ECE 271A: Statistical Learning I Quiz Report

JunPyung Kim
PID: A69044427
University of California, San Diego

December 3, 2025

1 Quiz 6: Gaussian Mixture Models for Cheetah Segmentation

1.1 Objective

In this quiz we revisit the cheetah image segmentation task using Gaussian mixture models (GMMs) estimated by the EM algorithm. The goals are:

- to study how sensitive the EM-trained GMMs are to random initialization when the number of components is fixed at $C = 8$, and
- to analyze how the number of mixture components $C \in \{1, 2, 4, 8, 16, 32\}$ affects the probability of error as we vary the feature dimension $d \in \{1, 2, 4, 8, 16, 24, 32, 40, 48, 56, 64\}$.

1.2 Methodology

1.2.1 Data and Feature Extraction

The training data are the DCT features provided in `TrainingSamplesDCT_8_new.mat`. There are 250 foreground (FG) samples and 1053 background (BG) samples, so the empirical class priors are

$$P(Y = \text{FG}) = \frac{250}{250 + 1053} \approx 0.192, \qquad P(Y = \text{BG}) = \frac{1053}{250 + 1053} \approx 0.808.$$

The test image is the cheetah image `cheetah.bmp`. It is converted to grayscale and processed with a sliding 8×8 window (stride 1) over the entire 255×270 image. For each block we compute the 8×8 2-D DCT and then reorder the 64 coefficients using the zig–zag pattern given in `Zig-Zag Pattern.txt`. This produces 68850 feature vectors $\mathbf{x} \in \mathbb{R}^{64}$, one for each pixel position. The ground-truth mask is obtained from `cheetah_mask.bmp` by thresholding at 0.5.

1.2.2 Gaussian Mixture Model

For each class $Y \in \{\text{FG}, \text{BG}\}$ we model the conditional density of the full 64-dimensional feature vector as a mixture of C Gaussians with diagonal covariance matrices:

$$p(\mathbf{x} \mid Y = i) = \sum_{c=1}^{C} \pi_{i,c} \mathcal{N}\left(\mathbf{x}; \boldsymbol{\mu}_{i,c}, \Sigma_{i,c}\right), \tag{1}$$

where $\sum_c \pi_{i,c} = 1$ and $\Sigma_{i,c} = \text{diag}(\sigma^2_{i,c,1}, \ldots, \sigma^2_{i,c,64})$.

The models are trained with the EM algorithm. For a given class and C:

- **Initialization:** The means $\boldsymbol{\mu}_{i,c}$ are initialized by selecting random training samples and adding small noise. All mixture weights are set to $\pi_{i,c} = 1/C$, and the initial diagonal variances are copied from the global variance of the training set (replicated across components).

- **E-step:** For each sample $\mathbf{x}_n$ and component c we compute the log responsibility $\log \gamma_{n,c} = \log p(c \mid \mathbf{x}_n, Y = i)$ using the current parameters. This uses the diagonal Gaussian log-likelihood together with the mixture weights.

- **M-step:** From the responsibilities $\gamma_{n,c}$ we update

$$N_c = \sum_n \gamma_{n,c}, \quad \pi_{i,c} = \frac{N_c}{N},$$

$$\boldsymbol{\mu}_{i,c} = \frac{1}{N_c} \sum_n \gamma_{n,c} \mathbf{x}_n, \qquad \sigma^2_{i,c,d} = \frac{1}{N_c} \sum_n \gamma_{n,c} (x_{n,d} - \mu_{i,c,d})^2 + \varepsilon,$$

  where a small ε ensures numerical stability.

Iterations stop when the log-likelihood improvement becomes smaller than a tolerance threshold, or when the maximum number of iterations is reached.

1.2.3 Bayes Decision Rule and Error Computation

Once the FG and BG mixtures have been trained, we classify each image block using the Bayes decision rule

$$g(\mathbf{x}) = \log p(\mathbf{x} \mid Y = \text{FG}) + \log P(Y = \text{FG}) - \log p(\mathbf{x} \mid Y = \text{BG}) - \log P(Y = \text{BG}).$$

The pixel is labeled as cheetah if $g(\mathbf{x}) > 0$ and as grass otherwise. To study the effect of the feature dimensionality d, we always train the mixtures in the full 64-dimensional space, but at test time we keep only the first d zig–zag coefficients. This is implemented by restricting the mean and variance vectors to their first d entries when computing the log-likelihood.

Let $\hat{Y}$ denote the classifier output and Y the true class (from the mask). The probability of error is estimated as

$$P_e = P(\hat{Y} \neq Y) = P(\hat{Y} \neq Y \mid Y = \text{FG}) P(Y = \text{FG}) + P(\hat{Y} \neq Y \mid Y = \text{BG}) P(Y = \text{BG}).$$

The conditional error terms correspond to the proportion of misclassified foreground and background pixels in the ground-truth mask.

1.3 Results and Discussion

1.3.1 Part (a): Effect of Initialization for $C = 8$ Components

For Part (a) we fix $C = 8$ components for both classes and train 5 independent mixtures for FG and 5 for BG using different random initializations. Combining them yields 25 classifier pairs; for each pair we compute the probability of error for all dimensions d in the set $\{1, 2, 4, 8, 16, 24, 32, 40, 48, 56, 64\}$. The resulting curves are shown in Figure 1.

Several qualitative trends can be observed:

- For all initialization pairs the error is highest when only the first DCT coefficient is used ($d = 1$) and remains relatively large for very low dimensions ($d \leq 8$), since a single or a few coefficients cannot capture the structure of the two classes.

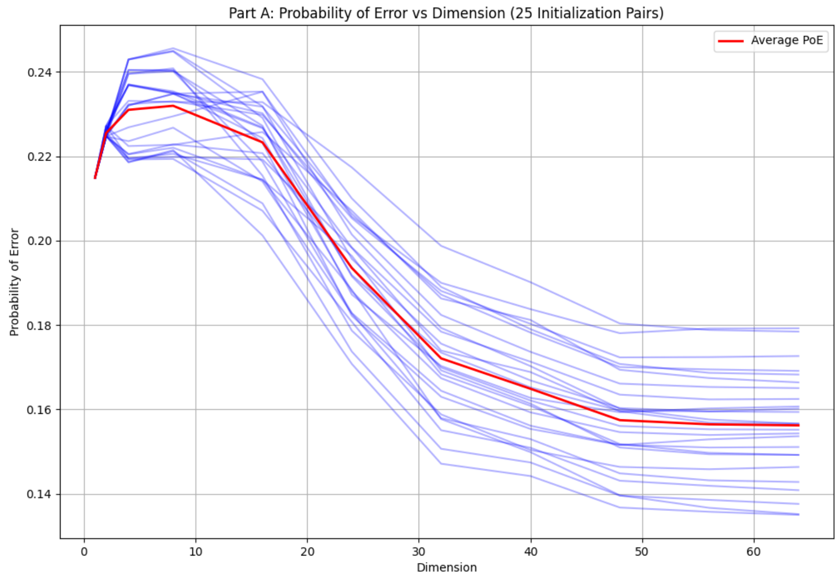

Figure 1: Part (a): probability of error vs. feature dimension for the 25 pairs of FG/BG mixtures obtained with different random initializations (blue curves) and their average (red curve).

- As more coefficients are included (d in the range roughly 16–40), the error drops significantly. The average curve shows a clear improvement when going from a handful of coefficients to a moderate-dimensional descriptor, reflecting that the mid-frequency DCT features are important for discriminating cheetah texture from grass.
- Beyond some dimension, gains become smaller and the curves tend to flatten. Some initializations even exhibit a slight increase in error at the largest dimensions, consistent with adding noisy or less informative coefficients and increasing estimation variance.
- The spread between the 25 curves is noticeable but not extreme: all runs follow the same global shape and have similar minima. This indicates that EM is somewhat sensitive to initialization (local maxima of the likelihood) but that the overall classifier performance is relatively robust: different initializations rarely change the error by more than a few percentage points.

Overall, Part (a) demonstrates that, although GMM training with EM does depend on the random start, the dominant factor affecting the probability of error is the feature dimension d: using only a very small number of coefficients is clearly sub-optimal, while using a moderate number of DCT features significantly improves segmentation quality.

1.3.2 Part (b): Effect of the Number of Components

For Part (b) we fix the initialization strategy and vary the number of mixture components

$$C \in \{1, 2, 4, 8, 16, 32\}$$

separately for FG and BG. For each C we train one GMM per class on the full 64-dimensional training data and then evaluate the probability of error as a function of the dimension d. The curves are shown in Figure 2.

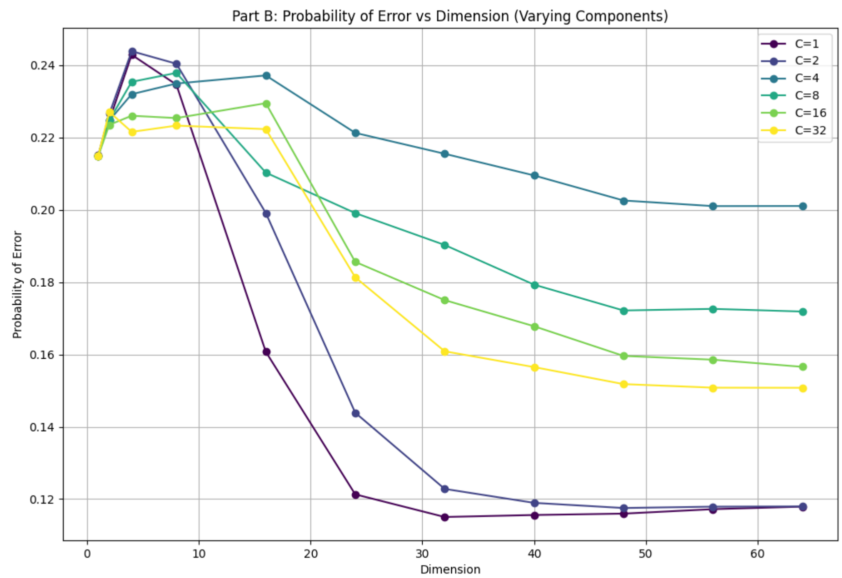

Figure 2: Part (b): probability of error vs. feature dimension for mixtures with different numbers of components $C \in \{1, 2, 4, 8, 16, 32\}$.

Some observations based on the values are:

- For very low dimensions ($d \leq 8$) all choices of C lead to very similar error rates (around 0.22–0.24). With so few features the representation is too limited for the additional mixture components to help.
- When more features are included, the models behave quite differently. For example, at $d = 32$ the estimated probability of error is approximately $P_e \approx 0.115$ for $C = 1$, 0.123 for $C = 2$, 0.216 for $C = 4$, 0.190 for $C = 8$, 0.175 for $C = 16$, and 0.161 for $C = 32$. The single-Gaussian model ($C = 1$) actually achieves the lowest error in this experiment.
- As C increases from 4 to 32 the curves tend to improve (e.g., the error for $C = 32$ at large d is lower than for $C = 4$ or $C = 8$), suggesting that additional components can help the model better fit the complex foreground and background distributions. However, none of the larger mixtures shows better performance than the simple $C = 1$ model on this task.
- For each fixed C, the error generally decreases as d grows from 16 to around 48–64, then slowly stabilizes. This is most evident for $C = 32$, whose error drops from about 0.22 at $d = 16$ to about 0.15 at $d = 56$–64.

These results highlight that in theory increasing the number of mixture components should never hurt and should allow a more flexible approximation of the true class-conditional densities. In practice, with limited training data and diagonal covariances, the EM algorithm can overfit and converge to poor local optima when C is large. In our experiment the single-Gaussian model is strong, and adding more components often increases the probability of error instead of reducing it.

1.4 Conclusion

In this quiz we built GMM-based classifiers for cheetah vs. grass segmentation using DCT features of 8×8 image blocks. Part (a) showed that, while EM initialization causes some variation in performance, the overall shape of the probability-of-error curves is consistent across runs: using only a few DCT coefficients leads to high error, and using a moderate-to-large subset of coefficients substantially improves segmentation.

Part (b) demonstrated that simply increasing the number of mixture components does not guarantee better performance. With the available training data and diagonal covariances, a single Gaussian per class already models the DCT feature distributions well, and larger mixtures may overfit or get trapped in suboptimal local maxima of the likelihood. This illustrates the trade-off between model complexity and robustness in generative classification and emphasizes the need to validate mixture complexity on a separate performance metric such as segmentation error.

1.5 Appendix: Source Code

The Python source code used to generate these results is included below.

```python
import numpy as np
import scipy.io
import scipy.stats
import imageio.v3 as imageio
import matplotlib.pyplot as plt
import os
from scipy.fftpack import dctn
from tqdm import tqdm

BLOCK_SIZE = 8
DATA_DIR = 'data/' if os.path.exists('data/') else '.'
TRAIN_FILE = os.path.join(DATA_DIR, 'TrainingSamplesDCT_8_new.mat')
IMAGE_FILE = os.path.join(DATA_DIR, 'cheetah.bmp')
MASK_FILE = os.path.join(DATA_DIR, 'cheetah_mask.bmp')
ZIGZAG_FILE = os.path.join(DATA_DIR, 'Zig-Zag Pattern.txt')
OUTPUT_DIR = 'hw4/output/'

os.makedirs(OUTPUT_DIR, exist_ok=True)


def load_zigzag_pattern(path):
    """Loads the Zig-Zag pattern from a text file."""
    if not os.path.exists(path):
        return np.array([
            0, 1, 5, 6, 14, 15, 27, 28,
            2, 4, 7, 13, 16, 26, 29, 42,
            3, 8, 12, 17, 25, 30, 41, 43,
            9, 11, 18, 24, 31, 40, 44, 53,
            10, 19, 23, 32, 39, 45, 52, 54,
            20, 22, 33, 38, 46, 51, 55, 60,
            21, 34, 37, 47, 50, 56, 59, 61,
            35, 36, 48, 49, 57, 58, 62, 63
        ])
    with open(path, 'r') as f:
        lines = f.readlines()
    data = []
    for line in lines:
        data.extend([int(x) for x in line.split()])
```

```
    return np.argsort(data)


def compute_dct_features(img, zigzag_order):
    """
    Computes DCT features for an image using an 8x8 sliding window.
    Returns an (N, 64) array of features and the image dimensions.
    """
    img = np.array(img, dtype=float) / 255.0

    if img.ndim == 3:
        img = np.mean(img, axis=2)

    h, w = img.shape

    pad_h = BLOCK_SIZE - 1
    pad_w = BLOCK_SIZE - 1
    img_padded = np.pad(img, ((0, pad_h), (0, pad_w)), 'constant', constant_values=0)

    features = []

    from numpy.lib.stride_tricks import sliding_window_view
    windows = sliding_window_view(img_padded, (BLOCK_SIZE, BLOCK_SIZE))

    num_blocks = h * w
    flat_blocks = windows.reshape(num_blocks, BLOCK_SIZE, BLOCK_SIZE)

    print(f"Computing DCT for {num_blocks} blocks...")

    dct_blocks = dctn(flat_blocks, type=2, norm='ortho', axes=(1, 2))

    dct_flat = dct_blocks.reshape(num_blocks, 64)
    features = dct_flat[:, zigzag_order]

    return features, h, w


class GMMDiagonal:
    """
    Gaussian Mixture Model with Diagonal Covariance matrices trained via EM.
    """

    def __init__(self, n_components, n_iter=100, tol=1e-4, min_covar=1e-6):
        self.n_components = n_components
        self.n_iter = n_iter
        self.tol = tol
        self.min_covar = min_covar
        self.weights = None
        self.means = None
        self.covariances = None
        self.converged_ = False

    def fit(self, X):
        """Trains the model using EM."""
        n_samples, n_features = X.shape

        indices = np.random.choice(n_samples, self.n_components, replace=False)
```

```
        self.means = X[indices] + np.random.rand(self.n_components, n_features) * 0.01

        global_var = np.var(X, axis=0)
        self.covariances = np.tile(global_var, (self.n_components, 1))

        self.weights = np.ones(self.n_components) / self.n_components

        log_likelihood_old = -np.inf

        for i in range(self.n_iter):
            log_resp, log_likelihood = self._e_step(X)

            if np.abs(log_likelihood - log_likelihood_old) < self.tol:
                self.converged_ = True
                break
            log_likelihood_old = log_likelihood

            self._m_step(X, log_resp)

    def _e_step(self, X):
        """Expectation step: calculate log responsibilities."""
        n_samples, n_features = X.shape
        weighted_log_prob = np.zeros((n_samples, self.n_components))

        const = -0.5 * n_features * np.log(2 * np.pi)

        for c in range(self.n_components):
            log_det = np.sum(np.log(self.covariances[c]))

            diff = X - self.means[c]
            mahalanobis = np.sum((diff ** 2) / self.covariances[c], axis=1)

            log_prob = const - 0.5 * (log_det + mahalanobis)
            weighted_log_prob[:, c] = np.log(self.weights[c] + 1e-300) + log_prob

        log_prob_norm = scipy.special.logsumexp(weighted_log_prob, axis=1)
        log_resp = weighted_log_prob - log_prob_norm[:, np.newaxis]

        return log_resp, np.mean(log_prob_norm)

    def _m_step(self, X, log_resp):
        """Maximization step: update parameters."""
        n_samples = X.shape[0]
        resp = np.exp(log_resp)

        Nk = np.sum(resp, axis=0) + 1e-10

        self.weights = Nk / n_samples

        self.means = (resp.T @ X) / Nk[:, np.newaxis]

        for c in range(self.n_components):
            diff = X - self.means[c]
            self.covariances[c] = np.sum(resp[:, c:c + 1] * (diff ** 2), axis=0) / Nk[c]

            self.covariances[c] += self.min_covar

```

```
    def score_samples(self, X):
        """Computes weighted log probability P(X|Model) for BDR."""
        n_samples, n_features = X.shape
        const = -0.5 * n_features * np.log(2 * np.pi)
        weighted_log_prob = np.zeros((n_samples, self.n_components))

        for c in range(self.n_components):
            log_det = np.sum(np.log(self.covariances[c]))
            diff = X - self.means[c]
            mahalanobis = np.sum((diff ** 2) / self.covariances[c], axis=1)
            log_prob = const - 0.5 * (log_det + mahalanobis)
            weighted_log_prob[:, c] = np.log(self.weights[c] + 1e-300) + log_prob

        return scipy.special.logsumexp(weighted_log_prob, axis=1)


def solve_problem():
    print("Loading data...")
    mat_data = scipy.io.loadmat(TRAIN_FILE)
    train_fg = mat_data['TrainsampleDCT_FG']
    train_bg = mat_data['TrainsampleDCT_BG']

    cheetah_img = imageio.imread(IMAGE_FILE)
    cheetah_mask = imageio.imread(MASK_FILE)

    cheetah_mask = (cheetah_mask > 127).astype(int)

    zigzag = load_zigzag_pattern(ZIGZAG_FILE)

    n_fg = train_fg.shape[0]
    n_bg = train_bg.shape[0]
    prior_fg = n_fg / (n_fg + n_bg)
    prior_bg = n_bg / (n_fg + n_bg)

    print(f"Priors: FG={prior_fg:.4f}, BG={prior_bg:.4f}")

    print("Extracting test image features...")
    test_features, h, w = compute_dct_features(cheetah_img, zigzag)

    dim_list = [1, 2, 4, 8, 16, 24, 32, 40, 48, 56, 64]

    print("\n--- Part A: 5 Mixtures of 8 Components ---")

    n_runs = 5
    n_components = 8

    fg_models = []
    bg_models = []

    print(f"Training {n_runs} models for FG (C={n_components})...")
    for i in range(n_runs):
        gmm = GMMDiagonal(n_components=n_components, n_iter=200)
        gmm.fit(train_fg)
        fg_models.append(gmm)

    print(f"Training {n_runs} models for BG (C={n_components})...")
    for i in range(n_runs):
        gmm = GMMDiagonal(n_components=n_components, n_iter=200)
        gmm.fit(train_bg)
```

```
        bg_models.append(gmm)

    part_a_errors = []

    plt.figure(figsize=(12, 8))

    print("Evaluating 25 classifier pairs...")
    for i in range(n_runs):
        for j in range(n_runs):
            fg_model = fg_models[i]
            bg_model = bg_models[j]

            errors = []

            for dim in dim_list:
                X_test_dim = test_features[:, :dim]

                def get_log_prob(model, X_d, d):
                    temp_gmm = GMMDiagonal(model.n_components)
                    temp_gmm.weights = model.weights
                    temp_gmm.means = model.means[:, :d]
                    temp_gmm.covariances = model.covariances[:, :d]
                    return temp_gmm.score_samples(X_d)

                log_prob_fg = get_log_prob(fg_model, X_test_dim, dim)
                log_prob_bg = get_log_prob(bg_model, X_test_dim, dim)

                discriminant = (log_prob_fg + np.log(prior_fg)) - (log_prob_bg + np.log(prior_bg))
                pred_mask = (discriminant > 0).astype(int).reshape(h, w)

                mask_flat = cheetah_mask.flatten()
                pred_flat = pred_mask.flatten()

                idx_fg = (mask_flat == 1)
                idx_bg = (mask_flat == 0)

                err_fg = np.sum(pred_flat[idx_fg] == 0) / np.sum(idx_fg)
                err_bg = np.sum(pred_flat[idx_bg] == 1) / np.sum(idx_bg)

                total_error = err_fg * prior_fg + err_bg * prior_bg
                errors.append(total_error)

            part_a_errors.append(errors)
            plt.plot(dim_list, errors, color='blue', alpha=0.3)

    avg_errors = np.mean(part_a_errors, axis=0)
    plt.plot(dim_list, avg_errors, color='red', linewidth=2, label='Average PoE')

    plt.title('Part A: Probability of Error vs Dimension (25 Initialization Pairs)')
    plt.xlabel('Dimension')
    plt.ylabel('Probability of Error')
    plt.grid(True)
    plt.legend()
    plt.savefig(os.path.join(OUTPUT_DIR, 'prob6_a_initialization.png'))
    plt.close()
    print(f"Part A Plot saved to {OUTPUT_DIR}")
```

```python
    # --- Part B: Varying Components ---
    print("\n--- Part B: Varying Component Counts ---")
    components_list = [1, 2, 4, 8, 16, 32]

    plt.figure(figsize=(12, 8))
    cmap = plt.get_cmap('viridis')
    colors = [cmap(i) for i in np.linspace(0, 1, len(components_list))]

    for idx, C in enumerate(components_list):
        print(f"Training mixture with C={C}...")

        gmm_fg = GMMDiagonal(n_components=C, n_iter=200)
        gmm_fg.fit(train_fg)

        gmm_bg = GMMDiagonal(n_components=C, n_iter=200)
        gmm_bg.fit(train_bg)

        errors_c = []

        for dim in dim_list:
            X_test_dim = test_features[:, :dim]

            def get_log_prob_c(model, X_d, d):
                temp_gmm = GMMDiagonal(model.n_components)
                temp_gmm.weights = model.weights
                temp_gmm.means = model.means[:, :d]
                temp_gmm.covariances = model.covariances[:, :d]
                return temp_gmm.score_samples(X_d)

            log_prob_fg = get_log_prob_c(gmm_fg, X_test_dim, dim)
            log_prob_bg = get_log_prob_c(gmm_bg, X_test_dim, dim)

            discriminant = (log_prob_fg + np.log(prior_fg)) - (log_prob_bg + np.
log(prior_bg))
            pred_flat = (discriminant > 0).astype(int)

            mask_flat = cheetah_mask.flatten()
            idx_fg = (mask_flat == 1)
            idx_bg = (mask_flat == 0)

            err_fg = np.sum(pred_flat[idx_fg] == 0) / np.sum(idx_fg)
            err_bg = np.sum(pred_flat[idx_bg] == 1) / np.sum(idx_bg)

            total_error = err_fg * prior_fg + err_bg * prior_bg
            errors_c.append(total_error)

        plt.plot(dim_list, errors_c, marker='o', label=f'C={C}', color=colors[idx
])
        print(f"  C={C} Errors: {['{:.4f}'.format(e) for e in errors_c]}")

    plt.title('Part B: Probability of Error vs Dimension (Varying Components)')
    plt.xlabel('Dimension')
    plt.ylabel('Probability of Error')
    plt.grid(True)
    plt.legend()
    plt.savefig(os.path.join(OUTPUT_DIR, 'prob6_b_components.png'))
    plt.close()
    print(f"Part B Plot saved to {OUTPUT_DIR}")
    print("Done.")
```

```python


if __name__ == "__main__":
    solve_problem()
```

Listing 1: Python code for HW4