Member-only story

Decoding Latent Variables: Comparing Bayesian, EM, and VAE Approaches

A Deep Dive into Mathematical Foundations, A/B Testing Applications, and Choosing the Right Method for Your Data Challenges.

Shenggang Li · Following

Published in Towards AI · 27 min read · Dec 14, 2024

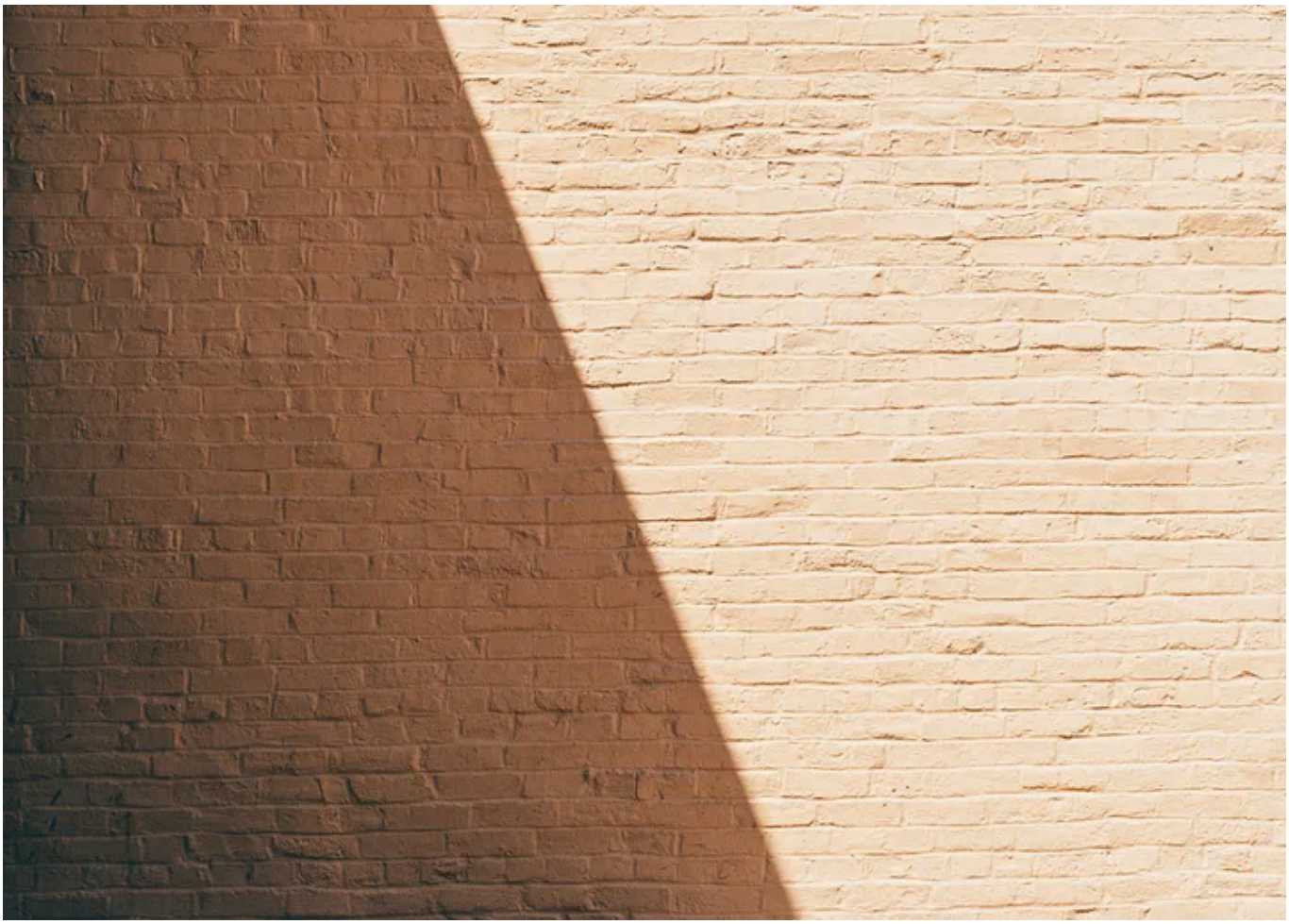


Photo by Khara Woods on Unsplash

Ever wondered how to uncover hidden details in your data when things are not fully clear? Consider running an A/B test for a marketing campaign — sales numbers may be available, but the true impact could remain hidden. This paper explores three methods to address such challenges: Expectation-Maximization (EM), Bayesian estimation, and Variational Autoencoders (VAEs), each offering unique insights into latent variable analysis.

The EM algorithm addresses missing information by iteratively guessing and refining hidden details. In A/B testing, it is effective for both masked and fully observed treatments, bridging gaps in incomplete data. Bayesian estimation incorporates a probabilistic framework, combining prior knowledge with observed data to reveal not only results but also confidence levels, making it ideal for comparing group performance and probabilities.

VAEs, a prominent method in AI, are widely recognized for recreating images and generating data. However, they go beyond these applications by uncovering hidden patterns, creating a "latent space", and simulating "what-if" scenarios. Unlike EM or Bayesian methods, VAEs generate new possibilities, making them particularly effective for exploratory analysis.

This paper examines how VAEs connect to traditional methods like EM and Bayesian estimation. Through experiments, their strengths are compared, and their relationships are explored. By the end, readers will understand when and how to apply each method effectively. Let us begin!

Expectation-Maximization (EM)

Theory Illustrated with an Example

Imagine you're analyzing the stock prices of three industry sectors, and you know the total daily price movement for each sector (X={5,6,7}, in dollars). However, you don't know how these movements are split between the individual stocks in each sector. The hidden contribution of each stock is like the latent variable Z — it's what drives the total sector price, but you can't see it directly.

Now, let's assume X is normally distributed, given Z:

$$X \mid Z \sim \mathcal{N}(\mu + Z, \sigma^2)$$

Z is also normally distributed:

$$Z \sim \mathcal{N}(\nu, \tau^2)$$

Here, the parameters are

$$\theta = (\mu, \sigma^2, \nu, \tau^2)$$

The EM algorithm's job is to estimate the parameters while figuring out how much each hidden stock (Z) contributes to the sector's total price (X). It's like uncovering the invisible "price pushes" from individual stocks that add up to the sector's overall movement. This can help you understand the hidden dynamics driving the market.

Now, let's see how EM works. Start by making initial guesses for the parameters:

$$\mu^{(0)} = 4,$$
$$\sigma^{2(0)} = 1$$
$$\nu^{(0)} = 0,$$
$$\tau^{2(0)} = 1$$

In the E-step, we compute the expected value of the latent variable Z and its squared value, given the observed data X and the current parameters θ^(t). From Bayes' rule, we have the Posterior Distribution of Z:

$$p(Z \mid X, \theta) \propto p(X \mid Z, \theta) p(Z \mid \theta)$$

Using the normal density functions:

$$p(X \mid Z, \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(X - (\mu + Z))^2}{2\sigma^2}\right)$$

$$p(Z \mid \theta) = \frac{1}{\sqrt{2\pi\tau^2}} \exp\left(-\frac{(Z - \nu)^2}{2\tau^2}\right)$$

Then we have:

$$p(Z \mid X, \theta) \sim \mathcal{N}(\tilde{\nu}, \tilde{\tau}^2)$$

This means:

$$\tilde{\nu} = \frac{\sigma^2 \nu + \tau^2 (X - \mu)}{\sigma^2 + \tau^2}, \quad \tilde{\tau}^2 = \frac{\sigma^2 \tau^2}{\sigma^2 + \tau^2}$$

For each observed *X_i*, we calculate:

$$\mathbb{E}[Z \mid X_i] = \tilde{\nu}$$

$$\mathbb{E}[Z^2 \mid X_i] = \mathrm{Var}(Z \mid X_i) + (\mathbb{E}[Z \mid X_i])^2 = \tilde{\tau}^2 + \tilde{\nu}^2$$

For example, using *X_1 = 5* and

$$\mu^{(0)} = 4,\ \sigma^{2(0)} = 1,\ \nu^{(0)} = 0,\ \tau^{2(0)} = 1$$

$$\tilde{\nu} = \frac{1 \cdot 0 + 1 \cdot (5 - 4)}{1 + 1} = \frac{1}{2} = 0.5$$

$$\tilde{\tau}^2 = \frac{1 \cdot 1}{1 + 1} = 0.5$$

Then, we have:

$$\mathbb{E}[Z \mid X_1] = 0.5, \quad \mathbb{E}[Z^2 \mid X_1] = 0.5^2 + 0.5 = 0.75$$

Repeat this calculation for *X_2 = 6* and *X_3 = 7*.

In the M-step, update the parameters *θ = (μ, σ^2, ν, τ^2)* by maximizing the expected complete-data log-likelihood:

$$Q(\theta \mid \theta^{(t)}) = \mathbb{E}_{Z|X,\theta^{(t)}} \left[\log p(X, Z \mid \theta)\right]$$

The complete-data log-likelihood is

$$\log p(X, Z \mid \theta) = \sum_{i=1}^{n} \left[-\frac{1}{2} \log(2\pi\sigma^2) - \frac{(X_i - (\mu + Z_i))^2}{2\sigma^2} \right] + \sum_{i=1}^{n} \left[-\frac{1}{2} \log(2\pi\tau^2) - \frac{(Z_i - \nu)^2}{2\tau^2} \right]$$

Differentiating w.r.t. μ, we can update μ:

$$\mu^{(t+1)} = \frac{1}{n} \sum_{i=1}^{n} (X_i - \mathbb{E}[Z \mid X_i])$$

Example for $X = \{5, 6, 7\}$, using $E[Z \mid X\_i] = \{0.5, 0.6, 0.7\}$:

$$\mu^{(t+1)} = \frac{1}{3} \left[(5 - 0.5) + (6 - 0.6) + (7 - 0.7) \right] = \frac{1}{3} (4.5 + 5.4 + 6.3) = 5.4$$

Then we update ν (differentiating w.r.t. ν similarly):

$$\nu^{(t+1)} = \frac{1}{n} \sum_{i=1}^{n} \mathbb{E}[Z \mid X_i]$$

For example:

$$\nu^{(t+1)} = \frac{1}{3} (0.5 + 0.6 + 0.7) = 0.6$$

Finally we update τ^2:

$$\tau^{2(t+1)} = \frac{1}{n} \sum_{i=1}^{n} \left[\mathbb{E}[Z^2 \mid X_i] - (\nu^{(t+1)})^2 \right]$$

Repeat the E-step and M-step until the parameter estimates stabilize. For example, after a few iterations, the estimates might converge to:

$$\mu = 5.5,\ \sigma^2 = 0.8,\ \nu = 0.6,\ \tau^2 = 0.7$$

Summary of EM Algorithm

> *The goal of EM is to estimate the parameter θ of a distribution p(X | θ), where X is the observed data. However, X is influenced by a hidden (latent) factor Z, which we can't observe directly. This hidden Z complicates things.*

$$p(X \mid \boldsymbol{\theta}) = \int p(X, Z \mid \boldsymbol{\theta})\, dZ$$

> *Here's the challenge: this integral is often too hard to compute analytically because* p(X, Z | θ) *doesn't have a simple closed form. EM solves this problem by approximating the marginal likelihood* p(X | θ) *iteratively.*

Here's how it works:

$$\theta^* = \arg\max_\theta \log p(X|\theta), \quad p(X|\theta) = \int p(X, Z|\theta)\, dZ$$

Since $p(X|\theta)$ is often hard to compute, EM iteratively optimizes the complete-data log-likelihood $\log(p(X, Z|\theta))$ in two steps:

E-Step: Compute the expected complete-data log-likelihood:

$$Q(\theta|\theta^{(t)}) = \mathbb{E}_{Z\sim p(Z|X,\theta^{(t)})}[\log p(X, Z|\theta)]$$

M-Step: Maximize $Q(\theta|\theta^{(t)})$ to update parameters:

$$\theta^{(t+1)} = \arg\max_\theta Q(\theta|\theta^{(t)})$$

Repeat until convergence to get the final estimates of θ and the latent variable distribution $p(Z|X, \theta^*)$.

EM for A/B Testing

The Expectation-Maximization (EM) algorithm can be applied to A/B testing scenarios. This section outlines two approaches for implementing the EM

algorithm in A/B testing, providing mathematical steps and highlighting the benefits of using EM in this context.

Suppose we are running an A/B test for a marketing campaign to compare the effectiveness of two groups:

- Control Group: Customers who received no special campaign (e.g., standard marketing approach).
- Treatment Group: Customers who received the campaign (e.g., personalized marketing).

Using EM for A/B Testing with Masked Campaign Treatment

In this approach, the Campaign Treatment information is treated as a latent variable (masked). The observed data consists of the sales values X, while the probabilities $p(X \mid \theta)$ and model parameters θ (such as group-specific means and variances) are estimated iteratively using the EM algorithm. This method is useful when group assignments (e.g., control or treatment) are incomplete or uncertain, since it can infer missing information while estimating the model parameters.

Setup:

X: Observed sales data.

Z: Latent variable representing the group assignment (control or treatment).

Estimate parameter $\theta = (\mu\_C, \mu\_T, \sigma\_C, \sigma\_T, \pi\_C, \pi\_T)$, where:

- μ_C: Mean and variance of sales for the control group.

- *μ_T, σ_T*: Mean and STD of sales for the treatment group.
- *π_C, π_T*: Mixing proportions *(π_C + π_T=1)*.

Step-by-Step EM Algorithm:

1. Initialize Parameters:

Initial guesses for

$$\theta^{(0)} = (\mu_C^{(0)}, \mu_T^{(0)}, \sigma_C^{2(0)}, \sigma_T^{2(0)}, \pi_C^{(0)}, \pi_T^{(0)})$$

2. E-Step: Compute Expected Group Assignment:

For each data point *x_i*, compute the posterior probability that it belongs to the control or treatment group:

$$\gamma_{i,C} = P(Z_i = C|x_i, \theta^{(t)}) = \frac{\pi_C^{(t)} \mathcal{N}(x_i|\mu_C^{(t)}, \sigma_C^{2(t)})}{\pi_C^{(t)} \mathcal{N}(x_i|\mu_C^{(t)}, \sigma_C^{2(t)}) + \pi_T^{(t)} \mathcal{N}(x_i|\mu_T^{(t)}, \sigma_T^{2(t)})}$$

$$\gamma_{i,T} = 1 - \gamma_{i,C}$$

Here, *N(x | μ,σ^2)* is the normal density function.

3. M-Step: Update Parameters:

$$\pi_C^{(t+1)} = \frac{1}{n}\sum_{i=1}^{n}\gamma_{i,C}, \quad \pi_T^{(t+1)} = 1 - \pi_C^{(t+1)}$$

$$\mu_C^{(t+1)} = \frac{\sum_{i=1}^{n}\gamma_{i,C}x_i}{\sum_{i=1}^{n}\gamma_{i,C}}, \quad \mu_T^{(t+1)} = \frac{\sum_{i=1}^{n}\gamma_{i,T}x_i}{\sum_{i=1}^{n}\gamma_{i,T}}$$

$$\sigma_C^{2(t+1)} = \frac{\sum_{i=1}^{n}\gamma_{i,C}(x_i - \mu_C^{(t+1)})^2}{\sum_{i=1}^{n}\gamma_{i,C}}, \quad \sigma_T^{2(t+1)} = \frac{\sum_{i=1}^{n}\gamma_{i,T}(x_i - \mu_T^{(t+1)})^2}{\sum_{i=1}^{n}\gamma_{i,T}}$$

5. Repeat Until Convergence:

Iterate between the E-step and M-step until the parameters stabilize.

A/B Testing Inference:

After convergence, compare μ_C and μ_T to evaluate the effectiveness of the treatment group relative to the control group.

Code Experiment for Masked Campaign Treatment

```
import numpy as np
import pandas as pd
from scipy.stats import norm, ttest_ind

## read data
data_abtest = pd.read_csv('ab_sales.csv')
```

```
# Initialize Parameters
mu_C = np.random.uniform(40, 60)  # Initial guess for Control mean
sigma_C = np.random.uniform(5, 15)  # Initial guess for Control STD
mu_T = np.random.uniform(50, 70)  # Initial guess for Treatment mean
sigma_T = np.random.uniform(10, 20)  # Initial guess for Treatment STD
pi_C = 0.5  # Initial guess for mixing proportion of Control group
pi_T = 0.5  # Initial guess for mixing proportion of Treatment group

# Define the log-likelihood function
def log_likelihood(data, mu_C, sigma_C, mu_T, sigma_T, pi_C, pi_T):
    likelihood_control = pi_C * norm.pdf(data, mu_C, sigma_C)
    likelihood_treatment = pi_T * norm.pdf(data, mu_T, sigma_T)
    return np.sum(np.log(likelihood_control + likelihood_treatment))

# EM Algorithm
max_iter = 100  # Maximum number of iterations
tolerance = 1e-6  # Convergence criterion

log_likelihoods = []
data = data_abtest["sales"]

for iteration in range(max_iter):
    # E-Step: Calculate responsibilities (posterior probabilities)
    resp_control = pi_C * norm.pdf(data, mu_C, sigma_C)
    resp_treatment = pi_T * norm.pdf(data, mu_T, sigma_T)
    total_resp = resp_control + resp_treatment

    gamma_control = resp_control / total_resp  # Responsibility for Control grou
    gamma_treatment = resp_treatment / total_resp  # Responsibility for Treatmen

    # M-Step: Update parameters
    N_C = np.sum(gamma_control)  # Effective number of points in Control group
    N_T = np.sum(gamma_treatment)  # Effective number of points in Treatment gro

    # Update means
    mu_C = np.sum(gamma_control * data) / N_C
    mu_T = np.sum(gamma_treatment * data) / N_T

    # Update standard deviations
    sigma_C = np.sqrt(np.sum(gamma_control * (data - mu_C)**2) / N_C)
    sigma_T = np.sqrt(np.sum(gamma_treatment * (data - mu_T)**2) / N_T)

    # Update mixing proportions
    pi_C = N_C / n_samples
    pi_T = N_T / n_samples

    # Compute log-likelihood and check for convergence
    current_log_likelihood = log_likelihood(data, mu_C, sigma_C, mu_T, sigma_T,
    log_likelihoods.append(current_log_likelihood)
```

```
        if len(log_likelihoods) > 1 and abs(log_likelihoods[-1] - log_likelihoods[-2
            print(f"Converged at iteration {iteration}")
            break

    # Output the results
    print(f"Final parameters:")
    print(f"Control Mean (mu_C): {mu_C:.2f}, Control STD (sigma_C): {sigma_C:.2f}, M
    print(f"Treatment Mean (mu_T): {mu_T:.2f}, Treatment STD (sigma_T): {sigma_T:.2f

    # Compare the means for inference
    if mu_T > mu_C:
        print("The Treatment group outperforms the Control group.")
    else:
        print("The Control group outperforms the Treatment group.")

    # Step 4: Perform a T-test for comparison
    labeled_data = data_with_labels.dropna()
    control_group = labeled_data[labeled_data["group"] == 0]["sales"]
    treatment_group = labeled_data[labeled_data["group"] == 1]["sales"]

    # Perform two-sample T-test
    t_stat, p_value = ttest_ind(control_group, treatment_group, equal_var=False)
    print(f"T-test results: t-statistic = {t_stat:.2f}, p-value = {p_value:.4f}")

    if p_value < 0.05:
        print("T-test: The difference between groups is statistically significant.")
    else:
        print("T-test: The difference between groups is not statistically significan
```

```
Final parameters:
Control Mean (mu_C): 48.87, Control STD (sigma_C): 9.52, Mixing Proportion (pi_C
Treatment Mean (mu_T): 58.13, Treatment STD (sigma_T): 11.38, Mixing Proportion

The Treatment group outperforms the Control group.
T-test results: t-statistic = -9.08, p-value = 0.0000
T-test: The difference between groups is statistically significant.
```

Explanation of Results

EM Results:

EM handled the missing labels and estimated parameters using all the data. The treatment group has higher mean sales, showing it outperforms the control group.

T-Test Results:

- t-statistic = *-9.08*, p-value = *0.0000*.

The T-test confirms a significant difference between the groups but only uses fully labeled data, ignoring *20%* of the dataset.

Why EM Wins Here:

- Missing Labels: EM used all *1200* rows, while the T-test dropped *20%* of the data.
- Mixing Proportions: EM captured proportions (*56%* control, *44%* treatment), adding insights beyond means.
- Group Variability: EM estimated group-specific STDs (Control: *9.52*, Treatment: *11.38*), which the T-test cannot do.

EM is better here because it handles missing labels, gives richer group details, and provides accurate results with overlapping data distributions.

Using EM for A/B Testing with Campaign Treatment as Observed Data

In this scenario, both Campaign Treatment and sales data are observed *(Z= (x, g))*, where *g* indicates group membership: Control or Treatment). EM is

used to estimate the parameters of $p(Z \mid \theta)$.

Setup:

$Z = (x, g)$: Observed data

Likelihood: $p(Z \mid \theta) = p(x \mid g, \theta) p(g \mid \theta)$

Estimate parameters: $\theta = (\mu\_C, \mu\_T, \sigma\_C, \sigma\_T)$

Step-by-Step EM Algorithm:

1. **E-Step: Compute Group Assignment Probabilities:**

The group assignments g are fully observed, so the E-step simply uses these observed values.

2. M-Step: Update Parameters:

Update $\mu\_C, \mu\_T, \sigma\_C, \sigma\_T$ based on the group assignments:

$$\mu_C^{(t+1)} = \frac{\sum_{g_i=C} x_i}{n_C}, \quad \mu_T^{(t+1)} = \frac{\sum_{g_i=T} x_i}{n_T}$$

$$\sigma_C^{2(t+1)} = \frac{\sum_{g_i=C} (x_i - \mu_C^{(t+1)})^2}{n_C}, \quad \sigma_T^{2(t+1)} = \frac{\sum_{g_i=T} (x_i - \mu_T^{(t+1)})^2}{n_T}$$

3. Repeat Until Convergence:

Iterate the steps until the parameter estimates stabilize.

A/B Testing Inference:

We can similarly compare μ_C and μ_T to evaluate the effect of the treatment group.

Code Experiment for Observed Campaign Treatment

```
import numpy as np
import pandas as pd
from scipy.stats import norm

## use the same data
data = data_abtest.copy()

# Initialize Parameters
mu_C = np.random.uniform(40, 60)  # Initial guess for Control mean
sigma_C = np.random.uniform(5, 15)  # Initial guess for Control STD
mu_T = np.random.uniform(50, 70)  # Initial guess for Treatment mean
sigma_T = np.random.uniform(10, 20)  # Initial guess for Treatment STD

# Define the log-likelihood function
def log_likelihood(data, mu_C, sigma_C, mu_T, sigma_T):
    likelihood_control = norm.pdf(data[data["group"] == 0]["sales"], mu_C, sigma
    likelihood_treatment = norm.pdf(data[data["group"] == 1]["sales"], mu_T, sig
    return np.sum(np.log(likelihood_control)) + np.sum(np.log(likelihood_treatme

# EM Algorithm (simplified due to observed group labels)
max_iter = 100  # Maximum number of iterations
tolerance = 1e-6  # Convergence criterion

log_likelihoods = []

for iteration in range(max_iter):
    # E-Step: No need for responsibilities as group labels are observed

    # M-Step: Update parameters
    control_data = data[data["group"] == 0]["sales"]
    treatment_data = data[data["group"] == 1]["sales"]

    # Update means
```

```
        mu_C = np.mean(control_data)
        mu_T = np.mean(treatment_data)

        # Update standard deviations
        sigma_C = np.std(control_data, ddof=1)
        sigma_T = np.std(treatment_data, ddof=1)

        # Compute log-likelihood and check for convergence
        current_log_likelihood = log_likelihood(data, mu_C, sigma_C, mu_T, sigma_T)
        log_likelihoods.append(current_log_likelihood)

        if len(log_likelihoods) > 1 and abs(log_likelihoods[-1] - log_likelihoods[-2
            print(f"Converged at iteration {iteration}")
            break

    # Output the results
    print(f"Final parameters:")
    print(f"Control Mean (mu_C): {mu_C:.2f}, Control STD (sigma_C): {sigma_C:.2f}")
    print(f"Treatment Mean (mu_T): {mu_T:.2f}, Treatment STD (sigma_T): {sigma_T:.2f

    # Compare the means for inference
    if mu_T > mu_C:
        print("The Treatment group outperforms the Control group.")
    else:
        print("The Control group outperforms the Treatment group.")
```

```
Converged at iteration 1

Final parameters:
Control Mean (mu_C): 49.86, Control STD (sigma_C): 9.72
Treatment Mean (mu_T): 56.08, Treatment STD (sigma_T): 12.03

The Treatment group outperforms the Control group.
```

Explanation of Results

- Control Group: Mean = *49.86*, STD = *9.72*
- Treatment Group: Mean = *56.08*, STD = *12.03*

With full group labels, EM directly calculates parameters without the need for latent variables. Results confirm the treatment group outperforms the control group. Group-specific variability is also clear ($\sigma\_T > \sigma\_C$).

Summary of All Methods:

- Masked Data (EM): Handles missing labels and gives detailed group info (e.g., proportions).
- T-Test: Quick and confirms differences but ignores missing labels.
- Observed Data (EM): Straightforward and precise, leveraging all data for richer insights.

This setup shows EM's flexibility and power, especially with complete group labels.

Understanding and Applications of Bayesian Estimators in A/B Testing

Let's consider a simple scenario with sales data from an A/B test:

$x\_C = \{10,12,11,9,13\}$: Sales data from the control group (5 customers).

$x\_T = \{15,17,16,14,18\}$: Sales data from the treatment group (5 customers).

We will use the Bayesian A/B testing to determine if the treatment group outperforms the control group. Specifically, we need to estimate the posterior distributions of the means (μ_C, μ_T) and evaluate the probability that the treatment group's mean sales are higher than the control group's mean $P(\mu\_T > \mu\_C)$.

1. Define the Likelihood

Assume the sales data for both groups follows a normal distribution:

$$x_C \sim \mathcal{N}(\mu_C, \sigma_C^2)$$
$$x_T \sim \mathcal{N}(\mu_T, \sigma_T^2)$$

Here:

μ_C: Mean sales of the control group.

μ_T: Mean sales of the treatment group.

σ_C, σ_T: STD for the control and treatment groups.

2. Define Priors

We place normal priors on the group means and an inverse-gamma prior on the variances:

$$\mu_C, \mu_T \sim \mathcal{N}(\mu_0, \tau^2)$$

$$\sigma_C^2, \sigma_T^2 \sim \text{Inverse-Gamma}(\alpha, \beta)$$

For simplicity, let's assume:

$\mu\_0 = 0$, $\tau\_2 = 10$ (weak prior on group means)

$\alpha = 2$, $\beta = 2$ (weak prior on variances)

3. Bayesian Posterior Estimation

Using Bayes' theorem, we compute the posterior distribution for each parameter:

$$p(\mu_C, \mu_T, \sigma_C^2, \sigma_T^2 | x_C, x_T) \propto p(x_C|\mu_C, \sigma_C^2)p(x_T|\mu_T, \sigma_T^2)p(\mu_C)p(\mu_T)p(\sigma_C^2)p(\sigma_T^2)$$

This posterior combines the likelihood:

$$p(x_C|\mu_C, \sigma_C^2) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma_C^2}} \exp\left(-\frac{(x_{C,i} - \mu_C)^2}{2\sigma_C^2}\right)$$

, $p(x\_T \mid \mu\_T, \sigma\_T)$ (similar to above equation), and those priors such as $p(\mu\_C)$, $p(\mu\_T)$....

4. Sampling the Posterior

To estimate the posterior, we use Markov Chain Monte Carlo (MCMC) or another Bayesian sampling technique. This gives us samples of $\mu\_C, \mu\_T, \sigma\_C, \sigma\_T$ from the posterior distribution.

4.1 Analytical Posterior for the Means

Assume known variances $\sigma\_C = \sigma\_T = 1$. The posterior for μ_C and μ_T simplifies to:

$$\mu_C|x_C \sim \mathcal{N}(\hat{\mu}_C, \frac{\sigma_C^2}{n_C})$$

$$\hat{\mu}_C = \frac{\sum_{i=1}^{n_C} x_{C,i}}{n_C}, \quad n_C = 5$$

$$\mu_T|x_T \sim \mathcal{N}(\hat{\mu}_T, \frac{\sigma_T^2}{n_T})$$

$$\hat{\mu}_T = \frac{\sum_{i=1}^{n_T} x_{T,i}}{n_T}, \quad n_T = 5$$

For the example data, the control group mean:

$$\hat{\mu}_C = \frac{10+12+11+9+13}{5} = 11$$

The treatment group mean:

$$\hat{\mu}_T = \frac{15+17+16+14+18}{5} = 16$$

So, the simplified posterior distributions is:

$$\mu_C|x_C \sim \mathcal{N}(11, 0.2), \quad \mu_T|x_T \sim \mathcal{N}(16, 0.2)$$

4.2 Compute Posterior Probabilities

The Bayesian test computes the probability that $\mu\_T > \mu\_C$:

$$P(\mu_T > \mu_C) = \int_{-\infty}^{\infty} \int_{\mu_C}^{\infty} p(\mu_C|x_C)p(\mu_T|x_T)\, d\mu_T\, d\mu_C$$

This can be estimated using Monte Carlo sampling:

- Draw μ_C and μ_T from their respective posteriors.
- Compute the proportion of samples where $\mu\_T > \mu\_C$.

For our example, let's assume sampling gives:

$$P(\mu_T > \mu_C) \approx 0.999$$

This suggests strong evidence that the treatment group outperforms the control group.

Summary of Bayesian Estimators

> *Bayesian estimation finds the parameter θ of a distribution p(X | θ) by combining what you already know (prior) with new data (likelihood) to update your beliefs (posterior). It doesn't just give one estimate — it shows a full range of possibilities and how confident you can be:*

$$p(\theta, Z \mid X) \propto p(X \mid \theta, Z)p(\theta)p(Z)$$

Here's how it works:

Start with a Prior: Your initial guess about θ, like assuming it follows a normal distribution.

Add Data (Likelihood): Use the data X to calculate how likely it is for different values of θ.

Update with Bayes' Rule: Combine the prior and likelihood to get the posterior $p(\theta \mid X)$, which tells you the most likely values of θ and the uncertainty around them:

$$p(\boldsymbol{\theta} \mid X) = \frac{p(X \mid \boldsymbol{\theta})p(\boldsymbol{\theta})}{p(X)}$$

Use the Posterior: From the posterior, you can make estimates (e.g., the mean or mode of θ or calculate probabilities, like the chance one group outperforms another in A/B testing.

Bayesian estimation is great for handling uncertainty and combining prior knowledge with new data. It's especially useful when data is limited or noisy because it gives you a clear picture of both likely values and how confident you can be.

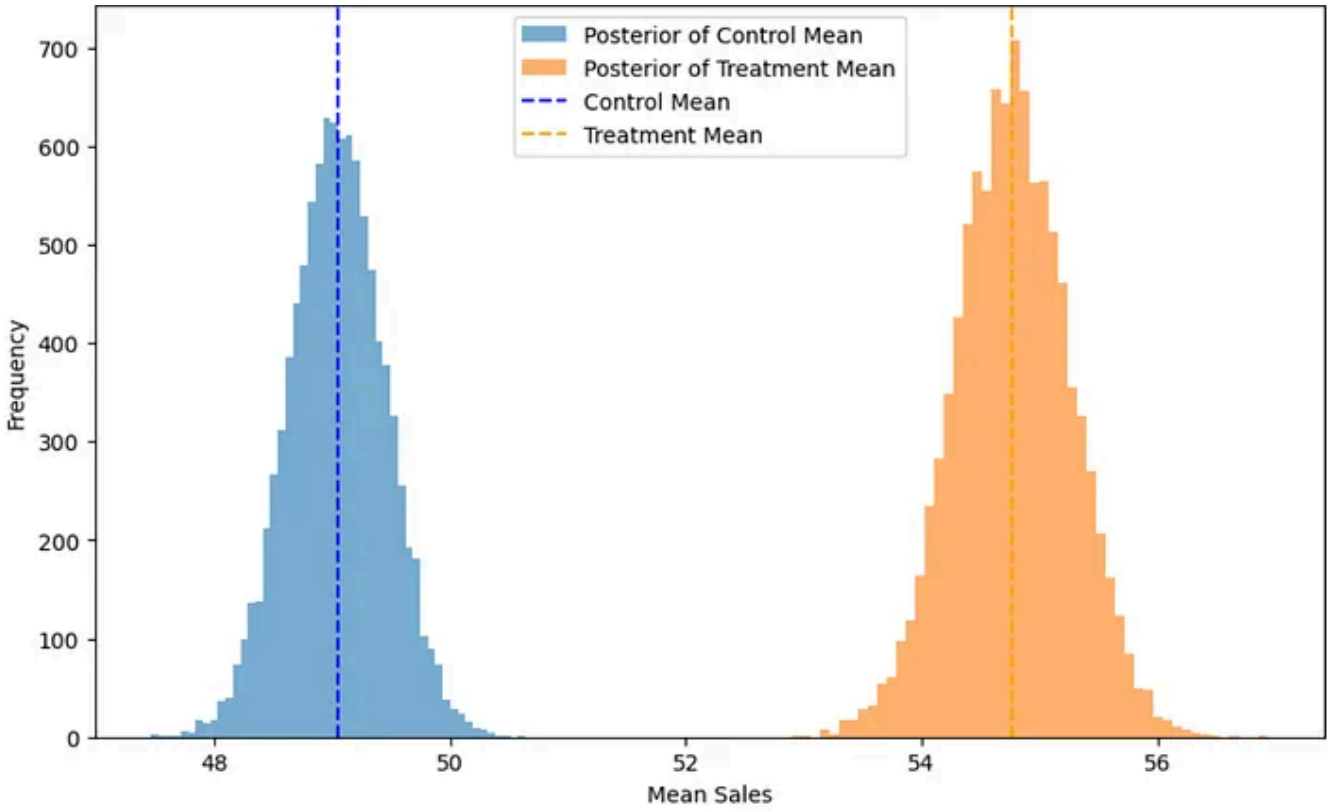
When it's too hard to calculate the posterior directly, methods like Markov Chain Monte Carlo (MCMC) or Variational Inference can be used to approximate it.

Code Experiment for Bayesian A/B Testing

```
import numpy as np
import pandas as pd
from scipy.stats import invgamma, norm
import matplotlib.pyplot as plt


# Define Priors
mu_0 = 0  # Prior mean for group means
tau_2 = 10  # Prior variance for group means
alpha = 2  # Shape parameter for inverse-gamma prior
beta = 2  # Scale parameter for inverse-gamma prior

# Bayesian Posterior Estimation
# Known variance for simplicity
sigma_C = 10
sigma_T = 12

# Compute posterior parameters for Control group
control_data = data[data["group"] == 0]["sales"]
n_C = len(control_data)
x_bar_C = np.mean(control_data)

post_mu_C = (mu_0 / tau_2 + n_C * x_bar_C / sigma_C**2) / (1 / tau_2 + n_C / sig
post_var_C = 1 / (1 / tau_2 + n_C / sigma_C**2)

# Compute posterior parameters for Treatment group
treatment_data = data[data["group"] == 1]["sales"]
n_T = len(treatment_data)
x_bar_T = np.mean(treatment_data)

post_mu_T = (mu_0 / tau_2 + n_T * x_bar_T / sigma_T**2) / (1 / tau_2 + n_T / sig
post_var_T = 1 / (1 / tau_2 + n_T / sigma_T**2)

# Monte Carlo Sampling
num_samples = 10000
samples_mu_C = np.random.normal(post_mu_C, np.sqrt(post_var_C), num_samples)
samples_mu_T = np.random.normal(post_mu_T, np.sqrt(post_var_T), num_samples)
```

```
# Compute probability that Treatment mean > Control mean
prob_treatment_better = np.mean(samples_mu_T > samples_mu_C)

# Step 5: Results
print(f"Posterior mean for Control group: {post_mu_C:.2f}, Variance: {post_var_C
print(f"Posterior mean for Treatment group: {post_mu_T:.2f}, Variance: {post_var
print(f"Probability that Treatment mean > Control mean: {prob_treatment_better:.

# Step 6: Visualization
plt.figure(figsize=(10, 6))
plt.hist(samples_mu_C, bins=50, alpha=0.6, label='Posterior of Control Mean')
plt.hist(samples_mu_T, bins=50, alpha=0.6, label='Posterior of Treatment Mean')
plt.axvline(x=post_mu_C, color='blue', linestyle='--', label='Control Mean')
plt.axvline(x=post_mu_T, color='orange', linestyle='--', label='Treatment Mean')
plt.title('Posterior Distributions of Group Means')
plt.xlabel('Mean Sales')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

```
Posterior mean for Control group: 49.05, Variance: 0.16
Posterior mean for Treatment group: 54.77, Variance: 0.23
Probability that Treatment mean > Control mean: 1.0000
```



Analysis of Results

- Bayesian analysis confirms the treatment group has a significantly higher mean, with a *100%* probability that the treatment outperforms the control group.
- The posterior variances are small (Control: *0.16*, Treatment: *0.23*), showing high confidence in the estimated means.
- Bayesian Estimator stands out by quantifying how likely the treatment is better and providing tighter estimates with clear posterior distributions. It's the most informative here.

Variational Auto-Encoder (VAE) by Illustrative Example

Let's dive into the Variational Autoencoder (VAE) with a story and a bit of math to make it fun and relatable. Imagine you're a teacher trying to

understand your students (dataset X), but you suspect there's something hidden about them — some latent traits — that explain their growth patterns. Our job is to find these hidden traits using a VAE.

We have three students and two features: age and height. The data looks like this:

$$X = \begin{bmatrix} 15 & 160 \\ 16 & 170 \\ 17 & 165 \end{bmatrix}$$

Here, rows represent students, and columns represent features (age and height).

You suspect that the students' growth patterns depend on a hidden factor (e.g., "growth stage" or "nutritional status"). The VAE's goal is to figure out this hidden trait (latent variable Z), and at the same time, ensure that we can reconstruct X accurately from Z.

What does a VAE Do in this job?

Encoder Neural Network

The encoder compresses X into a simpler, lower-dimensional representation:

- Think of Z as a "growth code" for each student.

- The encoder doesn't give you Z directly. Instead, it tells you the mean (μ) and variance (σ^2) of a distribution where Z might be found.

For example:

Student 1 might have $\mu = 0.5$ (they're in mid-growth) and $\sigma^2 = 0.1$ (we're confident about this guess).

Student 2 might have $\mu = -0.3$ (they're in late growth) and $\sigma^2 = 0.2$ (less confident).

Decoder Neural Network

The decoder works backward. It takes a sampled Z (from the encoder's distribution) and reconstructs X (e.g., age and height). This ensures that Z actually captures meaningful information about the original data.

Loss Function

The VAE minimizes two kinds of mistakes:

1. Reconstruction Loss: Ensures that X's prediction (the output of the decoder) looks like the original X (students' data).

$$\mathcal{L}_{\text{recon}} = \frac{1}{2}\|X - \hat{X}\|^2$$

1. KL Divergence Loss: Regularizes Z to stay near $N(0,1)$.

$$\mathcal{L}_{\text{KL}} = -\frac{1}{2}\sum(1 + \log\sigma^2 - \mu^2 - \sigma^2)$$

Encoder: Finding the Hidden Factor

Let's check how encoder is working using Neural Net mechanism.

Step 1: Data Transformation

The encoder is like a neural network that first transforms X into an intermediate representation H. The math is:

$$H = \text{ReLU}(X \cdot W_1 + b_1)$$

Example,

$$W_1 = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix}, \quad b_1 = \begin{bmatrix} 0.5 & 0.6 \end{bmatrix}$$

Input Calculation:

$$X \cdot W_1 = \begin{bmatrix} 15 & 160 \\ 16 & 170 \\ 17 & 165 \end{bmatrix} \cdot \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix} = \begin{bmatrix} 48.5 & 64.6 \\ 51.1 & 68.8 \\ 50.0 & 66.0 \end{bmatrix}$$

Adding bias *b_1*:

$$H = \text{ReLU}\left(\begin{bmatrix}48.5 & 64.6\\ 51.1 & 68.8\\ 50.0 & 66.0\end{bmatrix} + \begin{bmatrix}0.5 & 0.6\end{bmatrix}\right) = \begin{bmatrix}49.0 & 65.2\\ 51.6 & 69.4\\ 50.5 & 66.6\end{bmatrix}$$

Step 2: Outputting Latent Parameters

The encoder now computes μ (mean) and logσ^2 variance) using additional weights and biases:

$$\mu = H \cdot W_\mu + b_\mu, \quad \log \sigma^2 = H \cdot W_{\log \sigma^2} + b_{\log \sigma^2}$$

Example Values:

$$W_\mu = \begin{bmatrix}0.1\\ 0.2\end{bmatrix}, \quad b_\mu = 0.1, \quad W_{\log \sigma^2} = \begin{bmatrix}0.3\\ 0.4\end{bmatrix}, \quad b_{\log \sigma^2} = 0.2$$

Therefore,

$$\mu = \begin{bmatrix} 49.0 & 65.2 \\ 51.6 & 69.4 \\ 50.5 & 66.6 \end{bmatrix} \cdot \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} + 0.1 = \begin{bmatrix} 17.04 \\ 18.06 \\ 17.43 \end{bmatrix}$$

$$\log \sigma^2 = \begin{bmatrix} 49.0 & 65.2 \\ 51.6 & 69.4 \\ 50.5 & 66.6 \end{bmatrix} \cdot \begin{bmatrix} 0.3 \\ 0.4 \end{bmatrix} + 0.2 = \begin{bmatrix} 31.78 \\ 34.14 \\ 32.53 \end{bmatrix}$$

Resampling and Decoding

Instead of picking a single Z, we draw samples to make Z flexible:

$$Z = \mu + \sigma \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, 1)$$

For example, for μ = 17.04, $\epsilon = -0.1$

$$\sigma = e^{31.78/2} \approx 10^{6.57}$$

$$Z = 17.04 + \sigma \cdot (-0.1)$$

Decoder

The decoder takes Z and reconstructs

$$\hat{X} = \text{DecoderNN}(Z)$$

Summary of Variational Autoencoders (VAEs)

> *Variational Autoencoders (VAEs) are machine learning models that uncover hidden patterns (latent variables Z) in data (X) by compressing it into a smaller representation (latent space) and then reconstructing the original data. They achieve this by combining neural networks with probability distributions, learning key features while enabling the generation of new, similar data.*

1. Encoder (Inference Step): The encoder compresses data X into a simpler form, estimating statistics such as μ and σ^2 for the hidden variables Z.
2. Decoder: It models the posterior distribution $q(Z|X) \sim N(\mu,\sigma^2)$, which captures the uncertainty about Z.
3. The VAE minimizes two losses:

- Reconstruction Loss: Ensures X's prediction is close to X, derived from the likelihood $p(X|Z)$.
- KL Divergence: Keeps $q(Z|X)$ close to a simple prior $p(Z)$ (such as $N(0,1)$).

4. Reparameterization: Ensures smooth training by sampling Z as $\mu + \sigma \cdot \epsilon$, with $\epsilon \sim N(0,1)$.

VAEs are useful for generating new data, compressing data, and understanding hidden structures, combining neural networks and probability.

Variational Auto-Encoder (VAE) for A/B Testing

Variational Autoencoders (VAEs) bring a unique approach to A/B testing by turning your observed data into latent variables and then using a generative decoder to model and resample that data.

Let's break this down with two strategies inspired by the EM (Expectation-Maximization) method, explore how the decoding (resampling) stage works, and check out and highlight the benefits of using VAE for A/B testing.

VAE A/B Testing with Masked Campaign Treatment

The Campaign Treatment information is masked and treated as a latent variable. The sales data X is observed, and the encoder learns a latent representation of both sales and the missing treatment information.

Setup:

Observed Data *(X):* Sales data.

Latent Variables (Z): Representation of the campaign treatment and sales patterns.

Encoder: Learns the approximate posterior $q(z \mid x)$, where z encodes campaign treatment and sales patterns.

Decoder: Reconstructs the sales data x and resamples new data from z.

Steps for VAE in A/B Testing

1. Encoder Stage (Encoding and Posterior Inference):

Input: Sales data $X = \{x\_1, x\_2,\ldots,x\_n\}$.

The encoder outputs:

$$q(z|x) = \mathcal{N}(\mu_z(x), \sigma_z^2(x))$$

Where, the latent variable z captures a probabilistic representation of both the sales data and unobserved campaign treatment effects.

2. Loss Function (ELBO):

The VAE optimizes the Evidence Lower Bound (ELBO):

$$\text{ELBO} = \mathbb{E}_{q(z|x)}[\log p(x|z)] - \text{KL}(q(z|x)\|p(z))$$

- Reconstruction Loss: Measures how well the decoder reconstructs the observed sales data.
- KL Divergence: Regularizes the posterior $q(z|x)$ to stay close to the prior $p(z)$.

3. Decoder Stage (Reconstruction and Resampling):

- After training, the decoder reconstructs the sales data x from the latent variable z, allowing resampling for further analysis.

4. Inference for A/B Testing:

- Compare the latent distributions of z for Control and Treatment to evaluate the campaign effect.

- Use the decoder to generate new samples, simulate outcomes, and assess group differences.

Code Experiment: VAE for A/B Testing with Masked Treatment

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

data = data_abtest.copy()
data["group"] = np.nan  # Mask the group labels

# Normalize sales data
sales_data = (data["sales"] - np.mean(data["sales"])) / np.std(data["sales"])
sales_data = sales_data.values.astype(np.float32).reshape(-1, 1)

# Define VAE Components
latent_dim = 2
input_dim = 1
hidden_dim = 64

class VAE(nn.Module):
    def __init__(self, input_dim, hidden_dim, latent_dim):
        super(VAE, self).__init__()
        # Encoder
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU()
        )
        self.z_mean = nn.Linear(hidden_dim // 2, latent_dim)
        self.z_log_var = nn.Linear(hidden_dim // 2, latent_dim)

        # Decoder
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim)
```

```
            )

        def reparameterize(self, z_mean, z_log_var):
            std = torch.exp(0.5 * z_log_var)
            eps = torch.randn_like(std)
            return z_mean + eps * std

        def forward(self, x):
            h = self.encoder(x)
            z_mean = self.z_mean(h)
            z_log_var = self.z_log_var(h)
            z = self.reparameterize(z_mean, z_log_var)
            x_reconstructed = self.decoder(z)
            return x_reconstructed, z_mean, z_log_var

    # Initialize the VAE model
    vae = VAE(input_dim=input_dim, hidden_dim=hidden_dim, latent_dim=latent_dim)
    optimizer = optim.Adam(vae.parameters(), lr=1e-3)

    # Loss Function
    def loss_function(x, x_reconstructed, z_mean, z_log_var):
        reconstruction_loss = nn.MSELoss()(x_reconstructed, x)
        kl_loss = -0.5 * torch.sum(1 + z_log_var - z_mean.pow(2) - z_log_var.exp())
        return reconstruction_loss + kl_loss

    # Train VAE Model
    sales_tensor = torch.tensor(sales_data)
    epochs = 50
    batch_size = 32
    vae.train()

    for epoch in range(epochs):
        permutation = torch.randperm(sales_tensor.size(0))
        epoch_loss = 0
        for i in range(0, sales_tensor.size(0), batch_size):
            optimizer.zero_grad()
            indices = permutation[i:i + batch_size]
            batch = sales_tensor[indices]
            x_reconstructed, z_mean, z_log_var = vae(batch)
            loss = loss_function(batch, x_reconstructed, z_mean, z_log_var)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()
        print(f"Epoch {epoch + 1}, Loss: {epoch_loss / (sales_tensor.size(0) // batc

    # Step 4: Inference and Analysis
    vae.eval()
    with torch.no_grad():
        x_reconstructed, z_mean, z_log_var = vae(sales_tensor)
        z_mean = z_mean.numpy()
```

```
control_latents = z_mean[:control_size]
treatment_latents = z_mean[control_size:]

# Visualize latent space
plt.figure(figsize=(8, 6))
plt.scatter(control_latents[:, 0], control_latents[:, 1], alpha=0.6, label="Cont
plt.scatter(treatment_latents[:, 0], treatment_latents[:, 1], alpha=0.6, label="
plt.xlabel("Latent Dimension 1")
plt.ylabel("Latent Dimension 2")
plt.title("Latent Space Representation")
plt.legend()
plt.show()

# Generate new samples for A/B testing analysis
latent_samples = torch.randn(1000, latent_dim)
with torch.no_grad():
    generated_samples = vae.decoder(latent_samples).numpy()

plt.hist(generated_samples, bins=50, alpha=0.7, label="Generated Sales")
plt.xlabel("Sales")
plt.ylabel("Frequency")
plt.title("Generated Sales Data from Latent Space")
plt.legend()
plt.show()

# Additional Analysis
# Compute latent means and variances for control and treatment groups
control_mean = np.mean(control_latents, axis=0)
control_var = np.var(control_latents, axis=0)
treatment_mean = np.mean(treatment_latents, axis=0)
treatment_var = np.var(treatment_latents, axis=0)

print("Latent Space Statistics:")
print(f"Control Group Mean: {control_mean}, Variance: {control_var}")
print(f"Treatment Group Mean: {treatment_mean}, Variance: {treatment_var}")

# Compute the difference in latent means
diff_mean = treatment_mean - control_mean
print(f"Difference in Latent Means (Treatment - Control): {diff_mean}")

# Compute the Euclidean distance between latent group centers
euclidean_distance = np.linalg.norm(treatment_mean - control_mean)
print(f"Euclidean Distance Between Latent Group Centers: {euclidean_distance:.4f
```

```
Latent Space Statistics:
Control Group Mean: [-0.00166821 -0.17356496], Variance: [0.00108242 0.36401635]
Treatment Group Mean: [0.01375005 0.2213269 ], Variance: [0.0010667  0.57439774]

Difference in Latent Means (Treatment - Control): [0.01541825 0.39489186]
Euclidean Distance Between Latent Group Centers: 0.3952
```

Takeaways:

Distinct Treatment Effect: The treatment group shows a clear shift in the second latent dimension (*+0.3949*), indicating a significant treatment impact.

Higher Variability in Treatment: Treatment group variance in the second dimension (*0.5744*) is higher than the control group (*0.3640*), suggesting varied responses to the campaign.

Latent Space Separation: The Euclidean distance between group centers (*0.3952*) shows the separation between control and treatment groups in the latent space, even with masked labels.

In short, using a Variational Autoencoder (VAE) for A/B testing with masked treatment data is effective in capturing group differences, such as means and variances, even without complete labels. It can generate a latent space for visualizing treatment effects and handles uncertainty.

VAE A/B Testing with Campaign Treatment as Observed

In this approach, both the campaign treatment information and sales data are treated as observed variables. The encoder will learn a latent representation that combines group membership with sales patterns.

Setup:

Observed Data ($Z=(x, g)$): x is *the s*ales data, and g is the campaign treatment group (Control or Treatment).

Latent Variables (z): Representation of combined campaign treatment and sales effects.

Steps for VAE in A/B Testing:

1. Encoder Stage:

Input sales and group information:

$$Z = \{(x_i, g_i)\}_{i=1}^{n}$$

The encoder outputs:

$$q(z|x, g) = \mathcal{N}(\mu_z(x, g), \sigma_z^2(x, g))$$

Where, the latent variable z encodes the joint representation of campaign treatment and sales.

2. Loss Function (ELBO):

Similar to the masked case, the loss function is:

$$\text{ELBO} = \mathbb{E}_{q(z|x,g)}[\log p(x|z)] - \text{KL}(q(z|x,g)\|p(z))$$

3. Decoder Stage (Reconstruction and Resampling):

The decoder reconstructs the sales data from z:

$$x' \sim p(x|z)$$

Resampling: Generate synthetic data x' conditioned on the latent representation z, enabling simulations and comparisons across groups.

4. Inference for A/B Testing:

- Latent Encodings Comparison: Compare the latent encodings (z) for the Control and Treatment groups to identify differences in their underlying patterns.
- Simulated Outcomes: Use the decoder to resample sales outcomes (x') for hypothetical scenarios, such as testing the impact of applying the treatment to all customers.
- Validation of Group Effects: Leverage resampled outcomes to validate and quantify the impact of campaign strategies on sales performance (details below).

Using Decoding and Resampling in A/B Testing

Generative Analysis: The decoder generates synthetic sales data based on the latent variable z. For instance, it can simulate sales for a scenario where every customer receives the treatment.

Outcome Simulation: Resampling x' enables the evaluation of different campaign strategies and their potential outcomes.

Group Differences Evaluation: Analyze the distributions of $p(z \mid x)$ for control and treatment groups to infer the campaign's effectiveness and its overall impact.

Code Experiment: VAE for A/B Testing with Observed Treatment

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

# use the same data
data = data_abtest.copy()

# Normalize sales data
sales_data = (data["sales"] - np.mean(data["sales"])) / np.std(data["sales"])
sales_data = sales_data.values.astype(np.float32).reshape(-1, 1)

group_data = data["group"].values.astype(np.float32).reshape(-1, 1)
input_data = np.hstack((sales_data, group_data))

# Define VAE Components
latent_dim = 2
input_dim = 2
hidden_dim = 64

class VAE(nn.Module):
    def __init__(self, input_dim, hidden_dim, latent_dim):
        super(VAE, self).__init__()
        # Encoder
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU()
        )
        self.z_mean = nn.Linear(hidden_dim // 2, latent_dim)
```

```
            self.z_log_var = nn.Linear(hidden_dim // 2, latent_dim)

            # Decoder
            self.decoder = nn.Sequential(
                nn.Linear(latent_dim, hidden_dim // 2),
                nn.ReLU(),
                nn.Linear(hidden_dim // 2, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, input_dim)
            )

        def reparameterize(self, z_mean, z_log_var):
            std = torch.exp(0.5 * z_log_var)
            eps = torch.randn_like(std)
            return z_mean + eps * std

        def forward(self, x):
            h = self.encoder(x)
            z_mean = self.z_mean(h)
            z_log_var = self.z_log_var(h)
            z = self.reparameterize(z_mean, z_log_var)
            x_reconstructed = self.decoder(z)
            return x_reconstructed, z_mean, z_log_var

    # Initialize the VAE model
    vae = VAE(input_dim=input_dim, hidden_dim=hidden_dim, latent_dim=latent_dim)
    optimizer = optim.Adam(vae.parameters(), lr=1e-3)

    # Loss Function
    def loss_function(x, x_reconstructed, z_mean, z_log_var):
        reconstruction_loss = nn.MSELoss()(x_reconstructed, x)
        kl_loss = -0.5 * torch.sum(1 + z_log_var - z_mean.pow(2) - z_log_var.exp())
        return reconstruction_loss + kl_loss

    # Train VAE Model
    input_tensor = torch.tensor(input_data)
    epochs = 50
    batch_size = 32
    vae.train()

    for epoch in range(epochs):
        permutation = torch.randperm(input_tensor.size(0))
        epoch_loss = 0
        for i in range(0, input_tensor.size(0), batch_size):
            optimizer.zero_grad()
            indices = permutation[i:i + batch_size]
            batch = input_tensor[indices]
            x_reconstructed, z_mean, z_log_var = vae(batch)
            loss = loss_function(batch, x_reconstructed, z_mean, z_log_var)
            loss.backward()
```

```
            optimizer.step()
            epoch_loss += loss.item()
        print(f"Epoch {epoch + 1}, Loss: {epoch_loss / (input_tensor.size(0) // batc

    # Inference and Analysis
    vae.eval()
    with torch.no_grad():
        x_reconstructed, z_mean, z_log_var = vae(input_tensor)
        z_mean = z_mean.numpy()

    control_latents = z_mean[:control_size]
    treatment_latents = z_mean[control_size:]

    # Visualize latent space
    plt.figure(figsize=(8, 6))
    plt.scatter(control_latents[:, 0], control_latents[:, 1], alpha=0.6, label="Cont
    plt.scatter(treatment_latents[:, 0], treatment_latents[:, 1], alpha=0.6, label="
    plt.xlabel("Latent Dimension 1")
    plt.ylabel("Latent Dimension 2")
    plt.title("Latent Space Representation")
    plt.legend()
    plt.show()

    # Generate new samples for A/B testing analysis
    latent_samples = torch.randn(1000, latent_dim)
    with torch.no_grad():
        generated_samples = vae.decoder(latent_samples).numpy()

    plt.hist(generated_samples[:, 0], bins=50, alpha=0.7, label="Generated Sales")
    plt.xlabel("Sales")
    plt.ylabel("Frequency")
    plt.title("Generated Sales Data from Latent Space")
    plt.legend()
    plt.show()

    # Additional Analysis
    # Compute latent means and variances for control and treatment groups
    control_mean = np.mean(control_latents, axis=0)
    control_var = np.var(control_latents, axis=0)
    treatment_mean = np.mean(treatment_latents, axis=0)
    treatment_var = np.var(treatment_latents, axis=0)

    print("Latent Space Statistics:")
    print(f"Control Group Mean: {control_mean}, Variance: {control_var}")
    print(f"Treatment Group Mean: {treatment_mean}, Variance: {treatment_var}")

    # Compute the difference in latent means
    diff_mean = treatment_mean - control_mean
    print(f"Difference in Latent Means (Treatment - Control): {diff_mean}")
```

```
# Compute the Euclidean distance between latent group centers
euclidean_distance = np.linalg.norm(treatment_mean - control_mean)
print(f"Euclidean Distance Between Latent Group Centers: {euclidean_distance:.4f
```

```
Latent Space Statistics:
Control Group Mean: [-0.01510795 -0.01647374], Variance: [0.00782431 0.00122233]
Treatment Group Mean: [0.053562   0.01212562], Variance: [0.01066029 0.0015969 ]

Difference in Latent Means (Treatment - Control): [0.06866995 0.02859936]

Euclidean Distance Between Latent Group Centers: 0.0744
```

Explanation:

- Latent Mean Shift: Observed data shows a clear shift in the first latent dimension (*+0.0687*), while masked data shows a bigger shift in the second dimension (*+0.3949*). Observed labels provide sharper resolution in some areas but might miss broader patterns seen with masked data.
- Variance Differences: Observed data has smaller variances (Control: [*0.0078, 0.0012*], Treatment: [*0.0107, 0.0016*]), creating tighter latent representations. Masked data has larger variances, especially in the second dimension. This captures more uncertainty.
- Group Separation: Observed data gives a smaller Euclidean distance (*0.0744*), while masked data has a larger separation (*0.3952*). Masked data uncovers deeper group differences.

Two VAE A/B Testing Approaches:

- Observed Treatment Data: Gives precise and compact representations, making it great for clear and straightforward group comparisons.
- Masked Treatment Data: Captures broader group differences and variability, ideal for uncovering hidden patterns in incomplete data.

Our choice depends on what we prioritize — accuracy with observed data or exploring deeper patterns with masked data.

Why Use VAEs for A/B Testing?

VAEs, a newer method based on neural networks and deep learning, are widely used in AI for image processing but rarely applied to tabular data. Here's why they work well in A/B testing based on the data and experiments:

Handles Missing Data: VAEs can infer missing treatment info during encoding, making them reliable when data isn't complete.

Simulations with Decoder: VAEs allow outcome simulations for deeper group analysis, unlike traditional methods like EM or T-tests.

Uncertainty Quantification: They provide posterior distributions for latent variables, showing not just estimates but the uncertainty in group effects.

Outlier Handling: VAEs are robust against outliers due to their probabilistic approach.

Flexible Assumptions: They don't rely on strict Gaussian assumptions and can model complex, high-dimensional data effectively.

Final Thoughts

I examine the mechanics and mathematics of EM, Bayesian estimation, and VAEs through examples, demonstrating how each method works. From EM's iterative guessing to Bayesian estimation's probabilistic insights and VAE's latent space modeling, their core concepts have been explored. These methods were tested within A/B testing frameworks to compare their procedures and highlight their strengths.

Each method uncovers hidden variables in its own way. EM iteratively guesses and refines latent factors, making it effective for handling incomplete data. Bayesian estimation integrates prior knowledge to manage uncertainty and provide confident results. VAEs take a step further, utilizing neural networks to uncover hidden structures and generate new possibilities, making them particularly useful for simulations and exploratory analysis.

The study of latent factors has huge potential for future research. Combining EM's simplicity, Bayesian estimation's way of handling uncertainty, and VAE's generative power could lead to exciting new possibilities. Adding advanced neural network methods, like attention mechanisms or deep learning, to VAEs could make them even better at finding hidden patterns in time series data, resampling tabular data, and detecting subtle causal relationships. This could improve areas like recommendation systems, marketing strategies, and medical data analysis. Exciting opportunities lie ahead!

The code and dataset used in this study are publicly available at https://github.com/datalev001/em_bay_vae/.

About me

With over 20 years of experience in software and database management and 25 years teaching IT, math, and statistics, I am a Data Scientist with extensive expertise across multiple industries.

You can connect with me at:

Email: datalev@gmail.com | LinkedIn | X/Twitter

Bayesian Machine Learning | Em Algorithm | Vae | AI | Ab Testing

Responses (1)

Alex Mylnikov

What are your thoughts?

Momtchil Botev
Dec 15, 2024

Thank you for this article!

Reply

More from Shenggang Li and Towards AI

In Towards AI by Shenggang Li

Practical Guide to Distilling Large Models into Small Models: A Nove...

Comparing Traditional and Enhanced Step-by-Step Distillation: Adaptive Learning,...

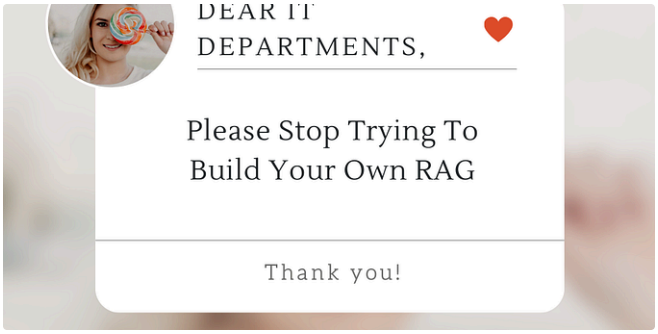
Mar 3 120 2

In Towards AI by Alex Punnen

Explaining Transformers as Simple as Possible through a Small...

And understanding Vector Transformations and Vectorizations

Feb 14 738 15



In Towards AI by Alden Do Rosario

Dear IT Departments, Please Stop Trying To Build Your Own RAG

IT departments convince themselves that building their own RAG-based chat is easy. It'...

Nov 11, 2024 6.2K 176

In Towards AI by Shenggang Li

Beyond Buy-and-Hold: Dynamic Strategies for Unlocking Long-...

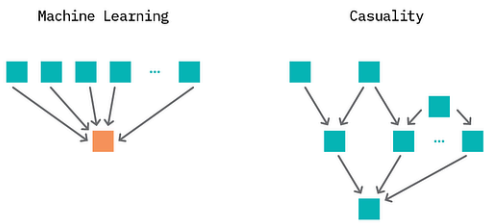
Harnessing Survival Analysis and Markov Decision Processes to Surpass Static ETF...

Feb 7 324 7

See all from Shenggang Li

See all from Towards AI

Recommended from Medium



Karan_bhutani

The Causal Revolution in Machine Learning: Moving Beyond...

Causal Machine Learning (Causal ML) represents a fundamental shift in how we...

Mar 3 10 4

Kishan A

Understanding the Hierarchical Bayesian Model for Price Elasticity

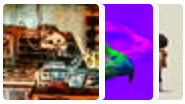
In the dynamic world of pricing and marketing, understanding how demand...

Oct 28, 2024 24

Lists

Generative AI Recommended Reading

52 stories · 1690 saves



What is ChatGPT?

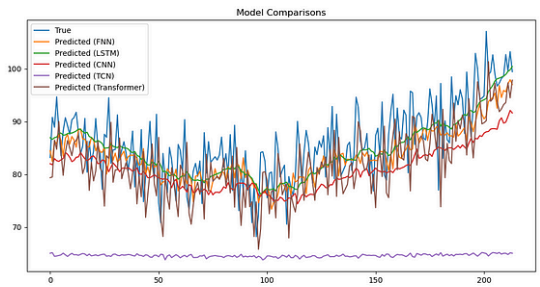
9 stories · 521 saves

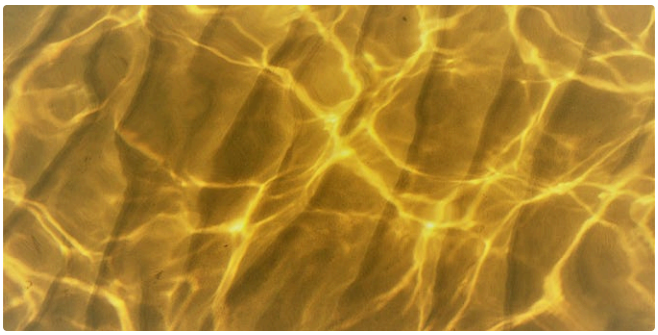
The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 562 saves

Natural Language Processing

1976 stories · 1619 saves





Kyle Jones

Comparing Deep Learning Architectures for Time Series

Deep learning has revolutionized time series analysis with architectures like FNNs, LSTMs...

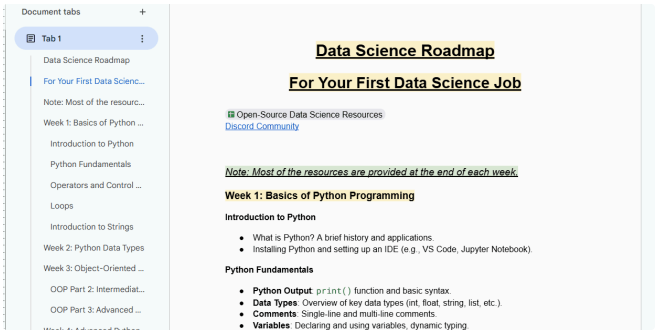
Jan 16 222 2

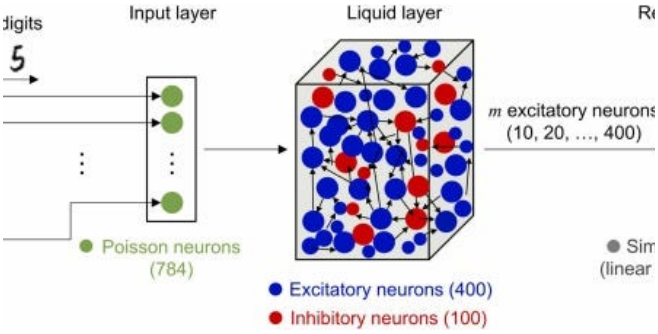
In Towards AI by Shenggang Li

Reimagining Diffusion Models: Autoregressive Priors for Efficien...

Exploring a Novel Approach to Diffusion Initialization with Intuitive Illustrations,...

3d ago 122





In Biased-Algorithms by Amit Yadav

SHAP Values for Binary Classification

I understand that learning data science can be really challenging...

Sep 19, 2024 17

Nivedita Bhadra

Timeseries prediction with Liquid State Machine Model in Python

A Liquid State Machine-a powerful technique for both prediction and classification task

Oct 30, 2024 420 1

See more recommendations