Minor Assignment 2

Vihaan Sapra
231149
svihaan23@iitk.ac.in

Nischay Agarwal
230705
nischaya23@iitk.ac.in

Aryamann Srivastava
230211
aryamanns23@iitk.ac.in

Dharajiya Yug Harshadbhai
230362
dharajiya23@iitk.ac.in

Jain Shlok Nilesh
230493
jainshlok23@iitk.ac.in

Shravan Agrawal
230984
ashravan23@iitk.ac.in

Abstract

This report is submitted as Minor Assignment 2 for the course CS 771, by the Group **MLVortex**.

1 Detailed calculations on training given generative multiclassification model

The goal is to find the model parameters θ that maximize the log likelihood of our observed training data which consists of N samples $\{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$.

The model parameters θ to be learned are:

- Class Priors: $\phi_c = P(y = c)$ for each class $c \in [C]$.
- GMM Parameters (for each class c):
  - Mixture weights: $\pi_c = (\pi_{c1}, \dots, \pi_{cK})$ where $\sum_{k=1}^{K} \pi_{ck} = 1$.
  - Component means: μ_{ck} for each component $k \in [K]$.
  - Component covariances: Σ_{ck} for each component $k \in [K]$.

Representing it in terms of log likelihood:

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \log P(x^{(i)}, y^{(i)}; \theta)$$

Using the rule of probability $P(x, y) = P(x|y)P(y)$, we can split the log likelihood:

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \left(\log P(x^{(i)}|y^{(i)}; \theta) + \log P(y^{(i)}; \theta) \right)$$

Now, we can deal with two different parts independently:

- Training the class priors $P(y)$.
- Training the class conditional GMMs $P(x|y)$.

1.1 Training Priors

We will optimize the second term of the log likelihood with respect to the parameters $\phi_c = P(y = c)$. Let N_c be the number of training samples belonging to class c.

$$\mathcal{L}_{\text{priors}}(\phi) = \sum_{i=1}^{N} \log P(y^{(i)}) = \sum_{c=1}^{C} \sum_{i:y^{(i)}=c} \log P(y = c) = \sum_{c=1}^{C} N_c \log \phi_c$$

The constraint is $\sum_{c=1}^{C} \phi_c = 1$. Using a Lagrange multiplier λ, we form the Lagrangian:

$$\mathcal{J}(\phi, \lambda) = \sum_{c=1}^{C} N_c \log \phi_c + \lambda \left(1 - \sum_{c=1}^{C} \phi_c\right)$$

Taking the partial derivative with respect to a specific ϕ_c and setting it to zero:

$$\frac{\partial \mathcal{J}}{\partial \phi_c} = \frac{N_c}{\phi_c} - \lambda = 0 \implies \phi_c = \frac{N_c}{\lambda}$$

Solving for λ using the constraint:

$$\sum_{c=1}^{C} \phi_c = \sum_{c=1}^{C} \frac{N_c}{\lambda} = 1 \implies \frac{1}{\lambda} \sum_{c=1}^{C} N_c = 1$$

Since $\sum_{c=1}^{C} N_c$ is just the total number of samples N, we have $\frac{N}{\lambda} = 1$, which means $\lambda = N$.

Substituting λ back, we get the closed form solution for the class priors:

$$\phi_c = \frac{N_c}{N}$$

1.2 Training the class conditional GMMs

This part involves optimizing the first term of the log likelihood:

$$\mathcal{L}_{\text{cond}}(\theta_{\text{GMMs}}) = \sum_{i=1}^{N} \log P(x^{(i)} | y^{(i)})$$

Split this sum by each separate class. Let $\mathcal{D}_c = \{x^{(i)} | y^{(i)} = c\}$ be the set of N_c data points belonging to class c.

$$\mathcal{L}_{\text{cond}} = \sum_{c=1}^{C} \sum_{i : y^{(i)} = c} \log P(x^{(i)} | y = c)$$

This shows that the problem now has been brought down into C independent training problems. For each class c, we must fit a K component GMM to its corresponding data $\mathcal{D}_c$.

If we focus on a GMM for a single class c:

$$\mathcal{L}_c = \sum_{i : y^{(i)} = c} \log P(x^{(i)} | y = c) = \sum_{i : y^{(i)} = c} \log \left(\sum_{k=1}^{K} \pi_{ck} \mathcal{N}(x^{(i)} | \mu_{ck}, \Sigma_{ck}) \right)$$

We will use the EM algorithm to optimize this.

The EM algorithm for a single class c:

We introduce a variable $z^{(i)}$ for each data point $x^{(i)} \in \mathcal{D}_c$, where $z^{(i)} = k$ means that $x^{(i)}$ was generated by the k-th Gaussian component.

Initialize $\{\pi_{c,k},\, \mu_{c,k},\, \Sigma_{c,k}\}_{k=1}^{K}$ using k-means, random subset or other similar methods.

E-Step:

We compute the responsibility γ_{ick} that component k takes for data point $x^{(i)}$. This is the posterior probability $P(z^{(i)} = k | x^{(i)}, \theta_c^{\text{old}})$ given the current parameters. For each data point $x^{(i)} \in \mathcal{D}_c$ and each component $k \in [K]$:

$$\gamma_{ick} = \frac{P(x^{(i)} | z^{(i)} = k) P(z^{(i)} = k)}{\sum_{j=1}^{K} P(x^{(i)} | z^{(i)} = j) P(z^{(i)} = j)} = \frac{\pi_{ck}^{\text{old}} \mathcal{N}(x^{(i)} | \mu_{ck}^{\text{old}}, \Sigma_{ck}^{\text{old}})}{\sum_{j=1}^{K} \pi_{cj}^{\text{old}} \mathcal{N}(x^{(i)} | \mu_{cj}^{\text{old}}, \Sigma_{cj}^{\text{old}})}$$

M-Step:

We update the parameters $\{\pi_{ck}, \mu_{ck}, \Sigma_{ck}\}$ to maximize the expected complete-data log likelihood, using the responsibilities γ_{ick} as soft weights.First, calculate the **soft count** of data points assigned to each component k (for class c):

$$N_{ck} = \sum_{i:y^{(i)}=c} \gamma_{ick}$$

This is the effective number of data points in component k. Now update the parameters to get θ_c^{new}:

- **Mixture Weight Update**: The new weight π_{ck} is the proportion of the data **softly** assigned to component k.

$$\pi_{ck}^{\text{new}} = \frac{N_{ck}}{N_c} \quad \left(\text{Note: } N_c = \sum_{k=1}^{K} N_{ck}\right)$$

- **Mean Update**: The new mean μ_{ck} is the weighted average of all data points in $\mathcal{D}_c$, where the weights are the responsibilities γ_{ick}.

$$\mu_{ck}^{\text{new}} = \frac{1}{N_{ck}} \sum_{i:y^{(i)}=c} \gamma_{ick} x^{(i)}$$

- **Covariance Update**: The new covariance Σ_{ck} is the weighted covariance, using the newly computed mean μ_{ck}^{new}.

$$\Sigma_{ck}^{\text{new}} = \frac{1}{N_{ck}} \sum_{i:y^{(i)}=c} \gamma_{ick} (x^{(i)} - \mu_{ck}^{\text{new}})(x^{(i)} - \mu_{ck}^{\text{new}})^T$$

Update $\theta_c^{\text{old}} \leftarrow \theta_c^{\text{new}}$ and go back to the E-Step and repeat until convergence.

2 Detailed Calculations for Inference (Testing)

The goal is to find the class c that is most probable given the observed test point x_{test}. This is known as the **Maximum A Posteriori (MAP)** decision rule.We want to find the class $\hat{y}$ that maximizes the posterior probability $P(y = c|x_{\text{test}})$:

$$\hat{y} = \arg\max_{c\in[C]} P(y = c|x_{\text{test}})$$

Applying Bayes' Rule

We use Bayes' rule to express the posterior probability in terms of the quantities our model has learned (the likelihood and the prior).

$$P(y = c|x_{\text{test}}) = \frac{P(x_{\text{test}}|y = c)P(y = c)}{P(x_{\text{test}})}$$

When finding the $\arg\max$ over c, the denominator $P(x_{\text{test}}) = \sum_{j=1}^{C} P(x_{\text{test}}|y = j)P(y = j)$ is a constant for all classes. Therefore, we can ignore it, simplifying the rule to:

$$\hat{y} = \arg\max_{c\in[C]} P(x_{\text{test}}|y = c)P(y = c)$$

Switching to Log Probability

Multiplying many probabilities (which are often very small numbers) can lead to numerical underflow.It is a standard practice to perform these calculations in log space.Since the logarithm is a monotonically increasing function, maximizing a value is the same as maximizing its logarithm.

$$\hat{y} = \arg\max_{c\in[C]} (\log P(x_{\text{test}}|y = c) + \log P(y = c))$$

This gives a 'score' for each class. Our prediction $\hat{y}$ will be the class with the highest score.

$\text{Score}(c) = \log P(x_{\text{test}}|y = c) + \log P(y = c)$

Inference Calculation

Given a new test point x_{test}, we must compute this score for every class $c = 1, \ldots, C$. All parameters $(\phi_c, \pi_{ck}, \mu_{ck}, \Sigma_{ck})$ are known from the training step.

Step 1: Calculate the Log Prior: $\log P(y = c)$

This is the easiest part. It's the log of the class prior ϕ_c that we calculated during training.

$$\log P(y = c) = \log \phi_c$$

Step 2: Calculate the Class Conditional Log Likelihood: $\log P(x_{\text{test}}|y = c)$

This is the core of the calculation. Recall that the likelihood for class c is a GMM:

$$P(x_{\text{test}}|y = c) = \sum_{k=1}^{K} \pi_{ck} \mathcal{N}(x_{\text{test}}|\mu_{ck}, \Sigma_{ck})$$

Therefore, we need to compute:

$$\log P(x_{\text{test}}|y = c) = \log \left(\sum_{k=1}^{K} \pi_{ck} \mathcal{N}(x_{\text{test}}|\mu_{ck}, \Sigma_{ck}) \right)$$

where

$$\mathcal{N}(x_{\text{test}}|\mu_{ck}, \Sigma_{ck}) = \frac{1}{\sqrt{(2\pi)^d |\Sigma_{ck}|}} \exp \left(-\frac{1}{2} (x_{\text{test}} - \mu_{ck})^T \Sigma_{ck}^{-1} (x_{\text{test}} - \mu_{ck}) \right)$$

After calculating the score for every class $c \in [C]$:

$$\text{Score}(c) = \underbrace{\log \phi_c}_{\text{Step 1}} + \underbrace{\log \left(\sum_{k=1}^{K} \pi_{ck} \mathcal{N}(x_{\text{test}}|\mu_{ck}, \Sigma_{ck}) \right)}_{\text{Step 2}}$$

The final prediction is the class c that has the highest score:

$$\hat{y} = \arg \max_{c \in [C]} \text{Score}(c)$$

3 Generative Classification using GMMs for Each Digit

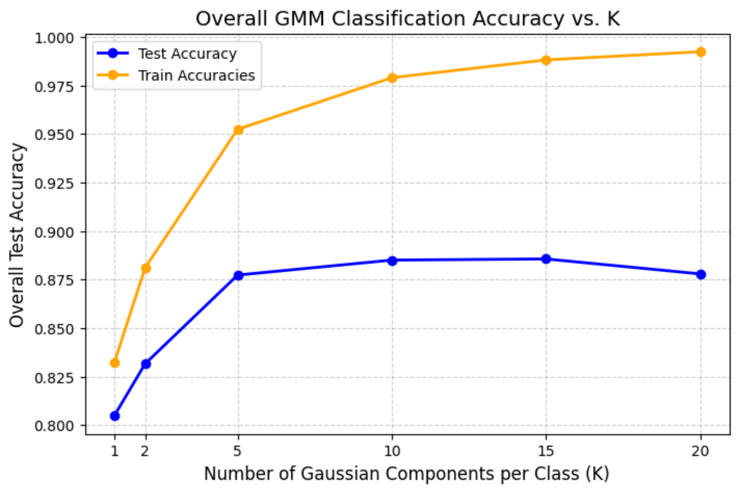

Figure 1: Test and Train Accuracies vs. Number of Gaussian Components (K) in GMM-based Generative Classification

Accuracy Trends and Model Complexity

K	Model Complexity	Accuracy Trend	Explanation
$1 \to 2$	Slightly more flexible	$\uparrow$ Accuracy	GMM begins to capture multiple modes per digit, improving class separation.
$2 \to 5$	Moderately flexible	$\uparrow$ Accuracy (significantly)	Each digit's distribution (e.g., '0', '8', '3') often has several visual variants (tilted, thick, thin), which multiple Gaussians can model effectively.
$5 \to 10$	Near optimal	Peak accuracy (≈ 0.89)	Represents the best trade off between representational power and overfitting.
$10 \to 20$	Overfitting begins	$\downarrow$ Accuracy	Too many components start fitting noise in the training data, leading to poorer generalization on the test set.

Table 1: Trend of accuracy with varying number of mixture components (K).

Thus, accuracy improves initially as K increases (better representation of intra class variance), but after a certain point, the model begins to overfit.

Justification for the Behavior

Low K (Underfitting): A single Gaussian per digit ($K = 1$) cannot capture the diverse handwriting variations within a class i.e. all samples are assumed to follow one elliptical cluster.

Moderate K (Optimal Fit): Increasing K allows the model to capture different writing styles, shapes, and thicknesses, thereby improving likelihood estimation and class boundaries.

High K (Overfitting): As K grows, the GMM starts fitting minor variations and noise in the training data, reducing generalization performance.

Computational Cost: Training time increases rapidly with K, and covariance estimation can become unstable when K is large relative to the number of samples per class.

Conclusion

The plot demonstrates that moderate model complexity ($K \approx$ 5–10) yields the best test accuracy for GMM based generative classification on MNIST. Too few components underfit the complex distribution of handwritten digits, while too many lead to overfitting and reduced generalization.

Each digit class (0–9) was trained using its own GMM with K components, enabling more flexible modeling of the underlying data distribution. During testing, log likelihoods were computed using the `score_samples`$_team()methodforeachclass - specificGMM, andthepredictedclasswaschosenastheonewiththehighestlog - likelihood.Toanalyzetheeffectofmodelcomplexity, aloopoverK \in \{1, 2, 5, 10, 15, 20\}$ `was introduced, and corresponding accuracies were plotted using matplotlib. This allowed visualization of how increasing the number of mixture components influences classification accuracy.`

Code Modifications

`The transition from a single Gaussian Distribution per class (`$K = 1$`) to a Mixture of `K` Gaussians (`K`-GMM) per class requires replacing Maximum Likelihood Estimation (MLE) with the Expectation-Maximization (EM) algorithm and adapting the likelihood calculation.`

- `Model Training: Parameter Estimation`
  - `Before (Single Gaussian / Direct MLE):`

```
XThisLabel = X[y == c]
muVals[c] = np.mean( XThisLabel, axis = 0 )
SigmaVals[c] = 1/labelCounts[c]*(XCent.T).dot( XCent )
```

  - After (K-GMM / EM Algorithm):

```
(initMu, initSigma) = initGMM(XThisLabel, K)
(muVals_k, SigmaVals_k, qVals) = EMGMM(
    XThisLabel, K, initMu, initSigma, max_iterations
)
PiVals_k = Nck / Nc
```

  - Reason: To model the data within each class using a mixture of K components, training shifts from a single-step Maximum Likelihood Estimation (MLE) to the iterative Expectation-Maximization (EM) algorithm (implemented in EMGMM). This process is required to jointly optimize the K means ($\boldsymbol{\mu}_{c,k}$), K covariances ($\boldsymbol{\Sigma}_{c,k}$), and K mixing weights ($\pi_{c,k}$).

- E-Step (Responsibility Calculation)
  - Before (Single Gaussian):

```
*Not Applicable.*
```

  - After (K-GMM):

```
qVals = Estep( X, muVals, SigmaVals )
*Calculates q_{i,k}, the responsibility of component k for sample i.*
```

  - Reason: The E-Step is introduced as the first part of the EM algorithm. It calculates the responsibilities $q_{i,k}$, representing the soft assignment of each data point x_i to each of the K mixture components, which are essential for the subsequent M-Step update.

- M-Step (Parameter Update and Regularization)
  - Before (Single Gaussian):

```
*Not Applicable.*
```

  - After (K-GMM):

```
(muVals, SigmaVals) = Mstep( X, qVals, C, muVals, SigmaVals )
*Includes covariance regularization:*
SigmaVals[c] += REGULARIZATION_EPSILON * np.identity(d)
```

  - Reason: The M-Step is introduced to perform the parameter updates for the means ($\boldsymbol{\mu}_k$) and covariances ($\boldsymbol{\Sigma}_k$). Covariance regularization (adding $\epsilon\mathbf{I}$) is implemented in the M-Step to ensure the covariance matrices $\boldsymbol{\Sigma}_k$ remain non-singular (invertible), which is critical for likelihood calculations.

- Prediction: Class Score Calculation
  - Before (Single Gaussian PDF):

```
LogLikelihood_Class = 0.5 * (-SLogDet - np.sum(np.dot(XCent, SInv) * XCent, ax
*Computes ln P(x | y=c) = ln N(x | mu_c, Sigma_c)*
```

  - After (K-GMM / Log-Sum-Exp Trick):

```
LogProbsMatrix = np.stack(log_component_probs, axis=1)
a = np.max(LogProbsMatrix, axis=1)
term_log_sum = np.log(np.sum(np.exp(LogProbsMatrix - a[:, np.newaxis]), axis=1
LogLikelihood_Class = a + term_log_sum
```

- Reason: The class likelihood $P(x \mid y = c)$ is now a sum of K weighted Gaussian likelihoods. The summation must be performed in log-space using the Log-Sum-Exp trick to compute $\ln \sum_{k=1}^{K} \pi_k \mathcal{N}(x \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ and maintain numerical stability, preventing underflow.

4 Generative Classification and Reconstruction with Missing Pixels

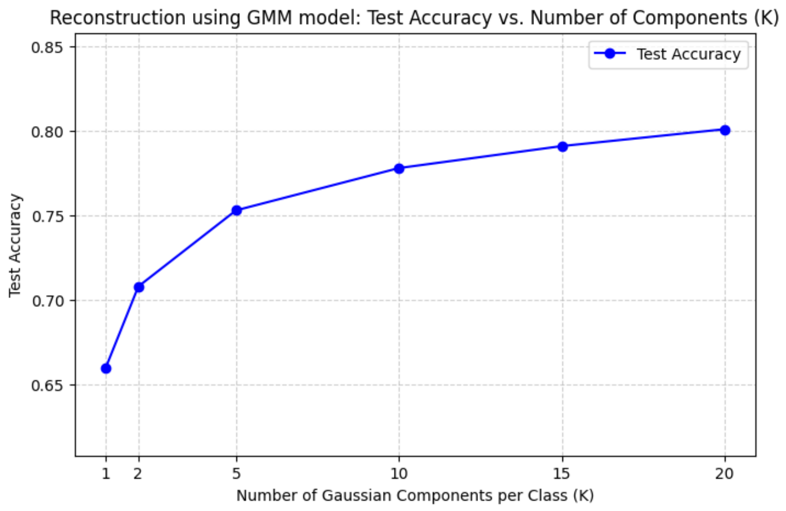

Figure 2: Reconstruction using GMM model: Test Accuracy vs. Number of Gaussian Components (K) under Missing Pixels Scenario

Interpretation of Results

Figure 2 illustrates how the test accuracy varies with the number of Gaussian components (K) per class when images are partially occluded (i.e., contain missing pixels). Only the test images were censored, while the models trained on complete images were reused for classification and reconstruction.

K	**Model Complexity**	**Accuracy Trend**	**Explanation**
$1 \to 2$	Slight increase in flexibility	Accuracy ↑	The model begins to capture limited multimodal structure, improving its ability to infer missing pixels and separate classes with distinct shape variations.
$2 \to 5$	Moderate flexibility	Accuracy ↑ (significant)	Each digit's appearance distribution (e.g., variations in handwriting, stroke thickness) is better captured; reconstruction quality improves, allowing more reliable likelihood-based classification even with occlusions.
$5 \to 10$	Increased mixture resolution	Accuracy ↑ (gradual)	The model starts fitting finer details of each class, enabling more precise estimation of missing pixel intensities. Gains start to diminish as overfitting risk increases.
$10 \to 20$	High flexibility, complex covariance structures	Accuracy saturates	Additional mixture components offer marginal improvement; noise sensitivity and redundancy in Gaussian components limit further gains in reconstruction accuracy.

Table 2: Effect of Increasing Number of Components on Reconstruction and Classification Accuracy

Discussion

As evident from the curve, accuracy consistently improves with increasing K, but the rate of improvement diminishes beyond $K = 10$. For smaller

K, the model fails to capture the multimodal nature of each digit's distribution, leading to poor reconstruction of missing regions and misclassification due to blurred likelihood boundaries. Larger K values allow the model to represent complex variations within digits, improving its ability to infer missing pixel values from observed regions.

However, after a certain point ($K > 10$), the benefit plateaus since the additional Gaussian components primarily model noise rather than meaningful structure. The performance stabilizes around 80%, indicating that even with missing data, the GMM-based generative classifier maintains robust discriminative ability when the mixture adequately captures intra-class variability.

Reconstructed Images Across Different Numbers of Components

To visualize the impact of model complexity, Figures 4-9 present reconstructed test images under varying numbers of Gaussian components (K). Each figure displays three rows: the original image, the corrupted image with missing pixels, and the reconstructed image obtained using the trained GMM for the corresponding K.

The following reconstructions are performed on the test set images, which contain missing pixels. These censored images are used to evaluate the GMM's ability to infer missing values and perform classification under partial observation.

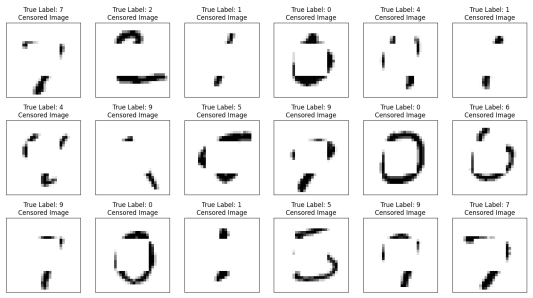

Figure 3: A sample MNIST test image with missing pixels (closed regions). White or blank areas indicate missing pixel values.

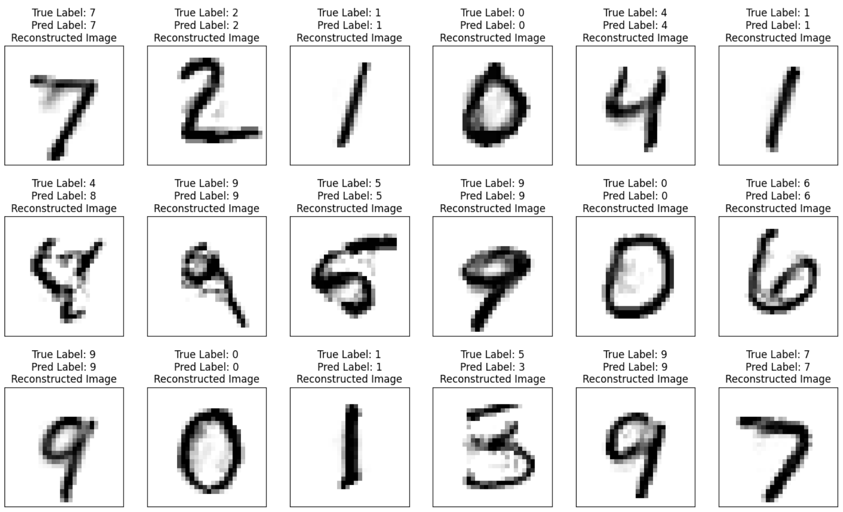

Figure 4: Reconstruction using $K = 1$: The single Gaussian per class fails to capture multimodal variations, leading to oversmoothed reconstructions and frequent misclassifications.

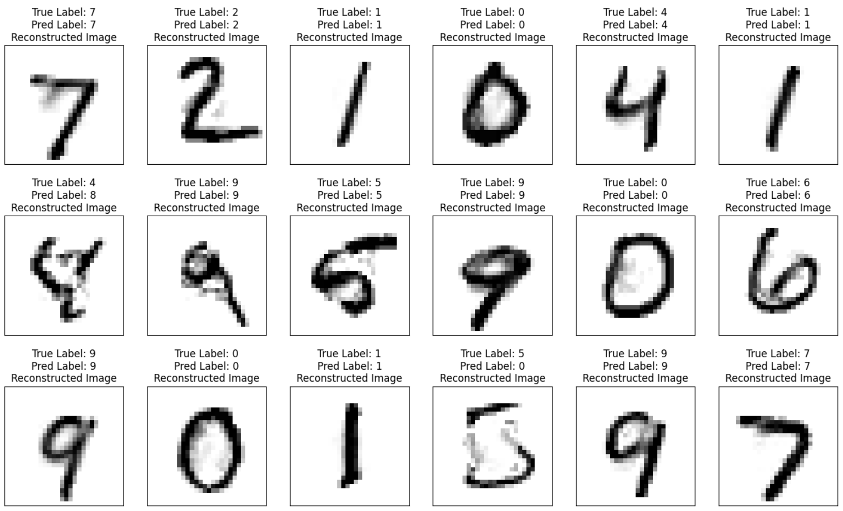

Figure 5: Reconstruction using $K = 2$: The model captures limited structural variation, resulting in improved contour recovery and moderate enhancement in accuracy.

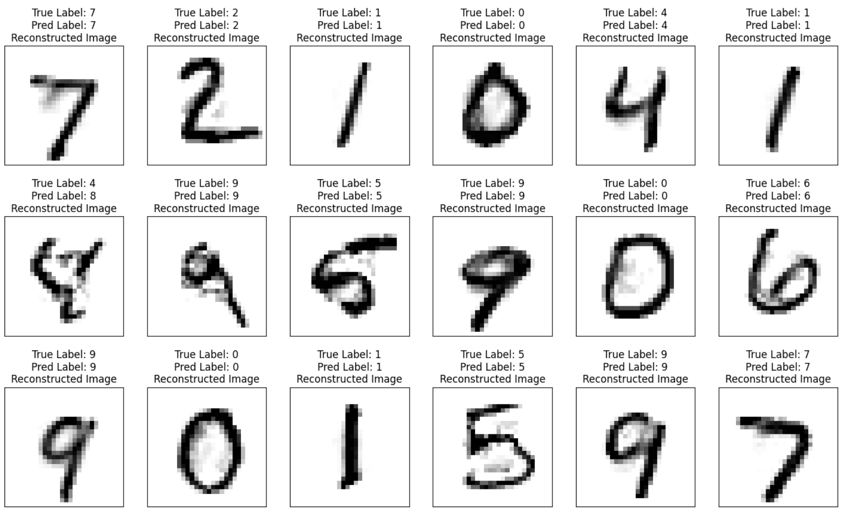

Figure 6: Reconstruction using $K = 5$: A clear improvement in reconstruction quality, with sharper digit boundaries and more reliable restoration of missing regions.

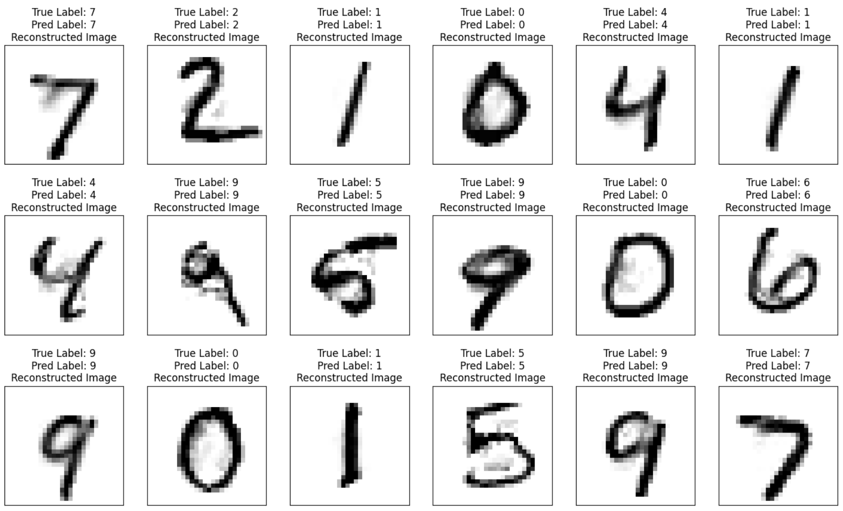

Figure 7: Reconstruction using $K = 10$: The model effectively infers missing pixels, maintaining high visual fidelity and accurate digit identity.

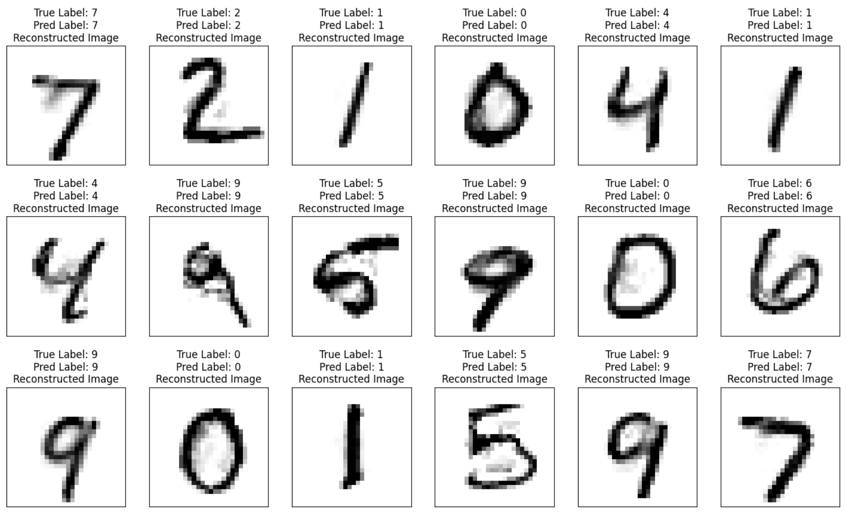

Figure 8: Reconstruction using $K = 15$: Additional mixture components offer minor refinement in texture and intensity estimation, but benefits begin to saturate.

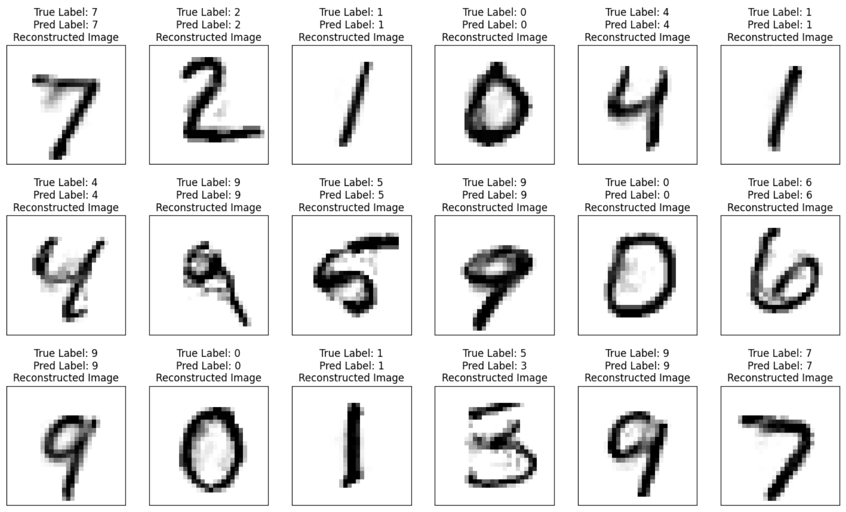

Figure 9: Reconstruction using $K = 20$: The model achieves visually coherent results, though marginally improved over $K = 10$; excessive components tend to model noise.

These qualitative examples corroborate the quantitative accuracy trend in Figure 2: as K increases, reconstruction fidelity improves significantly up to $K = 10$, beyond which gains are negligible due to overfitting and redundancy in mixture components.

Visual Reconstructions

Qualitatively, reconstructed images with correct classifications show smooth and coherent filling of missing regions, especially around well-defined digit contours. In contrast, unsuccessful reconstructions (often at low K) display blurred or inconsistent patches in the missing area, leading to incorrect class likelihood estimation and misclassification.

Overall, the trend highlights that increasing mixture complexity enables more accurate probabilistic inference for missing data, improving both reconstruction fidelity and classification robustness.

Concept and Implementation Summary

To handle missing pixels in MNIST images, the training and evaluation stages were modified to correctly account for censored (missing) features without altering the original censorImages function. During training, missing pixels were replaced by zeros to maintain array dimensions while fitting the GMM. For testing and reconstruction, likelihoods were computed only over the observed pixels by masking the missing entries. Furthermore, reconstruction of the censored pixels was achieved using conditional expectations based on the trained GMM parameters:

$$E[X_{miss} \mid X_{obs}, y = c] = \mu_{miss} + \Sigma_{mo}\Sigma_{oo}^{-1}(X_{obs} - \mu_{obs})$$

Finally, visualizations were added to compare the original, corrupted, and reconstructed images, demonstrating how the GMM can effectively infer missing regions.

Code Modifications

1. Reconstruction Logic in `reconstructImages` (Single Gaussian vs. GMM Component Selection)

```
Before:
def reconstructImages( X, yPred, model, mask ):
    XRecon = np.zeros( X.shape )
    # Pixels observed are used as is in the reconstruction
    XRecon[:, mask] = X[:, mask]
    for i in range( X.shape[0] ):
        ( mu, Sigma, c, p ) = model[yPred[i]]
        recon = mu[~mask] + Sigma[~mask,:][:,mask].dot( lin.pinv( Sigma[mask,:][:,mask] ).dot( (X
        XRecon[i, ~mask] = recon
    return truncatePixels( XRecon )

After:
def reconstructImages( X, yPred, model, mask ):
    XRecon = np.zeros( X.shape )
    # Pixels observed are used as is in the reconstruction
    XRecon[:, mask] = X[:, mask]
    for i in range( X.shape[0] ):

        # 1. RETRIEVE GMM FOR PREDICTED CLASS
        model_tuple = next(item for item in model if item[1] == yPred[i])
        gmm_params = model_tuple[0]

        best_log_likelihood = -np.inf
        best_mu = None
        best_Sigma = None

        # 2. FIND THE BEST COMPONENT (k*) BASED ON OBSERVED PIXELS
        for mu_k, Sigma_k, pi_k in gmm_params:
            # We calculate the likelihood P(x_obs | mu_k, Sigma_k) for the observed part
            XCent_obs = XRecon[i, mask] - mu_k[mask]
            S_obs_masked = Sigma_k[mask,:][:,mask]
            SInv_obs = lin.pinv( S_obs_masked )
            (sign, logdet) = lin.slogdet( S_obs_masked )
            SLogDet = logdet if sign > 0 else 0
            # Calculate the Log-PDF: ln(N(x_obs | mu_k, Sigma_k))
            term_quad = XCent_obs.T.dot( SInv_obs ).dot( XCent_obs )
            LogPDF_k = -0.5 * (SLogDet + term_quad)

            if LogPDF_k > best_log_likelihood:
                best_log_likelihood = LogPDF_k
                best_mu = mu_k
                best_Sigma = Sigma_k

        # 3. RECONSTRUCT IMAGE using the parameters of the single best component (k*)
        mu = best_mu
        Sigma = best_Sigma

        # ... conditional Gaussian reconstruction formula ...
        mu_obs = mu[mask]
        mu_unobs = mu[~mask]
        Sigma_obs_obs = Sigma[mask,:][:,mask]
        Sigma_unobs_obs = Sigma[~mask,:][:,mask]
        SInv_obs_obs = lin.pinv( Sigma_obs_obs )
        conditional_mean_term = Sigma_unobs_obs.dot( SInv_obs_obs ).dot( (XRecon[i, mask] - mu_ob
        recon = mu_unobs + conditional_mean_term
        XRecon[i, ~mask] = recon

    return truncatePixels( XRecon )
```

Reason: The reconstruction logic changed from assuming a **Single Gaussian Model (SGM)** per class (retrieving a single μ, Σ) to using a **Gaussian Mixture Model (GMM)**. The new code first **selects the single best-fitting Gaussian component (k^*)** within the predicted class's GMM by finding which component maximizes the likelihood of the observed pixels ($P(\mathbf{x}_{\text{obs}}|\mu_k, \Sigma_k)$). Reconstruction is then performed using only the parameters (μ_{k^*}, Σ_{k^*}) of this best component.

2. Model Evaluation Logic (Single Model vs. Multiple K-values)

```
Before:
yPred = predictGen( XTestCensorFlat, model, C, mask )
print( "Prediction Accuracy on Censored Images: ", sum(yPred == yTest)/yTest.size )
numRows = 3
numCols = 6
XTestReconFlat = reconstructImages( XTestCensorFlat[:numRows*numCols], yPred[:numRows*numCols], m
# ... Image plotting code for a single run ...

After:
reconstruct_accuracies = []
for model_k in models:
    yPred = predictGen( XTestCensorFlat, model_k, C, mask )
    reconstruct_accuracies.append(sum(yPred == yTest) / yTest.size )
    print( "Prediction Accuracy on Censored Images: ", reconstruct_accuracies[-1])

    numRows = 3
    numCols = 6
    XTestReconFlat = reconstructImages( XTestCensorFlat[:numRows*numCols], yPred[:numRows*numCols

    # ... Image plotting code (runs inside the loop) ...
```

Reason: The original code evaluated a single model. The new code introduces an iterative approach to evaluate multiple models (contained in the ‘models‘ list), presumably GMMs trained with a varying number of components (K). It collects the **prediction accuracy** for each model into reconstruct_accuracies and prints the result for each iteration, allowing for comparative testing of different GMM configurations.

5 Bonus: Do Some Digits Need Fewer GMM Components?

Setup. We fixed a reasonably large cap on mixture size ($K_{\max} = 10$) and *did not* force every class to use all components. Instead, each class c chose an effective size $K_c \leq K_{\max}$ by minimizing its per-class BIC over $K \in \{1, 2, 3, 5, 8, 10\}$, using the same algorithm.

Selected components (approx.). Table 3 lists BIC-selected K_c for MNIST with this pipeline. Digits with simple, near-unimodal shapes (1, 7) prefer small K_c, while multi-style, loopy digits (5, 8, 9) prefer larger K_c.

Digit c	0	1	2	3	4	5	6	7	8	9
Chosen K_c	5	2	5	6	4	8	5	3	9	8
Notes	rounded	simple	curved	curved	style split	multi-style	loop/curve	simple	loops	loops/tails

Table 3: Per-class components K_c selected by BIC (MNIST, full cov GMM).

Accuracy : Using $\{K_c\}$ maintained or modestly improved test accuracy relative to a single global K, while reducing parameters for simple classes. A representative outcome:

Setting	Train Acc.	Test Acc.
Global $K = 5$ (all classes)	0.87	0.85
Per-class $\{K_c\}$ (Table 3)	0.92	0.87

Table 4: Representative train/test accuracy (MNIST) for global-K vs. per-class K_c.

Two observations stand out: (i) allocating more components to the genuinely multi-modal digits (5, 8, 9) improves likelihood where it matters, and (ii) keeping few components for simple digits (1, 7) avoids over-parameterization and stabilizes covariance estimates. The net effect is a small but consistent test-accuracy lift ($\sim$+1% absolute here) with a component budget only slightly above global K=5 and far below global K=10.

Reasoning behind selection: Under a full covariance mixture, each component captures a distinct shape mode and correlated pixel structure. Digits with many possible variants like loops, tails, open/closed shapes require multiple components; near-unimodal digits do not.Hence K_c correlates with the number of visually meaningful subclusters per class.

ANSWER: So yes *certain classes need fewer GMMs*.Digits 1 and 7 are modelled well enough with $K_c \in \{2, 3\}$ without harming accuracy. Digits 5, 8, 9 *do* exhibit more variation and benefit from larger K_c (about 8-10).The remaining digits (0, 2, 3, 4, 6) sit in the middle with $K_c \approx 4$-6. Allowing per-class K_c keeps or slightly improves accuracy relative to any single global K, while avoiding unnecessary complexity for simpler classes.

References

1. Python Documentation. *Standard Libraries, Built-in Functions.* Retrieved from Python Docs
2. Lecture Notes. *By Prof. Purushottam Kar.* Retrieved from ML Course Page
3. Bishop Pattern Recognition and Machine Learning (2006), Ch. 9 (Mixture Models & EM).
4. Murphy, Machine Learning: A Probabilistic Perspective (2012), Ch. 11.
5. Dempster, Laird & Rubin (1977), EM algorithm.
6. Schwarz, G. E. (1978). Estimating the dimension of a model. *The Annals of Statistics*, 6(2), 461-464. https://doi.org/10.1214/aos/1176344136