_{\mathbb{R}} p_d(x)\log(p(x;\mu,\sigma))\,dx \approx \frac{1}{N}\sum_{n=1}^{N} \log(p(x_n;\mu,\sigma))$$



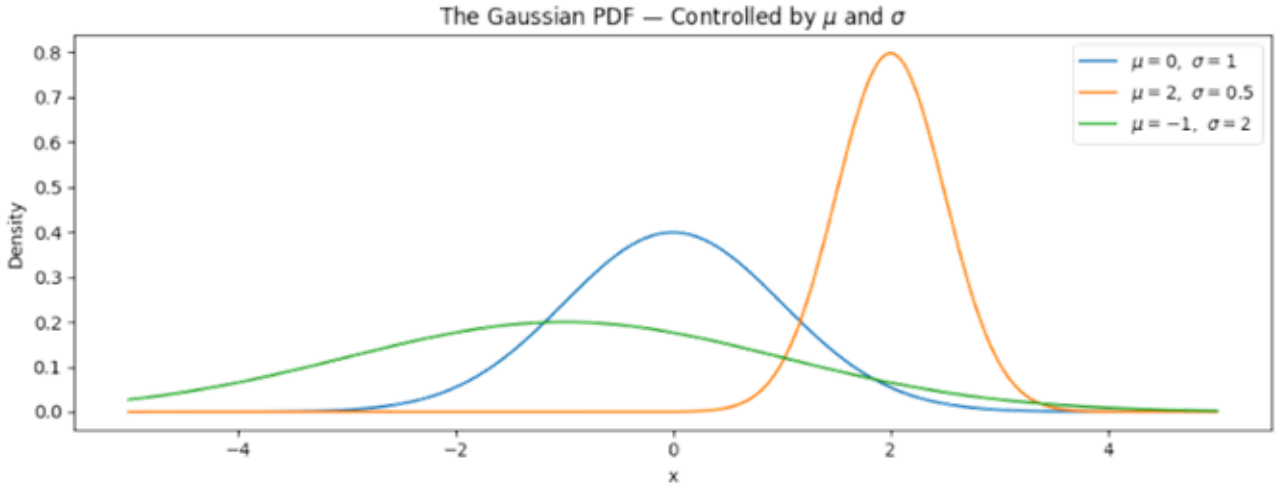
$$\underset{\mu,\sigma}{\operatorname{argmax}} \int_{\mathbb{R}} p_d(x)\log(p(x;\mu,\sigma))\,dx = \underset{\mu,\sigma}{\operatorname{argmax}} \int_{\mathbb{R}} p_d(x)\log(p(x;\mu,\sigma))\,dx - \int_{\mathbb{R}} p_d(x)\log(p_d(x))\,dx$$

$$= \underset{\mu,\sigma}{\operatorname{argmax}} \int_{\mathbb{R}} p_d(x)\log\left(\frac{p(x;\mu,\sigma)}{p_d(x)}\right)dx = \underset{\mu,\sigma}{\operatorname{argmin}} \int_{\mathbb{R}} p_d(x)\log\left(\frac{p_d(x)}{p(x;\mu,\sigma)}\right)dx$$

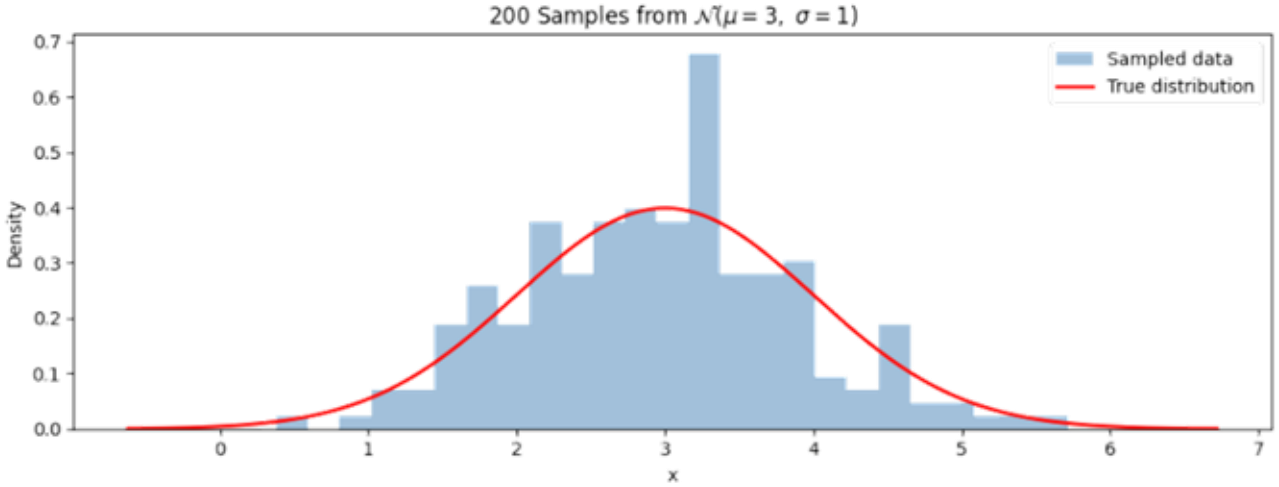
$$= \underset{\mu,\sigma}{\operatorname{argmin}} D_{KL}(p_d \,||\, p(\cdot;\mu,\sigma))$$

Maximize Log-Likelihood of parametric distribution = Minimize KL Divergence between the data distribution and the parametric data distribution

1D Gaussian Distribution



Gaussian PDF



Sample from one Gaussian

1D Gaussian Distribution

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

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

```
Candidate                    NLL
------------------------------
(0.0, 1.0)               1145.73
(3.0, 1.0)                270.20  <- best
(3.0, 2.5)                380.87
(2.0, 0.5)                758.18
(4.5, 1.0)                507.43
(3.2, 0.9)                276.35

Best candidate by NLL: μ=3.0, σ=1.0
```

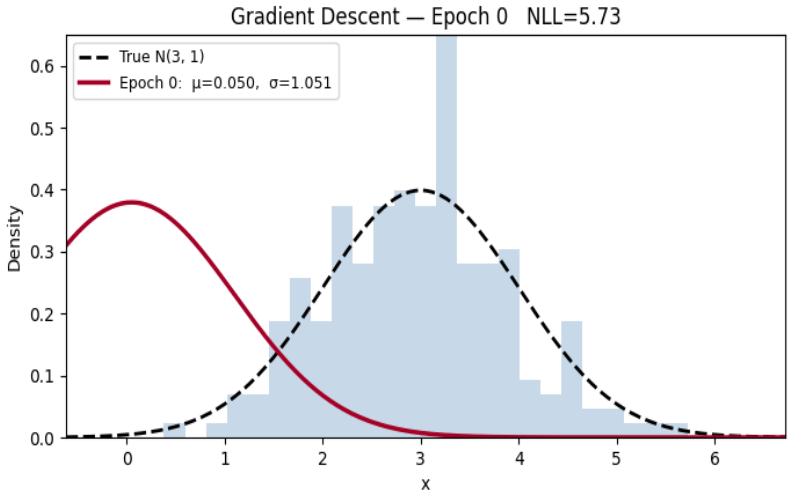
```python
def neg_log_likelihood(mu, sigma, data):
    """Negative log-likelihood of data under N(mu, sigma)."""
    N = len(data)
    nll = (
        0.5 * N * np.log(2 * np.pi)
        + N * np.log(sigma)
        + 0.5 * np.sum((data - mu) ** 2) / (sigma ** 2)
    )
    return nll

candidates = [(0.0, 1.0), (3.0, 1.0), (3.0, 2.5), (2.0, 0.5), (4.5, 1.0), (3.2, 0.9)]

nll_values = [(mu, sigma, neg_log_likelihood(mu, sigma, data)) for mu, sigma in candidates]
best = min(nll_values, key=lambda t: t[2])
```

1D Gaussian Distribution Estimation

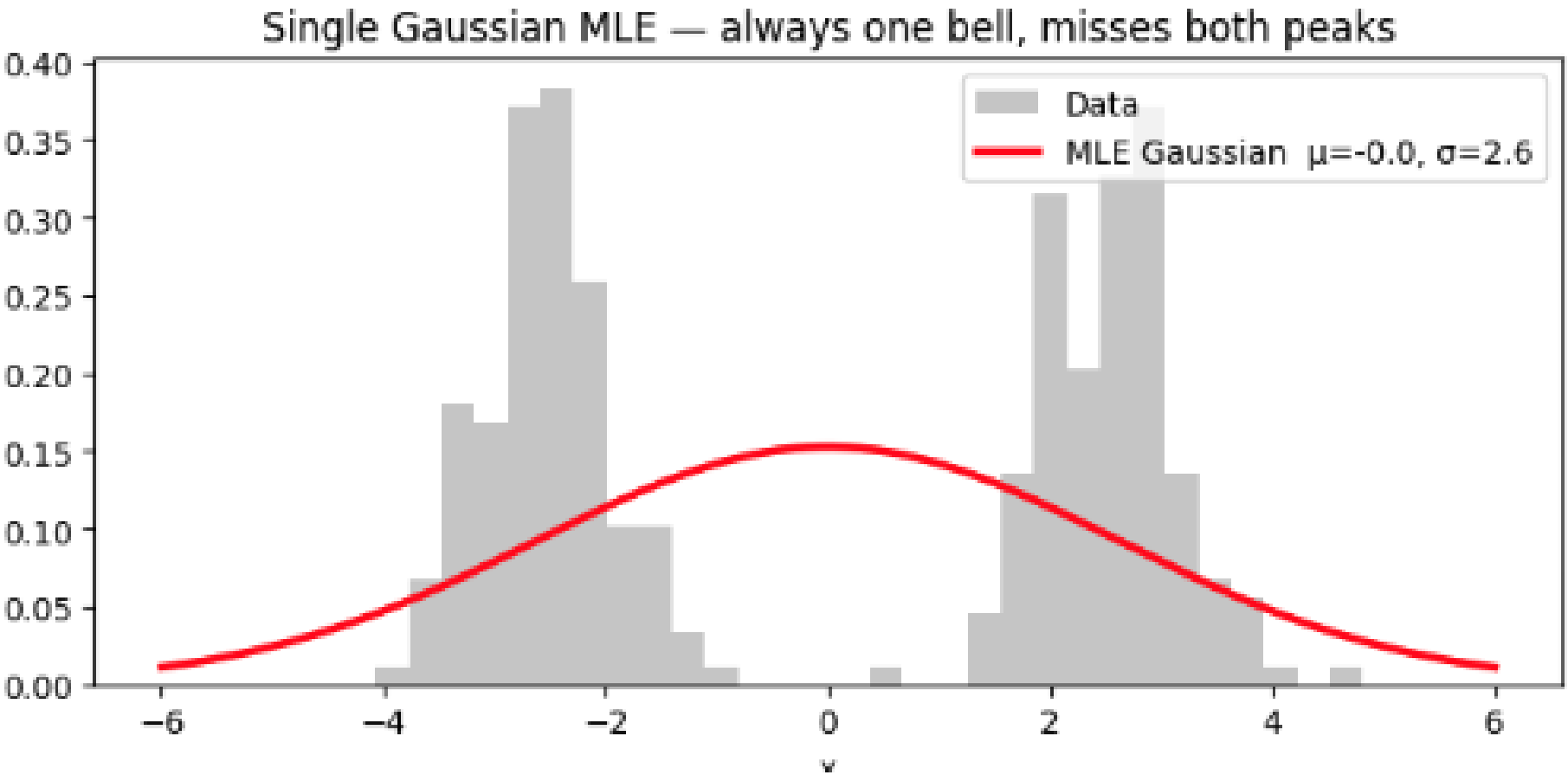
```python
import torch
import numpy as np

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



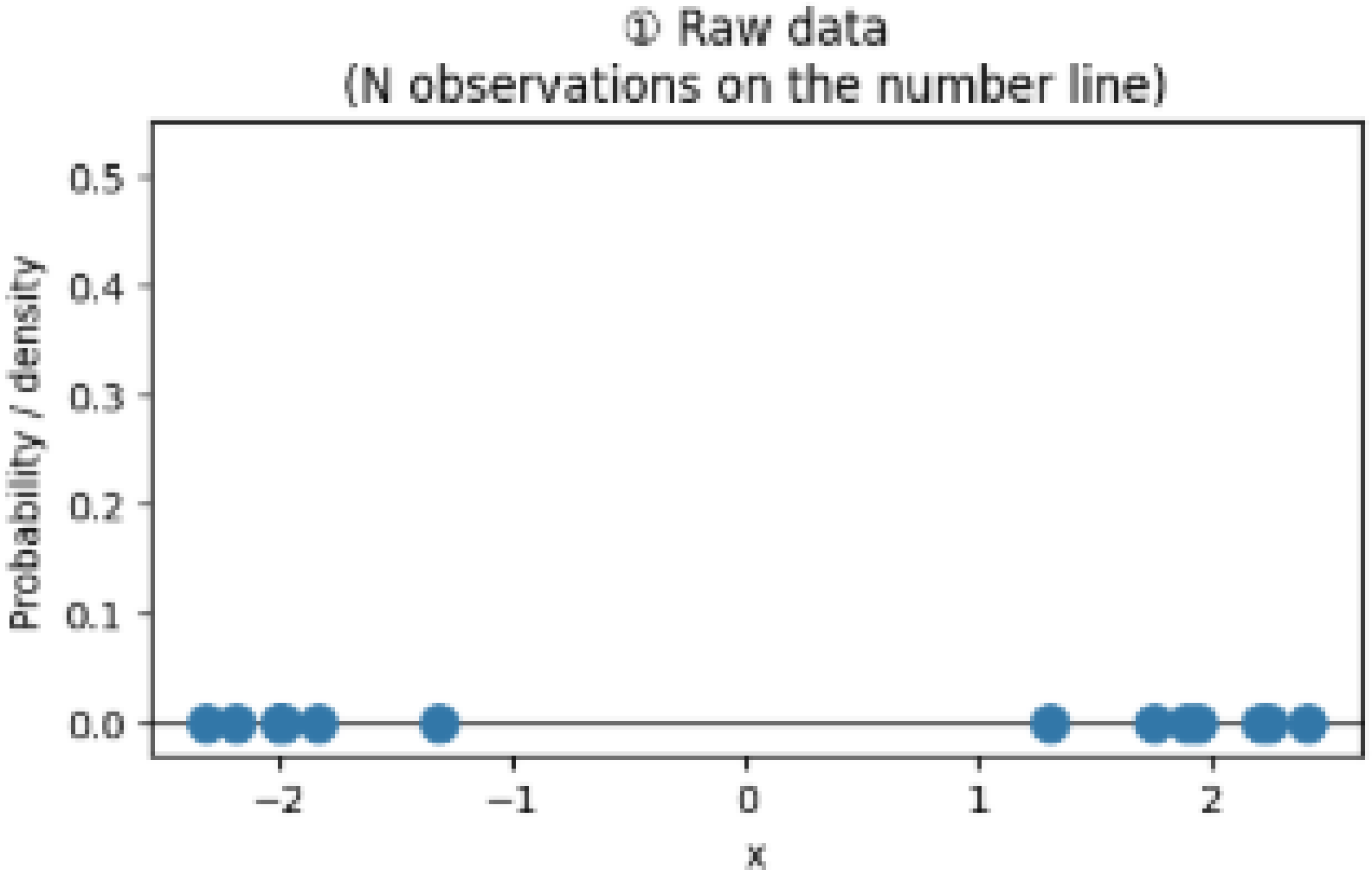
What if our data does not follow standard gaussian bell shape



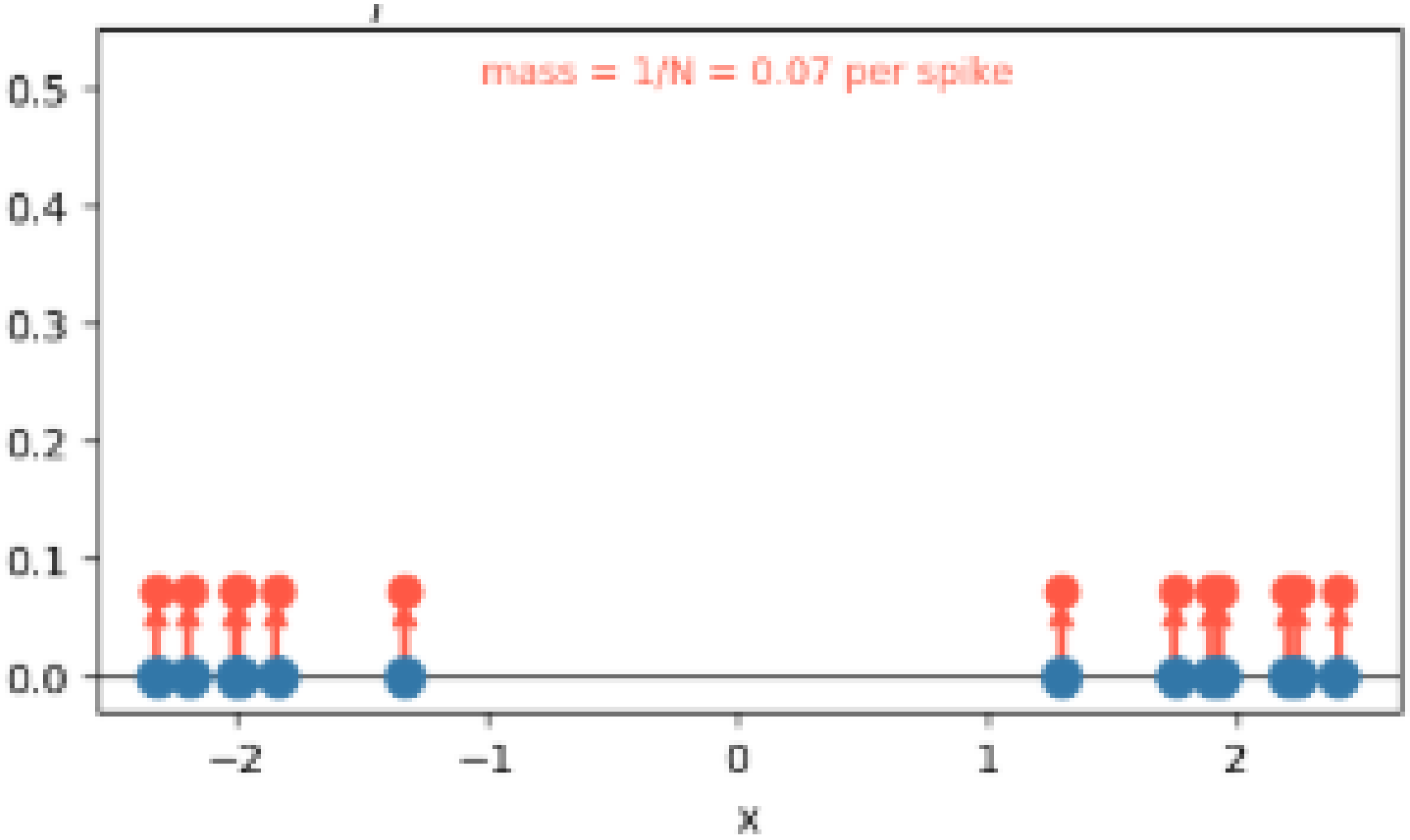
Two Spike shaped Data does not align with Bell-shaped Gaussian

Instead of Assuming a fixed shape – parametric, why not non-parametric?

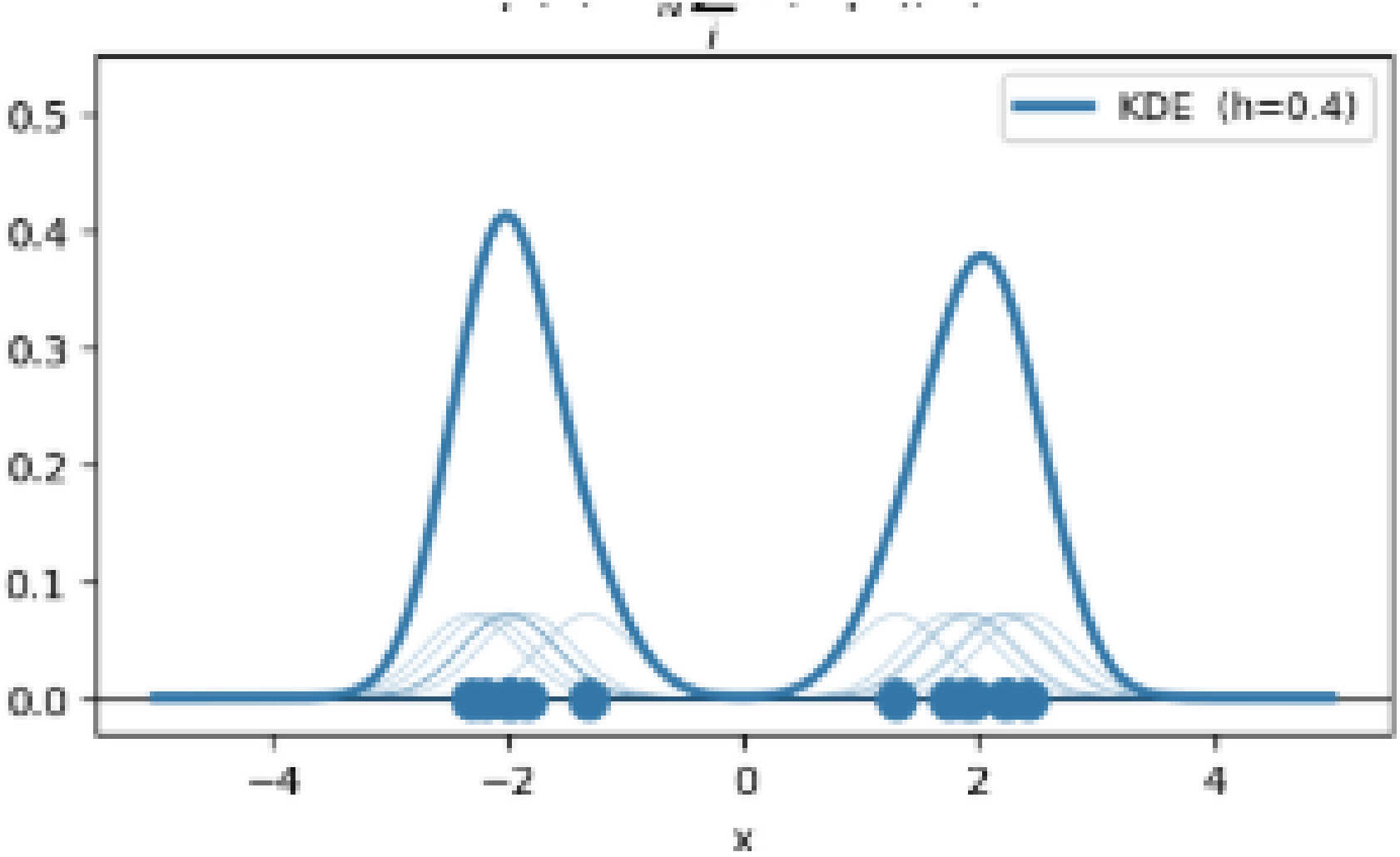
Data Decode Probability Mass



Data Decode Probability Mass – Dirac Distribution



Data Decode Probability Mass – Soft Dirac Distribution

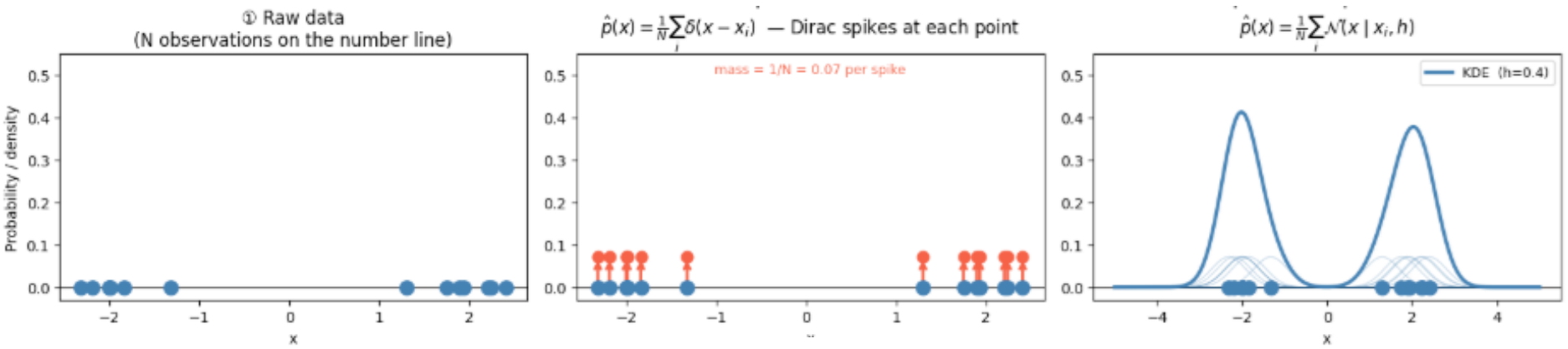


Non-Parametric Distribution

	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

From Raw Data to Dirac to Gaussian Kernal Density



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

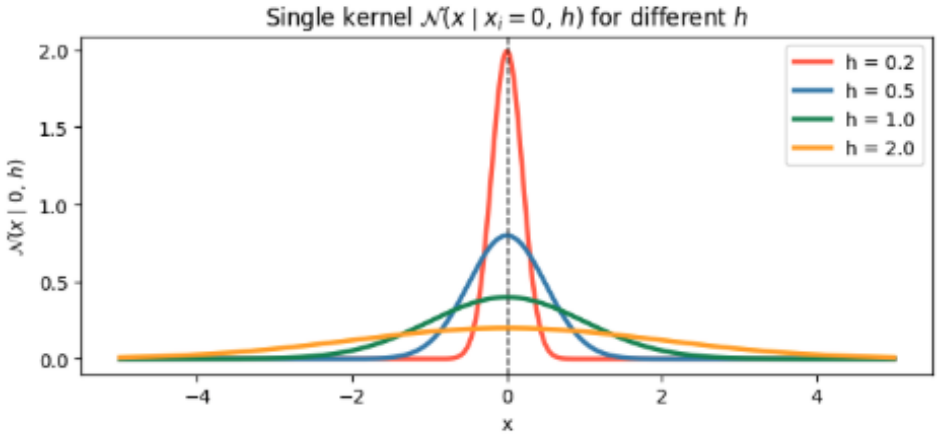
$$\hat{p}(x) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{N}(x \mid x_i,\, h)$$

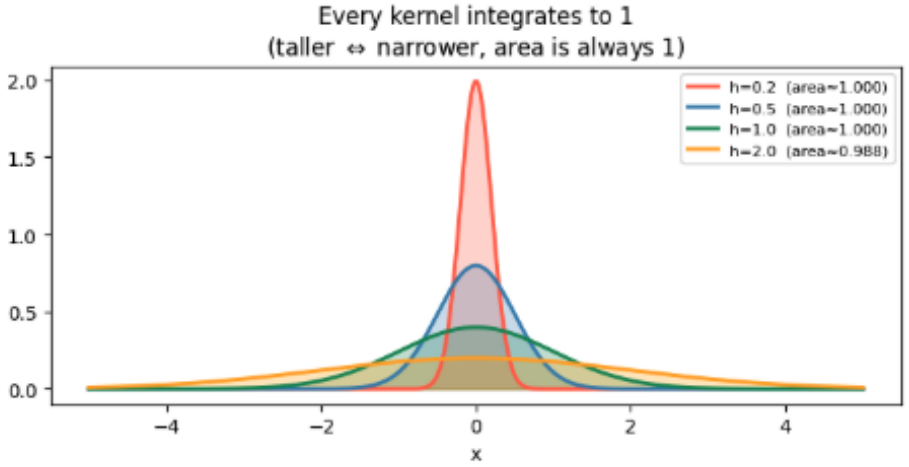
$$x = x_i, \delta = 1$$
$$x \neq x_i, \delta = 0$$

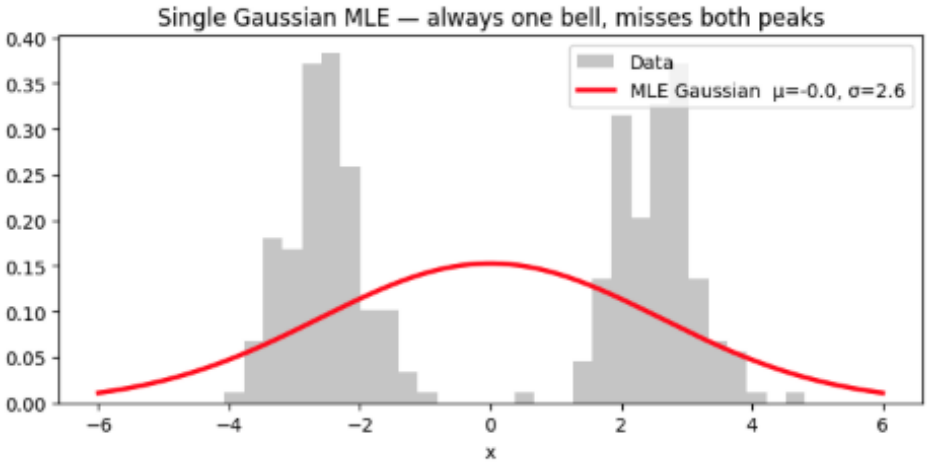
$$\hat{p}(x) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{h\sqrt{2\pi}}\exp\left(-\frac{(x - x_i)^2}{2h^2}\right)$$

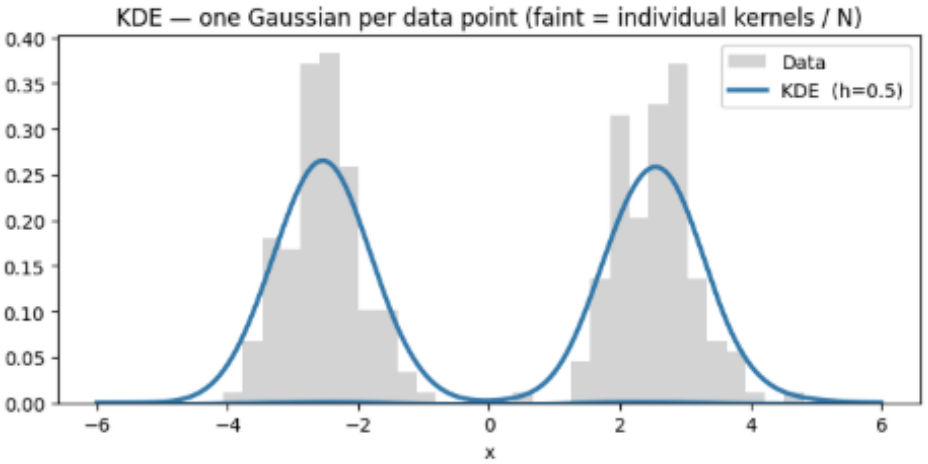
Limit	Kernel shape	KDE result
$h \to 0$	$\mathcal{N}(x \mid x_i, h) \to \delta(x - x_i)$	Recovers the spiky empirical distribution
$h \to \infty$	Kernel flattens to near-zero everywhere	Density smears out, all structure lost

From Raw Data to Dirac to Gaussian Kernal Density

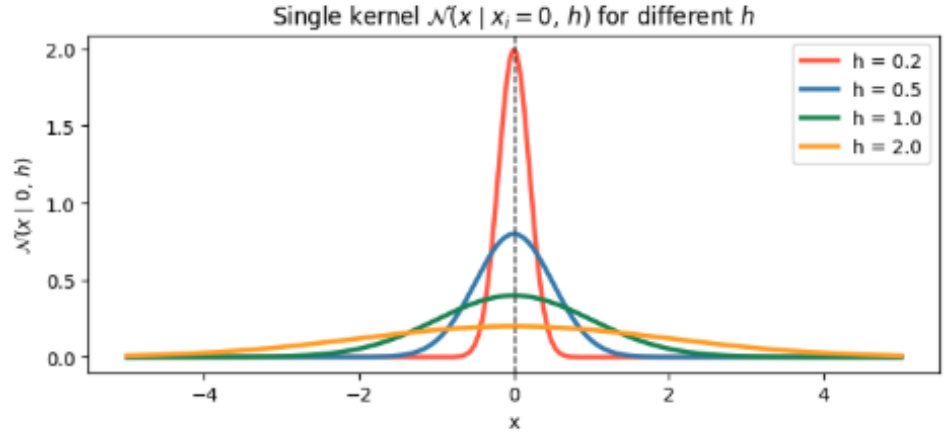


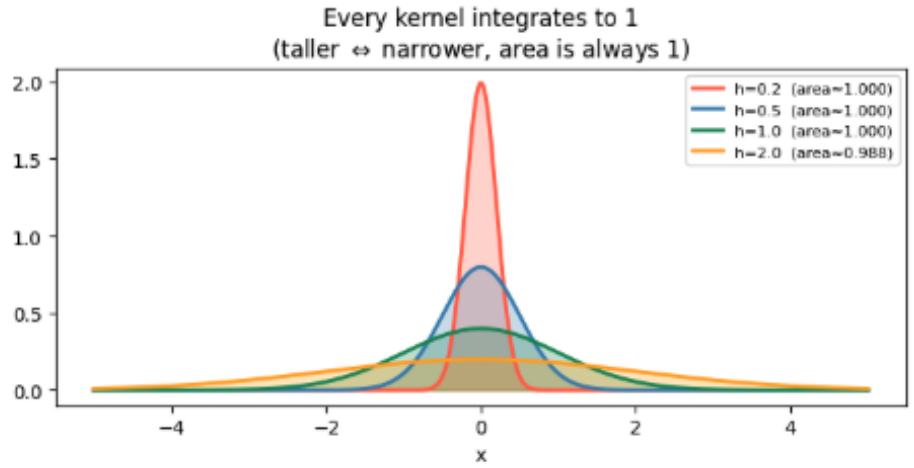


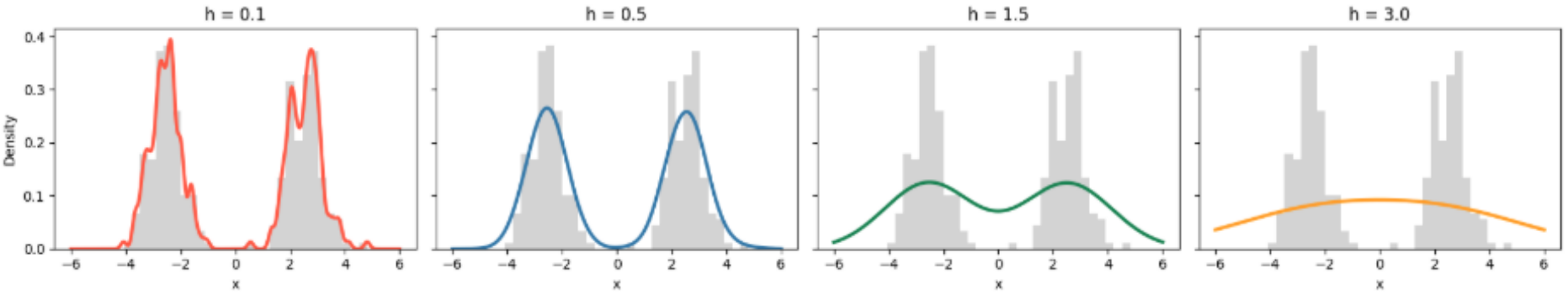




From Raw Data to Dirac to Gaussian Kernal Density





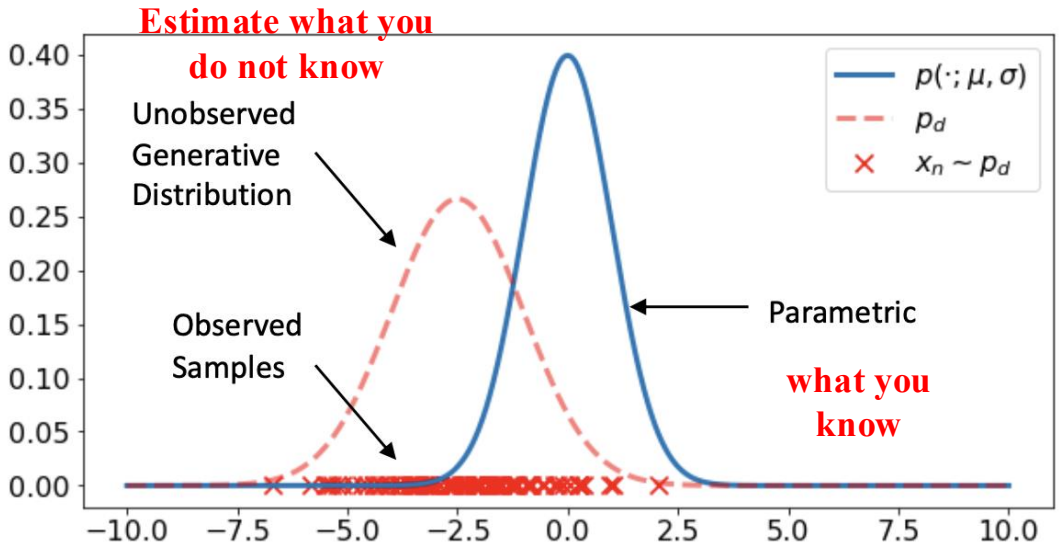


Limit	Kernel shape	KDE result
$h \to 0$	$\mathcal{N}(x \mid x_i, h) \to \delta(x - x_i)$	Recovers the spiky empirical distribution
$h \to \infty$	Kernel flattens to near-zero everywhere	Density smears out, all structure lost

Maximum Likelihood of Gaussian Kernal Density

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

$$\ell(h) = \sum_{i=1}^{N} \log \hat{p}(x_i)$$



How about we do the same thing for Gaussian Kernal Density?

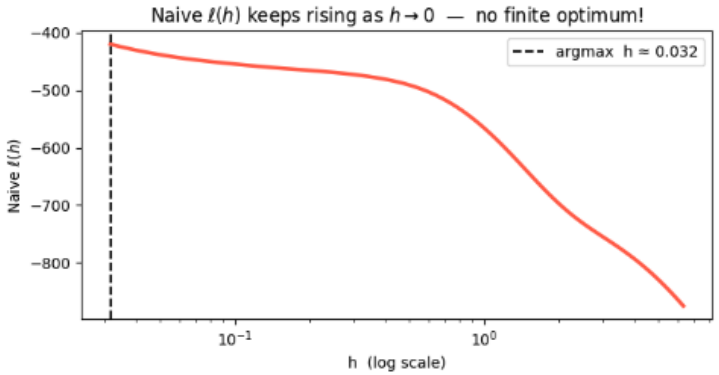
$$\hat{p}(x) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{N}(x \mid x_i, h)$$

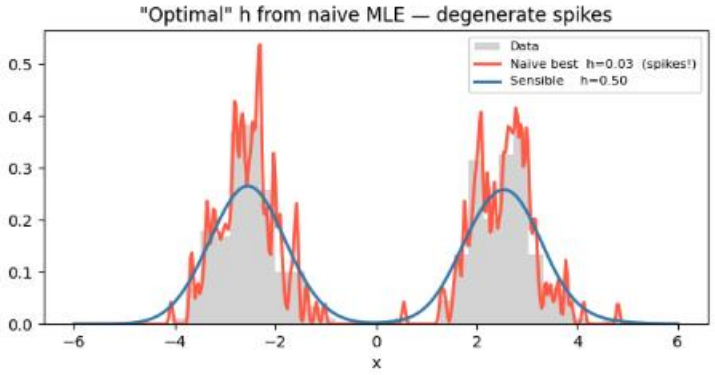
$$\hat{p}(x) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{h\sqrt{2\pi}} \exp\left(-\frac{(x - x_i)^2}{2h^2}\right)$$

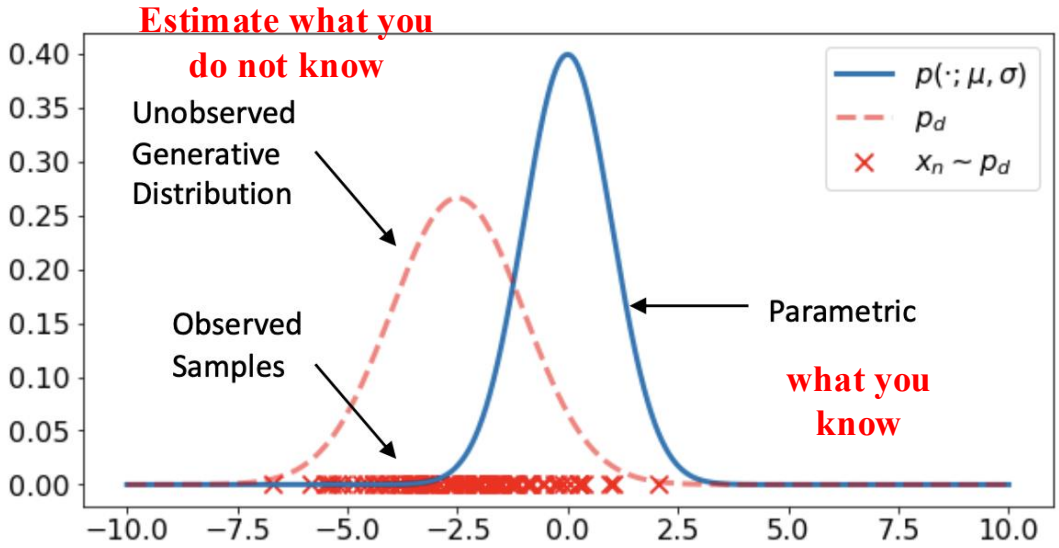
Maximum Likelihood of Gaussian Kernal Density

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

$$\ell(h) = \sum_{i=1}^{N} \log \hat{p}(x_i)$$







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

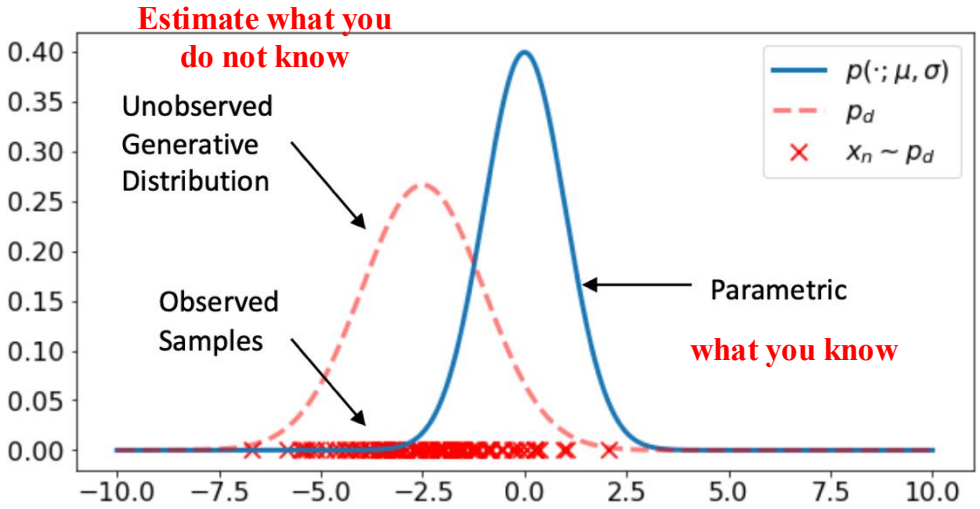
$$\hat{p}(x) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{h\sqrt{2\pi}} \exp\left(-\frac{(x - x_i)^2}{2h^2}\right)$$

Leave one out Estimation of Gaussian Kernal Density

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

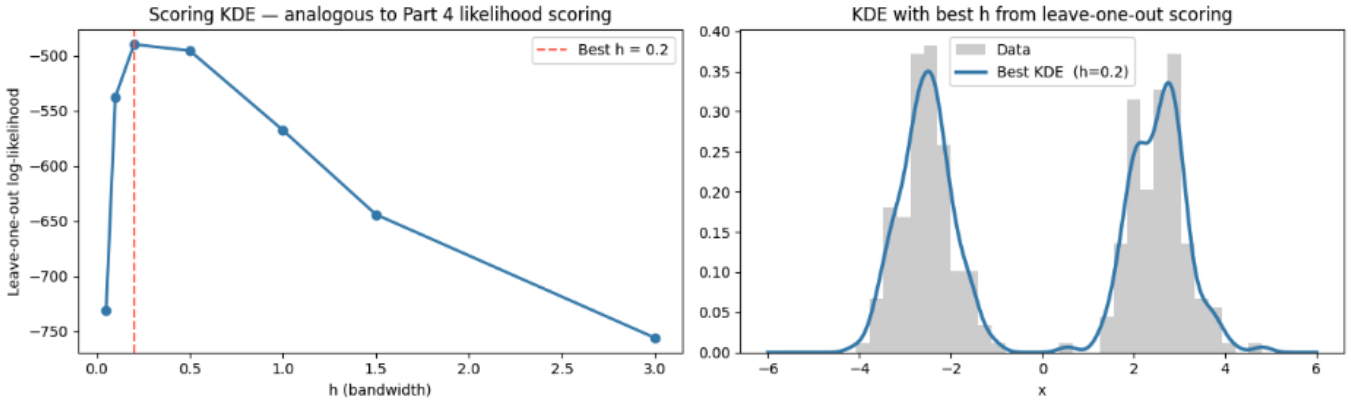
$$\ell(h) = \sum_{i=1}^{N} \log \hat{p}(x_i)$$

$$\ell_{\text{LOO}}(h) = \sum_{i=1}^{N} \log \left[\frac{1}{N-1} \sum_{j \neq i} \frac{1}{h\sqrt{2\pi}} \exp\left(-\frac{(x_i - x_j)^2}{2h^2} \right) \right]$$

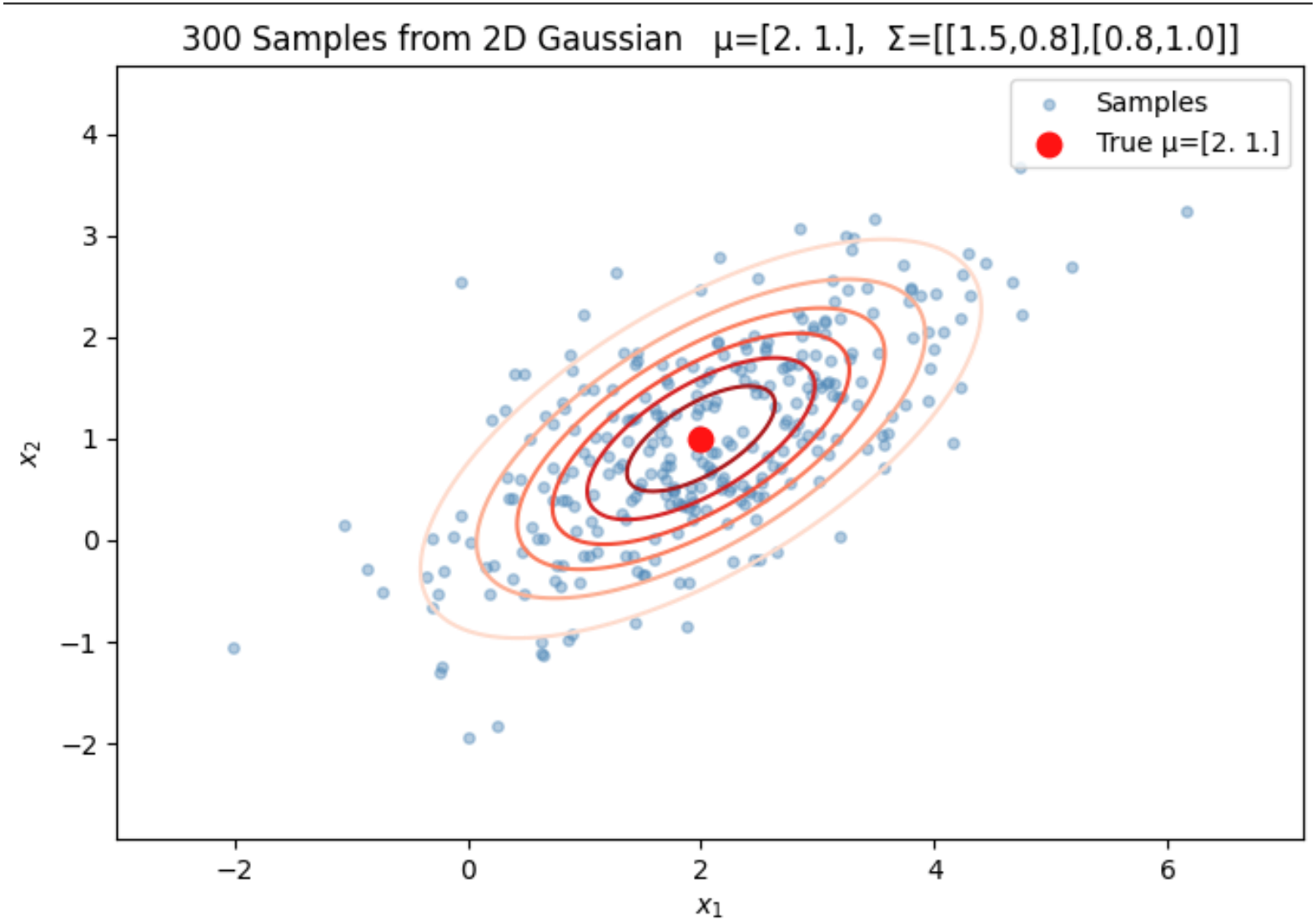


The fix: when scoring x_i, **exclude x_i from its own KDE**. This is called **leave-one-out** scoring, and it gives an honest estimate of how well the KDE generalises to data it has not seen.

The fix is **leave-one-out**: when scoring point x_i, compute the KDE using all *other* points $x_{j \neq i}$. This gives an honest measure of how well the KDE generalises to unseen data.



But what if our given data points are two dimensions?

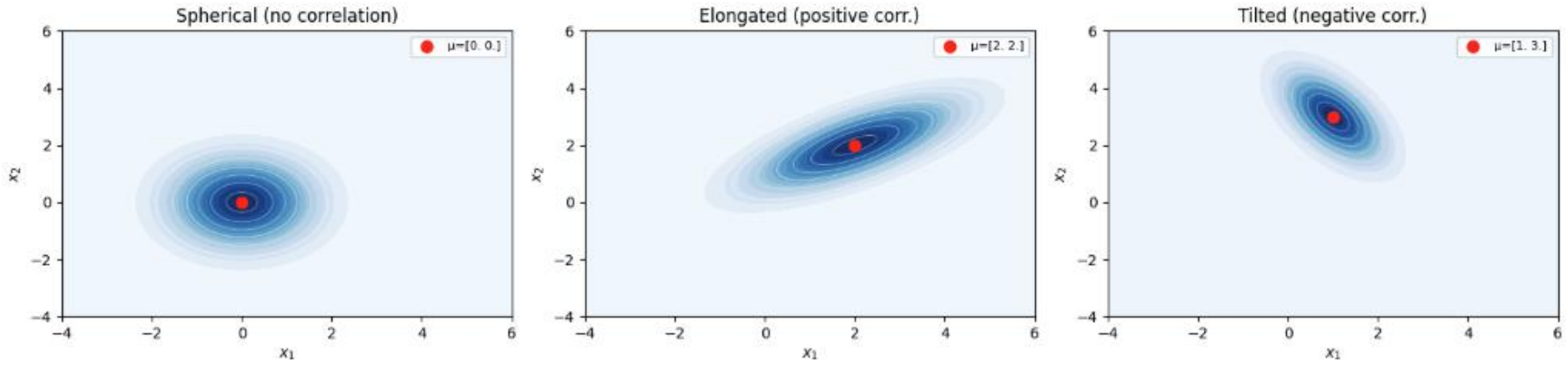


2D Gaussian Distribution

$$f(\mathbf{x}) = \frac{1}{2\pi|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$$

A mean vector $\boldsymbol{\mu} \in \mathbb{R}^2$

A covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^{2\times 2}$



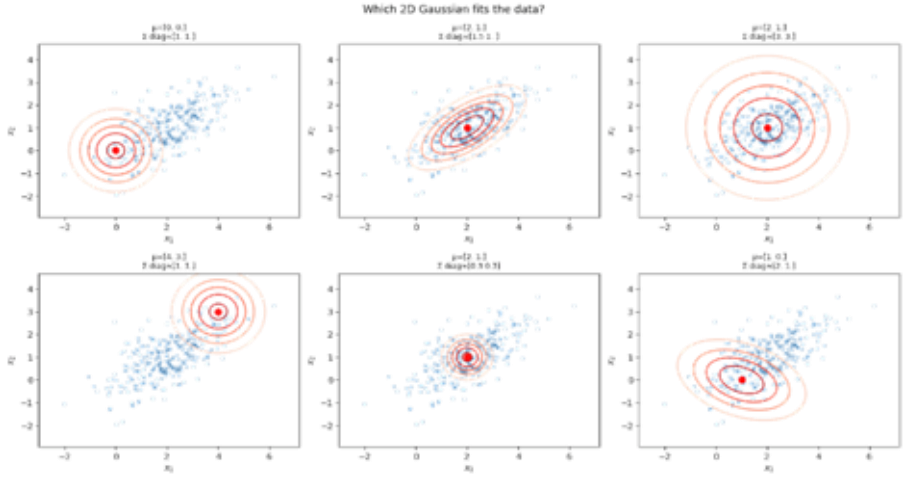
2D Gaussian Distribution

2D Gaussian Distribution

$$\log \mathcal{L}(\mu, \Sigma) = -\frac{N}{2} \log |2\pi\Sigma| - \frac{1}{2} \sum_{i=1}^{N} (\mathbf{x}_i - \mu)^\top \Sigma^{-1} (\mathbf{x}_i - \mu)$$

```python
def neg_log_likelihood_2d(mu, cov, data):
    """Negative log-likelihood for 2D Gaussian."""
    return -multivariate_normal(mean=mu, cov=cov).logpdf(data).sum()

print(f"{'Candidate':>30}  {'NLL':>12}")
print("-" * 46)
nll_scores = []
for mu_c, cov_c in candidates_2d:
    nll = neg_log_likelihood_2d(mu_c, cov_c, data_2d)
    nll_scores.append(nll)
    print(f"μ={mu_c}, Σ diag={np.diag(cov_c)}  {nll:>12.2f}")

best_idx = int(np.argmin(nll_scores))
print(f"\nBest candidate: μ={candidates_2d[best_idx][0]}, Σ diag={np.diag(candidates_2d[best_idx][1])}")
```

```
                     Candidate           NLL
----------------------------------------------
μ=[0. 0.], Σ diag=[1. 1.]       1669.82
μ=[2. 1.], Σ diag=[1.5 1. ]      811.87
μ=[2. 1.], Σ diag=[3. 3.]        999.94
μ=[4. 3.], Σ diag=[1. 1.]       2096.59
μ=[2. 1.], Σ diag=[0.3 0.3]      1380.06
μ=[1. 0.], Σ diag=[2. 1.]       1333.36

Best candidate: μ=[2. 1.], Σ diag=[1.5 1. ]
```

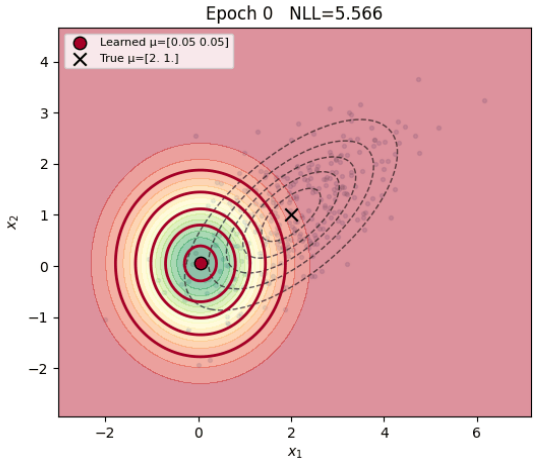


2D Gaussian Distribution Estimation

```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. ]]
```



Application – Single Digit Generation

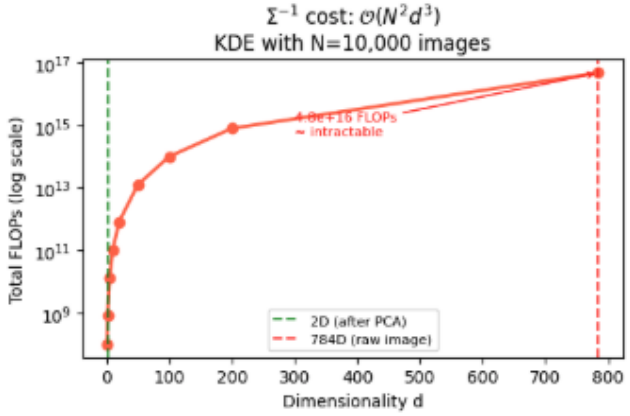
Dataset: 14780 images (6903 zeros, 7877 ones) $\mathbb{R}^{28\times28}$

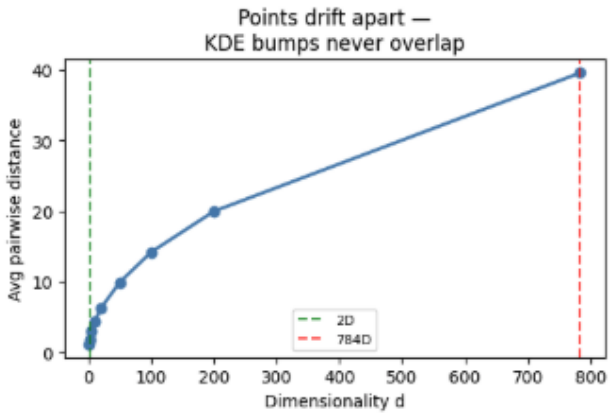
If we directly consider a 784 Dimensional Gaussian:

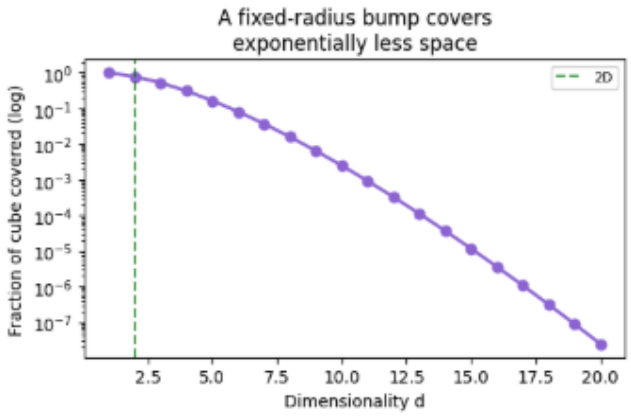
- **Too expensive** $f(\mathbf{x}) = \frac{1}{2\pi|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$ $\boldsymbol{\Sigma} \in \mathbb{R}^{784\times784}$ $\mathcal{O}(d^3)$
- **Curse of Dimensionality** ρ^d

Application – Single Digit Generation

Dataset: 14780 images (6903 zeros, 7877 ones) $\mathbb{R}^{28\times28}$







Application – Single Digit Generation

Dataset: 14780 images (6903 zeros, 7877 ones) $\mathbb{R}^{28\times28}$

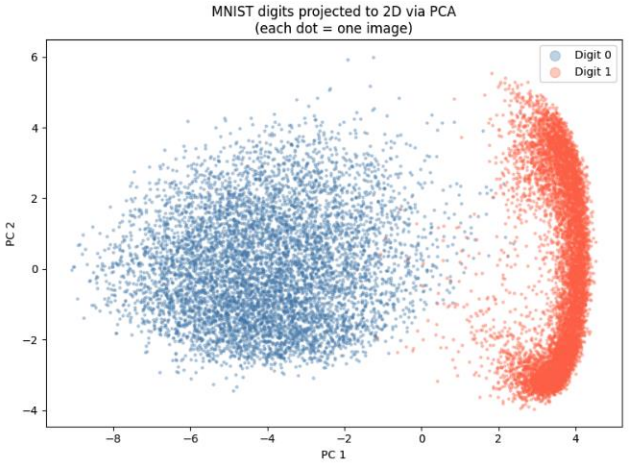
Most of the variation lives in a much lower-dimensional space.

We use PCA to compress each image to 2 numbers, fit a 2D KDE over that compressed space, sample from it, then uncompress back to pixels.

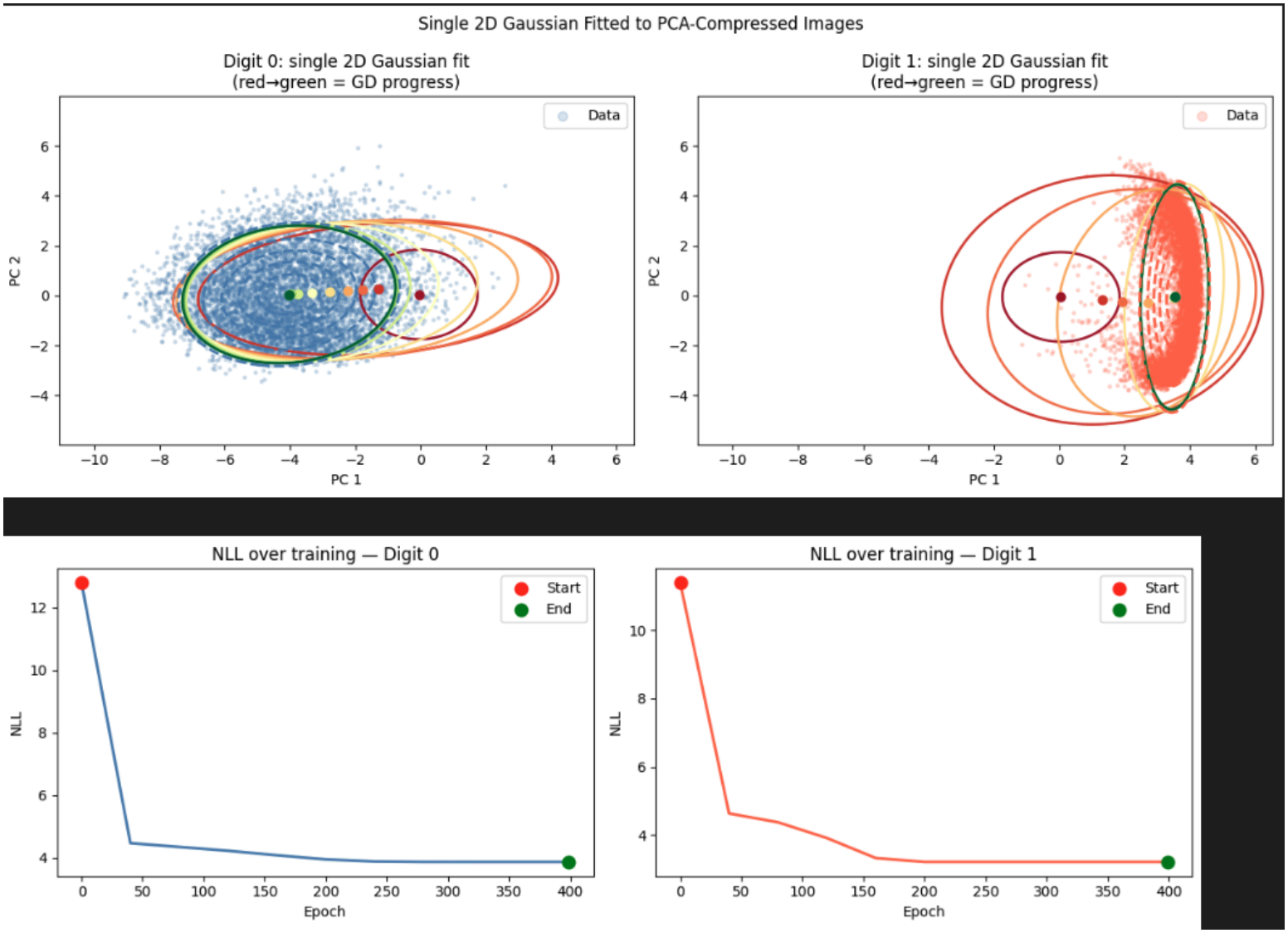
Application – Single Digit Generation

Dataset: 14780 images (6903 zeros, 7877 ones) $\mathbb{R}^{28\times28}$

```python
# ── Fit PCA: 784-D → 2-D ──────────────────────────────────────
pca = PCA(n_components=2, random_state=42)
X_2d = pca.fit_transform(X_raw)   # shape (N, 2)

print(f'Explained variance: PC1={pca.explained_variance_ratio_[0]:.1%},  '
      f'PC2={pca.explained_variance_ratio_[1]:.1%}')
print(f'Total captured: {pca.explained_variance_ratio_.sum():.1%} of variation')

# ── Scatter plot of the 2D PCA space ──────────────────────────
fig, ax = plt.subplots(figsize=(8, 6))
colors = {0: 'steelblue', 1: 'tomato'}
for digit in [0, 1]:
    idx = y_raw == digit
    ax.scatter(X_2d[idx, 0], X_2d[idx, 1],
               c=colors[digit], alpha=0.3, s=5, label=f'Digit {digit}')

ax.set_title('MNIST digits projected to 2D via PCA\n(each dot = one image)')
ax.set_xlabel('PC 1')
ax.set_ylabel('PC 2')
ax.legend(markerscale=4)
plt.tight_layout()
plt.show()
```

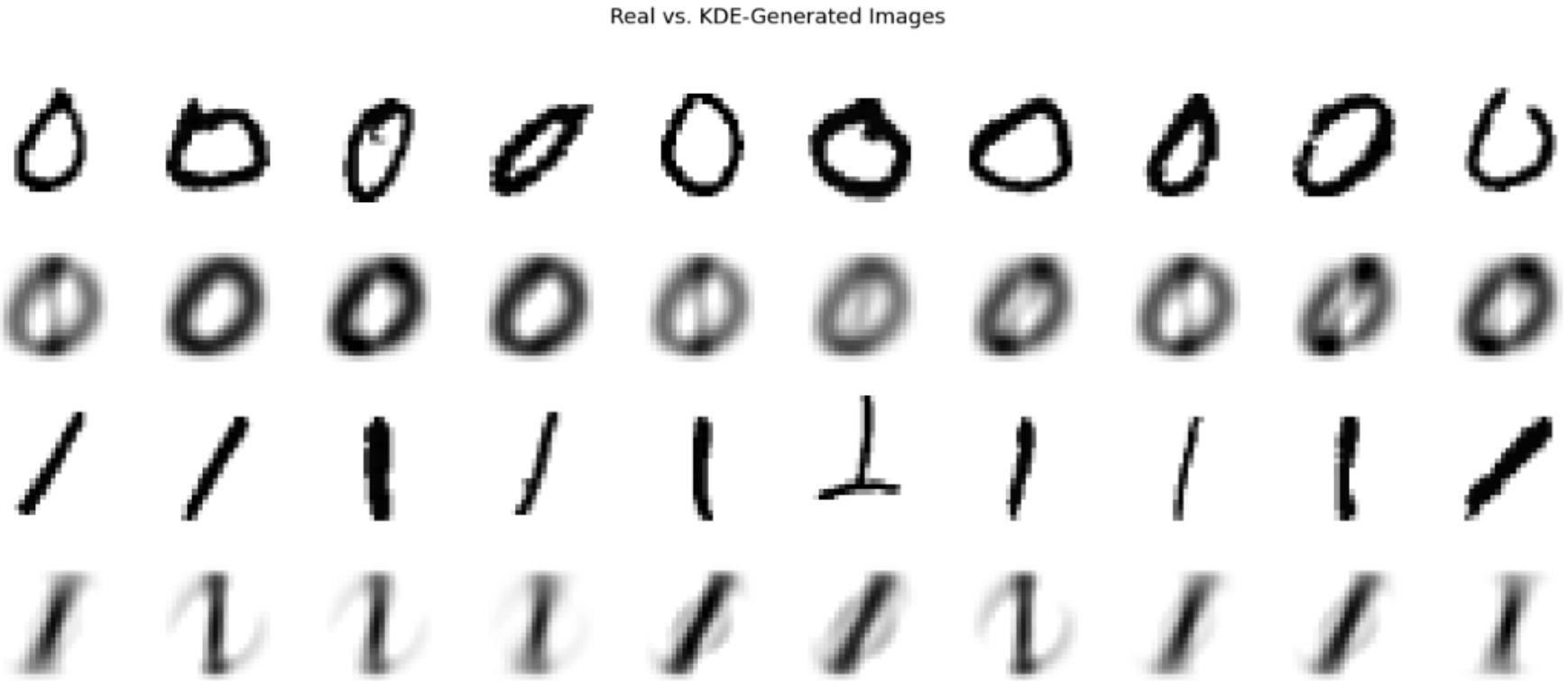


Application – Single Digit Generation



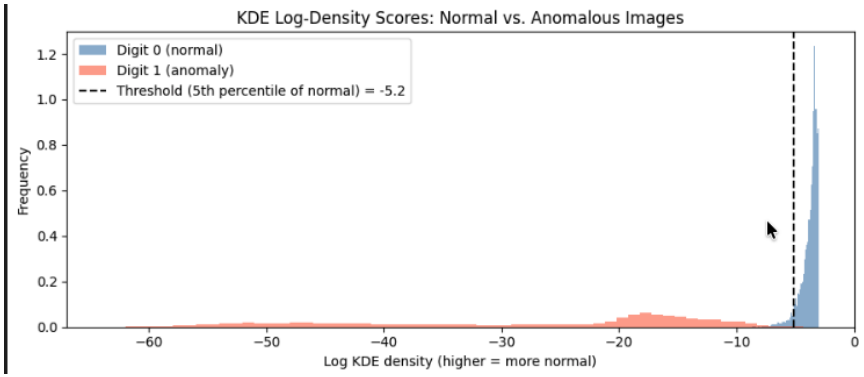
Application – Single Digit Generation

```
# ── Generated images via KDE sampling ──────────────────────
gen_2d = kdes[digit].resample(n_generate, seed=digit).T   # (n_generate, 2)
gen_images = pca.inverse_transform(gen_2d)                # (n_generate, 784)
gen_images = np.clip(gen_images, 0, 1)
```



Application – Single Digit Generation

```python
# — Score normal (digit 0) and anomalous (digit 1) images under KDE_0 ————
pts_0 = X_2d[y_raw == 0].T   # normal
pts_1 = X_2d[y_raw == 1].T   # anomalies from the perspective of KDE_0

scores_normal = np.log(kdes[0](pts_0) + 1e-300)   # log-density
scores_anomal = np.log(kdes[0](pts_1) + 1e-300)

# — Histogram of scores ————————————————————————————————————
fig, ax = plt.subplots(figsize=(9, 4))
ax.hist(scores_normal, bins=60, density=True, alpha=0.6,
        color='steelblue', label='Digit 0 (normal)')
ax.hist(scores_anomal, bins=60, density=True, alpha=0.6,
        color='tomato',    label='Digit 1 (anomaly)')
threshold = np.percentile(scores_normal, 5)   # flag bottom 5% as anomaly
ax.axvline(threshold, color='black', linestyle='—', linewidth=1.5,
           label=f'Threshold (5th percentile of normal) = {threshold:.1f}')
ax.set_title('KDE Log-Density Scores: Normal vs. Anomalous Images')
ax.set_xlabel('Log KDE density (higher = more normal)')
ax.set_ylabel('Frequency')
ax.legend()
plt.tight_layout()
plt.show()

tp = (scores_anomal < threshold).mean()   # fraction of digit-1 correctly flagged
fp = (scores_normal < threshold).mean()   # fraction of digit-0 incorrectly flagged
print(f'Detection rate  (digit 1 flagged as anomaly): {tp:.1%}')
print(f'False alarm rate (digit 0 flagged as anomaly): {fp:.1%}')
print()
print('A simple KDE threshold already separates the two digit classes reasonably well.')
print('No labels needed during training — just the normal data distribution.')
```



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

Given a fixed mean and std, what is the likelihood of observing such instance?

High Likelihood – Normal

Low Likelihood - Abnormal

QA