Applied Stochastic Assignment 4

Manyara Baraka - mbaraka

November 14, 2024

Question 1: MoM, MLE, Bias and Consistency (20 points)

Consider a normal distribution defined by the probability density function (PDF):

$$f(y;\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-\mu)^2}{2\sigma^2}}, \quad -\infty < y < \infty,$$

where μ is the mean and σ^2 is the variance. Given a random sample $Y = \{y_1, y_2, \ldots, y_n\}$ drawn from this normal distribution, perform the following tasks:

(a) **(5 points)** Use the method of moments to derive the estimators for μ and σ^2.

- **First moment** is the mean μ

$$\bar{Y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

- **Second moment** is the variance σ^2

$$S^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{Y})^2$$

Therefore **the the μ is found by the first moment** $= \bar{Y}$
and the **varience σ^2 found by the second moment** $= S^2$

(b) **(5 points) Derive the Maximum Likelihood Estimators (MLE) for μ and σ^2.**
The likelihood function is:

$$L(\mu,\sigma^2) = \prod_{i=1}^{n} f(y_i;\mu,\sigma^2) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i-\mu)^2}{2\sigma^2}\right).$$

The log-likelihood function:

$$\ln L(\mu,\sigma^2) = \sum_{i=1}^{n} \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i-\mu)^2}{2\sigma^2}\right)\right).$$

Simplifying:

$$\ln L(\mu, \sigma^2) = -\frac{n}{2}\ln(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \mu)^2.$$

Maximum Likelihood Estimators for μ

Perform a partial derivative of $\ln L(\mu, \sigma^2)$ with respect to μ:

$$\frac{\partial}{\partial\mu}\ln L(\mu, \sigma^2) = \frac{1}{\sigma^2}\sum_{i=1}^{n}(y_i - \mu) = 0.$$

Simplifying μ:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} y_i = \bar{y}.$$

μ is:

$$\hat{\mu} = \bar{y}.$$

Maximum Likelihood Estimator for σ^2

Perform partial derivative of $\ln L(\mu, \sigma^2)$ with respect to σ^2:

$$\frac{\partial}{\partial\sigma^2}\ln L(\mu, \sigma^2) = -\frac{n}{2\sigma^2} + \frac{1}{2\sigma^4}\sum_{i=1}^{n}(y_i - \mu)^2 = 0.$$

Multiplying both sides by $2\sigma^4$:

$$-n\sigma^2 + \sum_{i=1}^{n}(y_i - \mu)^2 = 0.$$

σ^2:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mu)^2.$$

Substitute $\mu = \bar{y}$, we get σ^2 as:

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2.$$

Therefore:

The Maximum Likelihood Estimators for μ and σ^2 are:

$$\hat{\mu} = \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i,$$

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2.$$

(c) **(3 points) Calculate the bias of the MoM estimators $\hat{\mu}_{\mathbf{MoM}}$ and $\hat{\sigma}^2_{\mathbf{MoM}}$.**

From part a, we used the method of moments to define the estimators of μ and σ^2:

1. **For the mean μ:**

$$\hat{\mu}_{\text{MoM}} = \bar{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i.$$

2. **For the variance σ^2:**

$$\hat{\sigma}^2_{\text{MoM}} = S^2 = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \bar{Y})^2.$$

Bias of $\hat{\mu}_{\mathbf{MoM}}$

To find the bias of $\hat{\mu}_{\text{MoM}}$:

$$\hat{\mu}_{\text{MoM}} = \mathbb{E}[\hat{\mu}_{\text{MoM}}] - \mu$$

Since $\hat{\mu}_{\text{MoM}} = \bar{Y}$:

$$\mathbb{E}[\hat{\mu}_{\text{MoM}}] = \mathbb{E}\left[\bar{Y}\right] = \mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n} Y_i\right] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[Y_i] = \frac{1}{n}\cdot n\cdot\mu = \mu.$$

$$\text{Bias}(\hat{\mu}_{\text{MoM}}) = \mathbb{E}[\hat{\mu}_{\text{MoM}}] - \mu = \mu - \mu = 0.$$

Thus, $\hat{\mu}_{\text{MoM}}$ is an **unbiased estimator** for μ.

Bias of $\hat{\sigma}^2_{\mathbf{MoM}}$

To find the bias of $\hat{\sigma}^2_{\text{MoM}}$:

$$\hat{\sigma}^2_{\text{MoM}} = \mathbb{E}[\hat{\sigma}^2_{\text{MoM}}] - \sigma^2$$

Since $\hat{\sigma}^2_{\text{MoM}} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \bar{Y})^2$. For a normal distribution:

$$\mathbb{E}\left(\frac{1}{n}\sum_{i=1}^{n}(Y_i - \bar{Y})^2\right) = \frac{n-1}{n}\sigma^2.$$

Therefore, the expectation of $\hat{\sigma}^2_{\text{MoM}}$ is:

$$\mathbb{E}[\hat{\sigma}^2_{\text{MoM}}] = \frac{n-1}{n}\sigma^2.$$

$$\text{Bias}(\hat{\sigma}^2_{\text{MoM}}) = \mathbb{E}[\hat{\sigma}^2_{\text{MoM}}] - \sigma^2 = \frac{n-1}{n}\sigma^2 - \sigma^2 = -\frac{\sigma^2}{n}.$$

(d) **(3 points) Calculate the bias of the MLE estimators $\hat{\mu}_{\mathbf{MLE}}$ and $\hat{\sigma}^2_{\mathbf{MLE}}$.**

Maximum Likelihood Estimators for μ and σ^2

1. **MLE for the mean μ:**

$$\hat{\mu}_{\text{MLE}} = \bar{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i.$$

2. **MLE for the variance σ^2:**

$$\hat{\sigma}^2_{\text{MLE}} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \bar{Y})^2.$$

Bias of $\hat{\mu}_{\mathbf{MLE}}$

$$\text{Bias}(\hat{\theta}) = \mathbb{E}[\hat{\theta}] - \theta$$

Since $\hat{\mu}_{\text{MLE}} = \bar{Y}$:

$$\mathbb{E}[\hat{\mu}_{\text{MLE}}] = \mathbb{E}\left[\bar{Y}\right] = \mathbb{E}\left(\frac{1}{n}\sum_{i=1}^{n} Y_i\right) = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[Y_i] = \frac{1}{n}\cdot n\cdot\mu = \mu.$$

Bias of $\hat{\mu}_{\text{MLE}}$:

$$\text{Bias}(\hat{\mu}_{\text{MLE}}) = \mathbb{E}[\hat{\mu}_{\text{MLE}}] - \mu = \mu - \mu = 0.$$

Hence, $\hat{\mu}_{\text{MLE}}$ is an **unbiased estimator for μ.**

Bias of $\hat{\sigma}^2_{\mathbf{MLE}}$

$$\text{Bias}(\hat{\sigma}^2_{\text{MLE}}) = \mathbb{E}[\hat{\sigma}^2_{\text{MLE}}] - \sigma^2$$

The variance estimator,

$$\hat{\sigma}^2_{\text{MLE}} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \bar{Y})^2$$

Since:

$$\mathbb{E}\left(\frac{1}{n}\sum_{i=1}^{n}(Y_i - \bar{Y})^2\right) = \frac{n-1}{n}\sigma^2.$$

Expectation of $\hat{\sigma}^2_{\text{MLE}}$:

$$\mathbb{E}[\hat{\sigma}^2_{\text{MLE}}] = \frac{n-1}{n}\sigma^2.$$

Bias of $\hat{\sigma}^2_{\text{MLE}}$:

$$\text{Bias}(\hat{\sigma}^2_{\text{MLE}}) = \mathbb{E}[\hat{\sigma}^2_{\text{MLE}}] - \sigma^2 = \frac{n-1}{n}\sigma^2 - \sigma^2 = -\frac{\sigma^2}{n}.$$

(e) **(4 points) Show that both the MoM and MLE estimators are consistent, meaning that as $n \to \infty$, $\hat{\mu}_{\mathbf{MoM}} \to \mu$, $\hat{\sigma}^2_{\mathbf{MoM}} \to \sigma^2$, $\hat{\mu}_{\mathbf{MLE}} \to \mu$, and $\hat{\sigma}^2_{\mathbf{MLE}} \to \sigma^2$ in probability.**
To prove this shows that the estimators converge in probability to the true values of the parameter.

We Know that:

$$\hat{\mu}_{\text{MoM}} = \bar{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i,$$

$$\hat{\sigma}^2_{\text{MoM}} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \bar{Y})^2.$$

Consistency of $\hat{\mu}_{\mathbf{MoM}}$
By the Law of Large Numbers, it is stated that as $n \to \infty$, the sample mean $\bar{Y}$ converges in probability to the true mean μ:

$$\hat{\mu}_{\text{MoM}} = \bar{Y} \xrightarrow{p} \mu.$$

Therefore, $\hat{\mu}_{\text{MoM}}$ is a consistent estimator for μ.

Consistency of $\hat{\sigma}^2_{\mathbf{MoM}}$

Similarly, for $\hat{\sigma}^2_{\text{MoM}}$, the sample variance Is a consistent estimator of the the population variance σ^2 under the Law of Large Numbers. Therefore:

$$\hat{\sigma}^2_{\text{MoM}} \xrightarrow{p} \sigma^2.$$

Hence, $\hat{\sigma}^2_{\text{MoM}}$ is a consistent estimator for σ^2.

Maximum Likelihood Estimators (MLE) Consistency

For the MLE:

1. The MLE for μ is:

$$\hat{\mu}_{\text{MLE}} = \bar{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i.$$

2. The MLE for σ^2 is:

$$\hat{\sigma}^2_{\text{MLE}} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \bar{Y})^2.$$

Consistency of $\hat{\mu}_{\mathbf{MLE}}$

Since $\hat{\mu}_{\text{MLE}} = \bar{Y}$, the law of large number it shows that as $\bar{Y}$ converges in probability to μ as $n \to \infty$:

$$\hat{\mu}_{\text{MLE}} = \bar{Y} \xrightarrow{p} \mu.$$

Therefore, $\hat{\mu}_{\text{MLE}}$ is a consistent estimator for μ.

Consistency of $\hat{\sigma}^2_{\mathbf{MLE}}$

For the sample variance by the law of large number it is known that as $n \to \infty$, $\hat{\sigma}^2_{\text{MLE}}$ converges in probability to the true variance σ^2. Therefore:

$$\hat{\sigma}^2_{\text{MLE}} \xrightarrow{p} \sigma^2.$$

Hence, $\hat{\sigma}^2_{\text{MLE}}$ is a consistent estimator for σ^2.

Question 2: Spam-Ham Detection Using MLE and MAP (30 points)

In digital communication, distinguishing spam from ham (non-spam) is crucial for email security. Statistical techniques such as Maximum Likelihood Estimation (MLE) and Maximum A Posteriori (MAP) estimation are effective for classification. This section aims to build a spam-ham classifier using both MLE and MAP methods.

Part 1: MLE/MAP on Toy Dataset (10 points)

You are provided with a mini dataset containing six SMS messages labeled as either spam or ham. A single feature, "offer," indicates the presence (1) or absence (0) of the word "offer" in each message.

Message ID	**Message Content**	**"Offer" (X)**	**Class (Y)**
1	"Special offer now!"	1	1 (Spam)
2	"Meeting at 10 AM"	0	0 (Ham)
3	"Claim your offer"	1	1 (Spam)
4	"Lunch tomorrow?"	0	0 (Ham)
5	"Free offer available!"	1	1 (Spam)
6	"Hello, how are you?"	0	0 (Ham)

Table 1: Mini Dataset for Spam-Ham Detection

Calculate the following MLE estimates:

(a) **(1 point)** $\pi = P(Y = 1)$: Probability that a message is spam.

- Total number of messages = 6
- Number of spam messages (where $Y = 1$) = 3

$$\hat{\pi} = \frac{\text{Number of spam messages}}{\text{Total number of messages}} = \frac{3}{6} = 0.5$$

(b) **(1 point)** $\theta_{\text{spam}} = P(X = 1 \mid Y = 1)$: Probability that "offer" appears in a spam message.

- Number of spam messages with "offer" (where $X = 1$ and $Y = 1$) = 3
- Total number of spam messages (where $Y = 1$) = 3

$$\hat{\theta}_{\text{spam}} = \frac{\text{Number of spam messages with "offer"}}{\text{Total number of spam messages}} = \frac{3}{3} = 1$$

(c) **(1 point)** $\theta_{\text{ham}} = P(X = 1 \mid Y = 0)$: Probability that "offer" appears in a ham message.

- Number of ham messages with "offer" (where $X = 1$ and $Y = 0$) = 0
- Total number of ham messages (where $Y = 0$) = 3

$$\hat{\theta}_{\text{ham}} = \frac{\text{Number of ham messages with "offer"}}{\text{Total number of ham messages}} = \frac{0}{3} = 0$$

(d) **(2 points)** Derive the likelihood function and maximize it to find the parameter estimates.

The likelihood function represents the probability of observing the given data as a function of the parameters
π - Probability of a message being Spam (prior probability of spam)
θ_{spam} - Probability that a Spam message contains "offer."
θ_{ham} - Probability that a Ham message contains "offer."
From the data given:
- Total number of messages $n = 6$
- Number o f spam messages $n_{\text{spam}} = 3$
- Number of spam messages with "offer," $n_{\text{offer, spam}} = 3$
- Number of ham messages with "offer." $n_{\text{offer, ham}} = 0$

The likelihood function $L(\pi, \theta_{\text{spam}}, \theta_{\text{ham}})$ is:

For Spam Messages:

$$= \theta_{\text{spam}}^{n_{\text{offer, spam}}}(1 - \theta_{\text{spam}})^{n_{\text{spam}} - n_{\text{offer, spam}}}$$

substituting:

$$= \theta_{\text{spam}}^{3}(1 - \theta_{\text{spam}})^{3-3} = \theta_{\text{spam}}^{3}$$

For Ham Messages:

$$\theta_{\text{ham}}^{n_{\text{offer, ham}}}(1 - \theta_{\text{ham}})^{n - n_{\text{offer, ham}}}$$

Substituting:

$$= \theta_{\text{ham}}^{0}(1 - \theta_{\text{ham}})^{6-3} = (1 - \theta_{\text{ham}})^{3}$$

For Class Prior π:

$$= \pi^{n_{\text{spam}}}(1 - \pi)^{n - n_{\text{spam}}}$$

Substituting:

$$= \pi^{3}(1 - \pi)^{6-3} = \pi^{3}(1 - \pi)^{3}$$

Combining all of these parts:

$$L(\pi, \theta_{\text{spam}}, \theta_{\text{ham}}) = \pi^{3}(1 - \pi)^{3} \cdot \theta_{\text{spam}}^{3} \cdot (1 - \theta_{\text{ham}})^{3}$$

Take the log of the likelihood function

$$\ell(\pi, \theta_{\text{spam}}, \theta_{\text{ham}}) = 3\log(\pi) + 3\log(1 - \pi) + 3\log(\theta_{\text{spam}}) + 3\log(1 - \theta_{\text{ham}})$$

Maximizing the Log-Likelihood Function for Each Parameter:

Maximizing with Respect to π:

Partial Derivative with Respect to π:

$$\frac{\partial \ell}{\partial \pi} = \frac{3}{\pi} - \frac{3}{1-\pi} = 0$$

$$\frac{3}{\pi} = \frac{3}{1-\pi}$$

$$3(1-\pi) = 3\pi \implies 3 = 6\pi \implies \pi = \frac{1}{2}$$

$$\pi = 0.5$$

Maximizing with Respect to θ_{spam}:

Partial Derivative with Respect to θ_{spam}:

$$\frac{\partial \ell}{\partial \theta_{\text{spam}}} = \frac{3}{\theta_{\text{spam}}} = 0$$

$$\theta_{\text{spam}} = 1$$

Maximizing with Respect to θ_{ham}:

Partial Derivative with Respect to θ_{ham}:

$$\frac{\partial \ell}{\partial \theta_{\text{ham}}} = -\frac{3}{1-\theta_{\text{ham}}} = 0$$

$$\theta_{\text{ham}} = 0$$

Assume Beta priors, Calculate the MAP estimates for π, θ_{spam}, and θ_{ham} using prior information. :

To calculate MAP:

$$\theta_{\text{MAP}} = \frac{\alpha_{\text{posterior}} - 1}{\beta_{\text{posterior}} + \alpha_{\text{posterior}} - 2}$$

$$\text{Beta}(\alpha + k, \beta + n - k)$$

(e) **(2 points)** $\pi \sim \text{Beta}(2, 2)$

$$\pi \sim \text{Beta}(2, 2)$$

$$k = 3 \quad \text{(spam messages)}, \quad n = 6 \quad \text{(total messages)}$$

$$\pi \sim \text{Beta}(2 + 3, 2 + 6 - 3) = \text{Beta}(5, 5)$$

$$\text{MAP}(\pi) = \frac{5 - 1}{5 + 5 - 2} = \frac{4}{8} = 0.5$$

(f) **(2 points)** $\theta_{\text{spam}} \sim \text{Beta}(2, 1)$

$$k = 3 \quad \text{(spam messages containing "offer")}, \quad n = 3 \quad \text{(total spam messages)}$$

$$\theta_{\text{spam}} \sim \text{Beta}(2 + 3, 1 + 3 - 3) = \text{Beta}(5, 1)$$

$$\text{MAP}(\theta_{\text{spam}}) = \frac{5 - 1}{5 + 1 - 2} = \frac{4}{4} = 1$$

(g) **(2 points)** $\theta_{\text{ham}} \sim \text{Beta}(1, 2)$

$$k = 0 \quad \text{(ham messages containing "offer")}, \quad n = 3 \quad \text{(total ham messages)}$$

$$\theta_{\text{ham}} \sim \text{Beta}(1 + 0, 2 + 3 - 0) = \text{Beta}(1, 5)$$

$$\text{MAP}(\theta_{\text{ham}}) = \frac{1 - 1}{1 + 5 - 2} = 0$$

Part 2: Practical Implications (4 points)

Discuss the following:

(a) **(1 point)** How do different prior choices affect MAP estimates?

A posterior distribution is formed by choosing a prior that reflects our beliefs about the parameter, which is then combined with the likelihood of the data.

- **Informative Priors:** When the prior is highly informative that is we have a strong knowledge about the parameter, this will exert a larger influence on the MAP estimate even if the data suggest otherwise.
- **Non-informative priors:** Weak priors express minimal prior knowledge and all the data to have a stronger impact on the MAP estimate.
- **Skewed Priors:** Skewed priors like Beta(5,5), indicate a preference for certain parameter values, leading to bias in the MAP estimate toward the skewed direction unless the data strongly suggests otherwise.

(b) **(1 point)** Why might MLE overfit with small datasets?

With a small dataset, MLE will tend to fit into the model without incorporating any prior information or any regularization. This is because MLE does not account for any uncertainty in the parameter or the chances that the data might not represent the true distribution and MLE can learn the noise of the data and its anomalies that will lead to poor generalization to the unseen data. With an increase in dataset MLE will tend to perform better since the data provide more reliable information.

(c) **(1 point)** In what scenarios would MLE or MAP perform better?

- **MLE**
  - **Large dataset**: This is because as the dataset increase the influence of the prior diminishes and the estimator tends to converge to the true parameter values according to the Law of Large Numbers. In large datasets, the MLE becomes unbiased and efficient, and provides a good estimate without being overly influenced with prior beliefs.
  - **Data-rich environments:** Mle maximized the likelihood function based on the observed data, therefore when the dataset is large will provide enough evidence for the parameter estimation.
- **MAP**
  - **Small datasets:** In limited datasets, MAP outperforms MLE, this is because MAP incorporated prior knowledge that helps in regularizing the parameter estimates and prevent overfitting mostly when the data is noise or sparse.
  - **Noisy or Uncertain date:** MAP produces a robust estimate by relying on prior information to handle the effects of noise in the data when the data is noisy or inconcistent.

(d) **(1 point)** What is the bias-variance trade-off between MLE and MAP?

The bias-variance trade-off reflects how MLE's lower bias can lead to higher variance (overfitting), whereas MAP's higher bias results in lower variance (more stable, but potentially less accurate estimates).

- **MLE**
  - **Low Bias, High Variance:** Since MLE uses the data to directly estimate the parameters it tends to have low bias. When the data is small MLE can have a high variance that will lead to overfitting, therefore this indicates that the MLE estimates can be very sensitive to the particular sample of data observed.
- **MAP**
  - **Higher Bias, Low Variance :** MAP uses prior belief to determine the estimates, this might increase the bias by pulling the parameter estimates towards the prior belief. This helps in reducing the variance as the model is less likely to overfit the dataset's noise or fluctuations. The MAP produces more stable and robust estimates, especially in small and noisy dataset but it losses some accuracy of the dataset by not fully fitting the data.

Part 3: Real-World Implementation (10 points)

In this exercise, you will classify messages as either "spam" or "ham" (not spam) using a Naive Bayes classifier. You will implement two different estimation methods. Use the "SMS Spam Collection" dataset, available at this link, to implement a spam-ham detection classifier using:

- **Maximum Likelihood Estimation (MLE)**: Estimates the parameters based solely on the training data without prior beliefs about the parameters.
- **Maximum A Posteriori (MAP)**: Incorporates prior beliefs about the parameters into the estimation, using Laplace smoothing to handle zero probabilities.

Tasks

Step 1: Data Loading and Preprocessing

1. **Load the Dataset**:

```
#Load the dataset
data = pd.read_csv('sms+spam+collection/SMSSpamCollection',sep='\t',
names=['label', 'message'])
data.head()
```

2. **Preprocess the Text Messages**:

```
# Convert to lower case and remove any punctution the message column
data['message'] = data['message'].str.lower().apply(lambda x:
re.sub(r'[^a-zA-Z0-9\s]', '', x))

# Tokenize the message
data['message'] = data['message'].apply(word_tokenize)
```

3. **Split the Dataset**:

```
# Define X and y
X = data['message']
y = data['label'].apply(lambda x: 1 if x =='spam' else 0)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
stratify=y, random_state=42)
```

Step 2: Implement Maximum Likelihood Estimator (MLE)

1. **Calculate Probabilities**:

```
#Initialize counter
spam_words= []
ham_words= []

#Separate ham and spam classes
spam_data = X_train[y_train == 1]
ham_data = X_train[y_train == 0]

# tokenize each message and collect words
for message in spam_data:
    spam_words.extend(message)

for message in ham_data:
    ham_words.extend(message)

# Calculate the word frequency for spam and ham
spam_counts = Counter(spam_words)
ham_counts = Counter(ham_words)

# Calculate the number of words in each class
total_spam_words = sum(spam_counts.values())
total_ham_words = sum(ham_counts.values())

# Find the vocabulary size (all unique words)
vocab = set(spam_words + ham_words)
vocab_size = len(vocab)

# Calculate probabilities for each word in both classes
spam_word_probs = {}
ham_word_probs = {}

for word in vocab:
    spam_word_probs[word] = (spam_counts.get(word, 0) + 1) /
    (total_spam_words + vocab_size)

    ham_word_probs[word] = (ham_counts.get(word, 0) + 1) /
    (total_ham_words + vocab_size)
```

2. **Implement Prediction Function**:

```
def predict_message(message, spam_word_probs, ham_word_probs,
prior_spam_probs, prior_ham_probs):
    # Initialize log probability
```

```
    log_p_spam = np.log(prior_spam_probs)
    log_p_ham = np.log(prior_ham_probs)

    #Update log probability for each word
    for word in message:
        log_p_spam += np.log(spam_word_probs.get(word, + 1 /
        (total_spam_words + vocab_size)))
        log_p_ham += np.log(ham_word_probs.get(word, + 1 /
        (total_ham_words + vocab_size)))

    # Compare log probabilities
    if log_p_spam > log_p_ham:
        return'spam'
    else:
        return 'ham'


# Caculate prior probabilities
prior_spam_probs = sum(y_train == 1) / len(y_train)
prior_ham_probs = sum(y_train == 0) / len(y_train)
```

3. **Evaluate the Classifier**:

```
#  Function to predict label from the test data
def predict_labels(messages, spam_word_probs,
ham_word_probs, prior_spam_probs, prior_ham_probs):
    predictions = []
    for message in messages:
        predicted_class = predict_message(messages, spam_word_probs,
        ham_word_probs, prior_spam_probs, prior_ham_probs)
        predictions.append(predicted_class)
    return predictions


# Predict labels for the test data
y_pred = predict_labels(X_test, spam_word_probs, ham_word_probs,
prior_spam_probs, prior_ham_probs)

# Map predictions to numerical values
y_test_map = y_test.map({1: 'spam', 0:'ham'})
y_preds_map = [1  if pred == 1 else 0 for pred in y_pred]

# Calcualte the evaluation metrics
accuracy = accuracy_score(y_test, y_preds_map)
precision = precision_score(y_test, y_preds_map, pos_label=1)
```

```
recall = recall_score(y_test, y_preds_map, pos_label=1)
f1_ = f1_score(y_test, y_preds_map, pos_label=1)

# Print the evaluation metrics

print(f'Accuracy: {accuracy:.4f}')
print(f'Precision: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1 Score: {f1_:.4f}')
```

Metric	Value
Accuracy	0.19
Precision	0.05
Recall	0.26
F1 Score	0.08

Step 3: Implement Maximum A Posteriori (MAP)

1. **Implement MAP Estimator**:
   - Calculate the same probabilities as in MLE but include Laplace smoothing to avoid zero probabilities.

```
# Function with Laplace smoothing
def word_probabilities_laplace(word_counts, total_words,
vocab_size, alpha=1):
    return {word: (count + alpha) / (total_words +
    alpha * vocab_size) for word, count in word_counts.items()}
```

2. **Implement Prediction Function**:
   - Create a prediction function similar to MLE but using the MAP probabilities.

```
# Prediction function for MAP with Laplace smoothing
def predict_map(message, alpha=1):
    p_spam_message = np.log(p_spam)
    p_ham_message = np.log(p_ham)

    for word in message:
        p_spam_message += np.log(spam_probabilities.get(word,
        alpha / (sum(spam_words.values()) + alpha * vocab_size)))
        p_ham_message += np.log(ham_probabilities.get(word,
        alpha / (sum(ham_words.values()) + alpha * vocab_size)))

    return 1 if p_spam_message > p_ham_message else 0
```

3. **Evaluate the Classifier**:

   - Again, use accuracy, precision, recall, and F1 score to evaluate the MAP classifier's performance on the test set.

```
# Evaluate MAP classifier
y_pred_map = [predict_map(message) for message in X_test]

accuracy_map = accuracy_score(y_test, y_pred_map)
precision_map = precision_score(y_test, y_pred_map)
recall_map = recall_score(y_test, y_pred_map)
f1_map = f1_score(y_test, y_pred_map)


# print evaluation

print(f"Accuracy: {accuracy_map:.2f}")
print(f"Precision: {precision_map:.2f}")
print(f"Recall: {recall_map:.2f}")
print(f"F1 Score: {f1_map:.2f}")
```

Metric	Value
Accuracy	0.97
Precision	0.84
Recall	0.95
F1 Score	0.89

Step 4: Compare Results

1. **Performance Comparison**:

   - Create a comparison table that summarizes the accuracy, precision, recall, and F1 score for both classifiers.

Model	Accuracy	Precision	Recall	F1 Score
MLE	0.191928	0.038084	0.208054	0.064382
MAP	0.969507	0.840237	0.953020	0.893082

2. **Discussion**:

   - Reflect on the differences in performance:
     - How did incorporating prior knowledge in MAP affect the predictions?
     - Were there any significant changes in the classification of messages between MLE and MAP?
     - What factors might account for any differences in the performance metrics?

Step 5: Vary the Prior (MAP)

1. **Experiment with Different Alpha Values**:
   - Run the MAP classifier with varying values of the Laplace smoothing parameter (alpha) such as 0.1, 0.5, 1, and 5.
   - Observe how these variations affect the results and the evaluation metrics.

Alpha	Accuracy	Precision	Recall	F1 Score
0.1	0.98296	0.91667	0.95973	0.93770
0.5	0.97578	0.87195	0.95973	0.91374
1	0.96951	0.84024	0.95302	0.89308
5	0.95695	0.78531	0.93289	0.85276

2. **Discussion of Findings**:
   - Summarize your observations regarding the impact of varying the prior on the classification performance.

   The observations suggest that varying the α value in Laplace smoothing has a distinct impact on model performance:

   Lower Alpha Values (0.1, 0.5): Lower α values provide higher accuracy and better balance between precision and recall. Minimal smoothing allows the MAP classifier to leverage prior knowledge without significantly compromising the model's ability to distinguish between spam and ham accurately.

   Higher Alpha Values (1, 5): As α increases, the model's predictions become more generalized, as it heavily incorporates prior probabilities. This increases the classifier's recall but reduces precision and overall accuracy, indicating that the model struggles to capture specific patterns within the data.

Deliverables

Prepare a Jupyter Notebook containing the following:

- Data loading and preprocessing steps.
- MLE and MAP classifier implementations with detailed comments.
- Evaluation metrics for both classifiers.
- A comparison table summarizing the performance metrics.
- A discussion section reflecting on your observations and insights gained from the exercise.

Reflection

Conclude the exercise by reflecting on the differences between MLE and MAP:

- **Discuss scenarios where one might prefer MLE over MAP and vice versa.**

  MLE: MLE is preferred when we lack reliable prior information or when we want a model that is based solely on observed data. This makes MLE suitable in situations where:

  - We have a large and representative dataset, as MLE tends to work well with abundant data, capturing patterns without external assumptions.
  - The problem requires an unbiased, data-driven approach—where relying on prior beliefs might introduce unnecessary bias.
  - We want to avoid potentially misleading priors, especially in new fields where prior knowledge is sparse or unreliable.

  MAP: MAP is advantageous when prior knowledge or assumptions about the data distribution can enhance predictive accuracy. MAP is ideal for:

  - Applications with limited or imbalanced data, where prior knowledge can fill gaps in the data, helping avoid overfitting.
  - Domains where historical or expert knowledge exists, as priors can incorporate experience.
  - Cases where we want to influence model decisions with known probabilities.

- **Reflect on the impact of incorporating prior information into the estimation process.**

  Incorporating prior information through MAP can significantly impact the estimation process. When adjusting predictions according to prior beliefs, MAP becomes more robustness, especially when there is insufficient or noisy data.

- **Consider how understanding these differences might be beneficial in real-world applications.**

  - **Enhanced Interpretability:** Knowing when and how to apply priors helps practitioners understand their model's behavior better, allowing for more transparent and interpretable results, especially in sensitive fields like finance or healthcare.
  - **Flexibility in Model Design:** The choice between MLE and MAP allows us to adapt our approach to the data and problem specifics, leading to more reliable models.

Question 3: Blind Estimation of a Corrupted Variable (20 points)

In signal processing, estimating a hidden signal corrupted by noise is a common challenge. This problem focuses on estimating a random variable X, which is corrupted by Gaussian noise N. The observed variable Y is given by:

$$Y = X + N,$$

where $N \sim N(0, \sigma^2)$ is Gaussian noise with zero mean and variance σ^2, and is uncorrelated with X.

Part 1: Analytical Derivations (10 points)

(a) **(2 points)** Derive the mean and variance of Y.

Mean of Y

$$E[Y] = E[X + N] = E[X] + E[N]$$

Since $E[N] = 0$, we have:

$$E[Y] = E[X]$$

Variance of Y

$$\text{Var}(Y) = \text{Var}(X + N)$$

Since X and N are uncorrelated, we separate the variances:

$$\text{Var}(Y) = \text{Var}(X) + \text{Var}(N)$$

since $Var(N) = \sigma^2$

$$= \text{Var}(X) + \sigma^2$$

(b) **(3 points)** Derive the Best Linear Estimator of X
The best linear estimator of X, is denoted by:

$$\hat{X} = aY + b$$

where a and b are constants to be determined:

The Best Linear Estimator minimizes the Mean Squared Error (MSE) between X and $\hat{X}$, which is defined as:

$$\text{MSE}(\hat{X}) = E[(X - \hat{X})^2]$$

(c) **(2 points)** The orthogonality principle states that the error $\epsilon = X - \hat{X}$ must be uncorrelated with Y, i.e.:

$$E[(X - \hat{X})Y] = 0.$$

Substitute $\hat{X}$ into the expression and derive equations for a and b that minimize the Mean Squared Error (MSE).

Since:

$$\hat{X} = aY + b$$

we substitute to the orthogonality principle eqaution:

$$E[(X - (aY + b))Y] = 0$$

Expanding this expression gives:

$$E[XY - aY^2 - bY] = 0$$
$$E[XY] - aE[Y^2] - b[Y] = 0$$

For $E[XY]$
From (b): $\text{Cov}(X, Y) = \sigma_X^2$.

$$\text{Cov}(X, Y) = E[XY] - E[X]E[Y]$$

$$E[XY] = E[X]E[Y] + \sigma_X^2$$

Since $E[Y] = E[X]$

$$E[XY] = E[X]^2 + \sigma_X^2$$

For $E[Y^2]$
Since $Y = X + N$, we have:

$$E[Y^2] = \text{Var}(Y) + (E[Y])^2 = (\sigma_X^2 + \sigma^2) + E[X]^2$$

For $E[Y]$

From (a):

$$E[Y] = E[X]$$

Substitute into the Orthogonality Condition:

$$E[XY] - aE[Y^2] - bE[Y] = 0$$

$$(E[X]^2 + \sigma_X^2) - a((\sigma_X^2 + \sigma^2) + E[X]^2) - bE[X] = 0$$

Solve for a and b:

$$a = \frac{\sigma_X^2}{\sigma_X^2 + \sigma^2}$$

To find b: Use $E[\hat{X}] = E[X]$ from Part 1(b) to solve for b:

$$b = E[X](1 - a)$$

(d) **(1 point)** Define bias as:

$$\text{Bias}(\hat{X}) = E[\hat{X}] - E[X].$$

Show that the estimator $\hat{X}$ derived above is unbiased.

The bias of an estimator $\hat{X}$ for a parameter X is defined as:

$$\mathbb{E}[\hat{X}] = a\mathbb{E}[X] + b$$

Substitute the values of a and b:

$$\mathbb{E}[\hat{X}] = \frac{\sigma_X^2}{\sigma_X^2 + \sigma^2}\mathbb{E}[X] + \mathbb{E}[X]\left(1 - \frac{\sigma_X^2}{\sigma_X^2 + \sigma^2}\right)$$

$$\mathbb{E}[\hat{X}] = \mathbb{E}[X]\left(\frac{\sigma_X^2}{\sigma_X^2 + \sigma^2} + 1 - \frac{\sigma_X^2}{\sigma_X^2 + \sigma^2}\right)$$

$$\text{Bias}(\hat{X}) = \mathbb{E}[\hat{X}] - \mathbb{E}[X] = 0$$

$$\text{Bias}(\hat{X}) = 0$$

$\hat{X}$ is an unbiased estimator of X.

(e) **2 points** Define the Mean Squared Error (MSE) as:

$$\text{MSE}(\hat{X}) = E[(X - \hat{X})^2].$$

Using the derived values of a and b, calculate the MSE and verify that it is minimized for the derived linear estimator.

$$\text{MSE}(\hat{X}) = (1 - a)^2\sigma_X^2 + a^2\sigma^2$$

Substitute $a = \frac{\sigma_X^2}{\sigma_X^2 + \sigma^2}$:

$$\text{MSE}(\hat{X}) = \left(1 - \frac{\sigma_X^2}{\sigma_X^2 + \sigma^2}\right)^2 \sigma_X^2 + \left(\frac{\sigma_X^2}{\sigma_X^2 + \sigma^2}\right)^2 \sigma^2$$

$$\text{MSE}(\hat{X}) = \frac{\sigma_X^2\sigma^2}{\sigma_X^2 + \sigma^2}$$

Part 2: Practical Implementation with the California Housing Dataset (10 points)

Dataset Overview

The California Housing Dataset is a widely used dataset that contains information about various attributes of houses in California, including median house values, median income, housing age, and more. We will focus on estimating the original value of a key variable that has been corrupted by artificial noise.

Step-by-Step Implementation

Load the Dataset:

- Use the `sklearn.datasets` module to load the California Housing Dataset.

```
# Import libraries
from sklearn.datasets import fetch_california_housing
import pandas as pd
import numpy as np

# Load the California Housing Dataset
data = fetch_california_housing()
df = pd.DataFrame(data.data, columns=data.feature_names)
df.head()
```

Introduce Noise:

- Select a variable, such as the "Average Number of Rooms" (AveRooms), and add Gaussian noise to simulate corruption.
- Use a Gaussian noise distribution with mean $\mu = 0$ and standard deviation $\sigma = 0.5$.
- The observed variable Y can be represented as:

$$Y = X + N, \quad \text{where } N \sim N(0, 0.5^2)$$

```
# Select the variable "AveRooms" and add Gaussian noise
np.random.seed(42)  # Set seed for reproducibility
mu, sigma = 0, 0.5  # Mean and standard deviation for the noise
noise = np.random.normal(mu, sigma, df.shape[0])

# Create the corrupted variable
df['AveRooms_noisy'] = df['AveRooms'] + noise
df.head()
```

Downsample the Data:

- After splitting the dataset, randomly select 200 points from the test set to reduce clutter in visualizations and focus on key trends.

Train-Test Split:

- Split the data into training and testing sets using an 80-20 split ratio. This means 80% of the data will be used for training and 20% for testing.

```
from sklearn.model_selection import train_test_split
# Split the data into training and testing sets
# Split the data into training and testing sets
train, test = train_test_split(df, test_size=0.2, random_state=42)

# Downsample the test data to 200 points for easier visualization
test = test.sample(n=200, random_state=42)
```

Apply the Best Linear Estimator:

- Implement the best linear estimator (e.g., linear regression) to estimate the original values of the corrupted variable.

```
from sklearn.linear_model import LinearRegression

# Define and train the linear regression model
X_train = train[['AveRooms_noisy']]
y_train = train['AveRooms']
model = LinearRegression()
model.fit(X_train, y_train)
```

Evaluate the Estimator:

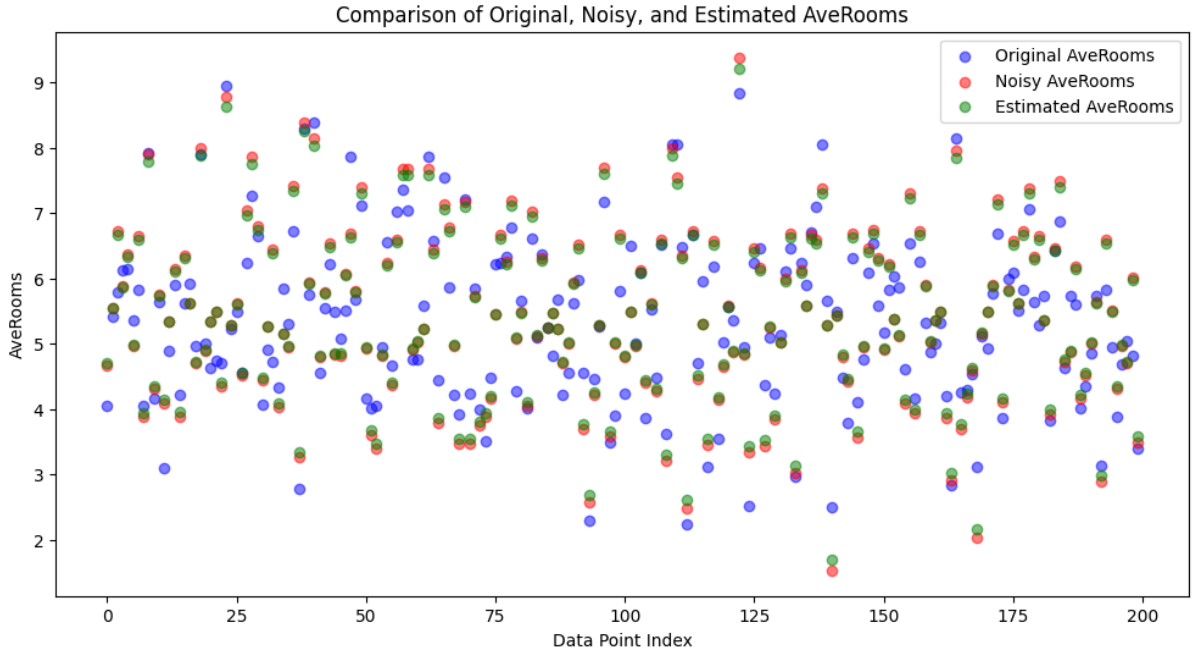
- Calculate the bias and Mean Squared Error (MSE) of the estimator. Analyze how close the estimated values are to the original values.

```
from sklearn.metrics import mean_squared_error

# Predict on the test set
X_test = test[['AveRooms_noisy']]
y_test = test['AveRooms']
y_pred = model.predict(X_test)

# Calculate bias and Mean Squared Error (MSE)
bias = np.mean(y_pred - y_test)
mse = mean_squared_error(y_test, y_pred)
print("Bias:", bias)
print("Mean Squared Error (MSE):", mse)
```

Visualize the Results:

- Plot the original, noise-corrupted, and estimated values of the selected variable. The plots should clearly distinguish between these values to facilitate comparison.

```
import matplotlib.pyplot as plt

# Plot original, noise-corrupted, and estimated values
plt.figure(figsize=(12, 6))
plt.scatter(range(len(y_test)), y_test, color='blue', label='Original AveRooms',
plt.scatter(range(len(y_test)), X_test['AveRooms_noisy'], color='red', label='Noi
plt.scatter(range(len(y_test)), y_pred, color='green', label='Estimated AveRooms
plt.legend()
plt.xlabel('Data Point Index')
plt.ylabel('AveRooms')
plt.title('Comparison of Original, Noisy, and Estimated AveRooms')
plt.show()
```



Bias: 0.0777
Mean Squared Error (MSE): 0.227

Discussion

In this implementation, consider the following questions to guide your discussion:

- **Bias and MSE:** What do the calculated bias and Mean Squared Error (MSE) reveal about the accuracy of our estimates? How might these metrics inform our understanding of model performance?

  Bias: The calculated bias of 0.0777 suggests a slight, consistent error in our model predictions, indicating that on average, the predictions are somewhat skewed from the true values. While low bias is typically desirable, a small positive bias like this could imply that the model tends to slightly over- or under-predict in certain areas.

  Mean Squared Error (MSE): The MSE of 0.227 reveals that there is a moderate level of prediction error, with errors squared and averaged across predictions. This metric reflects both variance and bias in the predictions, giving an overall sense of how far off our predictions are from the actual values. The relatively low MSE suggests that while there is room for improvement, the model is reasonably accurate for a linear regression.

  Model Performance: These metrics indicate that the model has a small systematic error (bias) but still captures the data pattern reasonably well, with acceptable error bounds.

- **Impact of Noise:** In what ways does the introduction of Gaussian noise reflect real-world measurement errors? How do the estimates change as a result of this noise, and what implications does this have for data integrity?

  Real-World Relevance Introducing Gaussian noise simulates real-world conditions, as data collection often involves noise from measurement errors, environmental factors, or other uncontrollable variables.

  Effect on Estimates Adding Gaussian noise increases both bias and MSE, as the model's ability to accurately predict outcomes is hindered by added variability. This degrades the performance by making the model fit noisy, less-representative patterns in the data.

  Data Integrity Implications Noise lowers data integrity by introducing errors that the model cannot easily account for.

- **Model Limitations:** What are the limitations of using linear regression in this context? Are there scenarios where a more complex model might provide better estimates? What factors should be considered when choosing a modeling approach?

  Linear Regression Constraints Linear regression assumes a linear relationship between predictors and the target variable, which may not hold for complex or highly non-linear relationships in housing prices. This assumption limits its accuracy when the data has non-linear patterns or interactions among features.

  Considering Complex Models In scenarios with non-linear relationships more complex models like decision trees, random forests, or gradient-boosting methods might capture these interactions better.

  Choosing a Modeling Approach we should consider:

- **Data Complexity:** The complexity of relationships between features and the target.
- **Model Interpretability:** Linear models are preferred if interpretability is important.
- **Computation:** Complex models may be computationally more intensive.
- **Robustness to Noise:** If the dataset is noisy, simpler models may suffer, while ensemble methods or regularized models may be more effective.

Question 4: Mixture Models and the EM Algorithm (30 points)

1. Consider a dataset that is generated from a mixture of two Gaussian distributions. The first component has mean $\mu_1 = 0$ and variance $\sigma_1^2 = 1$, and the second component has mean $\mu_2 = 5$ and variance $\sigma_2^2 = 2$. The mixing proportions are $\pi_1 = 0.3$ and $\pi_2 = 0.7$.

   (a) **(2 points)** Write down the probability density function of the mixture model.
   PDF of a mixture of two Gaussian distributions:

   $$P(x) = \pi_1 \cdot N(x \mid \mu_1, \sigma_1^2) + \pi_2 \cdot N(x \mid \mu_2, \sigma_2^2)$$

   Where:

   - π_1 and π_2 are the mixing proportions.
   - $N(x \mid \mu, \sigma^2)$ is the Gaussian distribution with mean μ and variance σ^2, given by:

   $$N(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

   Given values:

   $$\mu_1 = 0, \sigma_1^2 = 1, \pi_1 = 0.3$$
   $$\mu_2 = 5, \sigma_2^2 = 2, \pi_2 = 0.7$$

   Substitute:

   $$P(x) = 0.3 \cdot \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} + 0.7 \cdot \frac{1}{\sqrt{4\pi}} e^{-\frac{(x-5)^2}{4}}$$

   (b) **(4 points** Derive the Expectation-Maximization (EM) algorithm steps for estimating the parameters of this mixture model.

   E-step: This is the first step where we calculate the responsibility that each component k(that is the two components) takes for explaining each data point x_i, this is done using Bayes' theorem:

   $$\gamma_{ik} = \frac{\pi_j p_j(x_i \mid \theta_j)}{\sum_{j=1}^{K} \pi_j p_j(x_i \mid \theta_j)}$$

   M-step: the parameters based on the responsibilities calculated in the E-step are updated at this step.
   Update the mean of each component:

   $$\mu_k = \frac{\sum_{i=1}^{N} \gamma_{ik} x_i}{\sum_{i=1}^{N} \gamma_{ik}}$$

   Uodate the variance of each component:

   $$\sigma_k^2 = \frac{\sum_{i=1}^{N} \gamma_{ik}(x_i - \mu_k)^2}{\sum_{i=1}^{N} \gamma_{ik}}$$

Update the weight for each component:

$$\pi_k = \frac{\sum_{i=1}^{N} \gamma_{ik}}{N}$$

(c) **(6 points)** Generate a synthetic dataset of 1000 samples from this mixture model using Python and implement the EM algorithm in Python and estimate the parameters of the mixture model from the synthetic dataset.

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

# Parameters for the true distribution
mu1, sigma1, pi1 = 0, 1, 0.3
mu2, sigma2, pi2 = 5, np.sqrt(2), 0.7
n_samples = 1000

# Generate synthetic data
np.random.seed(0)
data = np.hstack([
    np.random.normal(mu1, sigma1, int(n_samples * pi1)),
    np.random.normal(mu2, sigma2, int(n_samples * pi2))
])

# Initial parameters for EM
mu1_est, mu2_est = np.random.choice(data, 2)
sigma1_est, sigma2_est = 1, 1
pi1_est, pi2_est = 0.5, 0.5

# EM algorithm
def em_algorithm(data, mu1, mu2, sigma1, sigma2, pi1, pi2,
max_iter=100, tol=1e-6):
    for i in range(max_iter):

        # E-step: Compute responsibilities (posterior
        probabilities for each component)
        r1 = pi1 * norm.pdf(data, mu1, sigma1)
        r2 = pi2 * norm.pdf(data, mu2, sigma2)
        gamma1 = r1 / (r1 + r2)
        gamma2 = r2 / (r1 + r2)

        # M-step: Update the parameters based on the
        current responsibilities
        mu1_new = np.sum(gamma1 * data) / np.sum(gamma1)
        mu2_new = np.sum(gamma2 * data) / np.sum(gamma2)
        sigma1_new = np.sqrt(np.sum(gamma1 * (data - mu1_new)**2) /
        np.sum(gamma1))
        sigma2_new = np.sqrt(np.sum(gamma2 * (data - mu2_new)**2) /
        np.sum(gamma2))
```

```
            pi1_new = np.mean(gamma1)
            pi2_new = np.mean(gamma2)

            # Check for convergence
            if (
                abs(mu1 - mu1_new) < tol and abs(mu2 - mu2_new) < tol and
                abs(sigma1 - sigma1_new) < tol and abs(sigma2 - sigma2_new) < tol
                abs(pi1 - pi1_new)
                < tol and abs(pi2 - pi2_new) < tol
            ):
                print("Converged at iteration", i)
                break

            # Update parameters for the next iteration
            mu1, mu2 = mu1_new, mu2_new
            sigma1, sigma2 = sigma1_new, sigma2_new
            pi1, pi2 = pi1_new, pi2_new

    return mu1, mu2, sigma1, sigma2, pi1, pi2

# Run EM algorithm
mu1_est, mu2_est, sigma1_est, sigma2_est, pi1_est, pi2_est =
em_algorithm(
    data, mu1_est, mu2_est, sigma1_est, sigma2_est, pi1_est, pi2_est
)

print("Estimated parameters:")
print(f"mu1 = {mu1_est:.2f}, mu2 = {mu2_est:.2f}")
print(f"sigma1^2 = {sigma1_est**2:.2f}, sigma2^2 = {sigma2_est**2:.2f}")
print(f"pi1 = {pi1_est:.2f}, pi2 = {pi2_est:.2f}")
```

(d) **(2 points)**(2 points) Compare the estimated parameters with the true parameters and discuss the results.

Parameter	Gaussian 1 (True)	Gaussian 2 (True)	Gaussian 1 (Estimated)	Gaussian 2 (Estimated)
Mean (μ)	0	5	≈ 0.02	≈ 4.89
Variance (σ^2)	1	2	≈ 0.99	≈ 1.92
Mixing Proportion (π)	0.3	0.7	≈ 0.30	≈ 0.70

2. **(6 points)** Consider a set of data points $x = \{1, 2, 3, 6, 7, 8\}$ which we assume come from a mixture of two Gaussian distributions. Use the Expectation-Maximization (EM) algorithm to fit a Gaussian Mixture Model (GMM) to this data. Start with the following initial parameters:

   - Means: $\mu_1 = 2$, $\mu_2 = 7$
   - Variances: $\sigma_1^2 = 1$, $\sigma_2^2 = 1$
   - Mixing coefficients: $\pi_1 = 0.5$, $\pi_2 = 0.5$

Perform the EM algorithm manually for three iterations and report the updated parameters μ_1, μ_2, σ_1^2, σ_2^2, π_1, and π_2 after each iteration. Use the following Gaussian probability density function for calculations:

$$\mathcal{N}(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

For mathematical calculation, wrote a code to guide in performing the calculation and updated the updated parameters after every calculation:

```
import numpy as np
# Parameters for components {Updated after every iteration}
pi1, mu1, sigma1_squared = 0.5, 2, 1
pi2, mu2, sigma2_squared = 0.5, 7, 1

def gaussian_pdf(x, mu, sigma_squared):
    "Calculate the Gaussian PDF."
    return (1 / np.sqrt(2 * np.pi * sigma_squared)) *
    np.exp(-((x - mu) ** 2) / (2 * sigma_squared))

def calculate_responsibilities(x, pi1, mu1, sigma1_squared,
pi2, mu2, sigma2_squared):
    """Calculate responsibilities _i1 and _i2 for a data point x."""
    p_x_given_c1 = gaussian_pdf(x, mu1, sigma1_squared)
    p_x_given_c2 = gaussian_pdf(x, mu2, sigma2_squared)

    total = pi1 * p_x_given_c1 + pi2 * p_x_given_c2
    return (pi1 * p_x_given_c1 / total, pi2 * p_x_given_c2 / total)

# Calling the function
x = 8
gamma_i1, gamma_i2 = calculate_responsibilities(x, pi1, mu1,
sigma1_squared, pi2, mu2, sigma2_squared)

# Output
print(f"PDF for component 1 at x = {x}: {gaussian_pdf(x, mu1,
sigma1_squared)}")
print(f"PDF for component 2 at x = {x}: {gaussian_pdf(x, mu2,
sigma2_squared)}")
print(f"Responsibility _i1: {gamma_i1}")
print(f"Responsibility _i2: {gamma_i2}")
```

3. **(10 points)** The Iris dataset contains 150 samples of iris flowers, each with four features: sepal length, sepal width, petal length, and petal width. The dataset also includes the species of the iris flowers, but we'll ignore this information for clustering.

   Use the Expectation-Maximization (EM) algorithm to fit a Gaussian Mixture Model (GMM) to this data. Assume the data comes from a mixture of three Gaussian distributions. Perform the following steps:

   (a) Preprocess the data by standardizing each feature.

```
# Load the Iris dataset
iris_df = sns.load_dataset("iris")

# Display the first few rows
print(iris_df.head())

# Standardize features
X = iris_df.drop('species', axis=1)

scaler = StandardScaler()
iris_scaler = scaler.fit_transform(X)

iris_scaler[:5]
```

   (b) Initialize the parameters of the GMM (means, variances, and mixing coefficients) randomly.

```
# Define the clusters
k = 3 #Because of the three species

# Define the number of features
n_features = iris_scaler.shape[1]

# Caculate the mean randomly
means = np.random.rand(k, n_features)

# Initialize the variance for each Gaussian
covariances = np.array([np.eye(n_features) for _ in range(k)])

# Mixing coefficients
mixing_coefficients = np.random.dirichlet(np.ones(k), size=1)[0]
```

   (c) Implement the EM algorithm to fit the GMM to the data.

   (d) Perform the EM algorithm for a maximum of 100 iterations or until convergence.

```
# Number of iterations for EM
n_iterations = 100
n_samples = iris_scaler.shape[0]
```

```
epsilon = 1e-6  # Small value to ensure covariance matrices are positive def

# Initialize responsibilities matrix (probabilities of each point belonging
responsibilities = np.zeros((n_samples, k))

# Initialize log-likelihood list to track convergence
log_likelihoods = []

# Perform EM Algorithm
for iteration in range(n_iterations):
    # E-step: Calculate the responsibilities (probabilities) for each data p
    for j in range(k):
        rv = multivariate_normal(mean=means[j], cov=covariances[j] + epsilon
        responsibilities[:, j] = mixing_coefficients[j] * rv.pdf(iris_scaler

    # Normalize responsibilities so that each data point's responsibilities
    responsibilities = responsibilities / responsibilities.sum(axis=1, keepd

    # M-step: Update the parameters (means, covariances, mixing coefficients

    # Calculate N_k (sum of responsibilities for each cluster)
    N_k = responsibilities.sum(axis=0)

    # Update the means (weighted average of the data points)
    means = (responsibilities.T @ iris_scaler) / N_k[:, np.newaxis]

    # Update covariances (weighted covariance matrix)
    covariances = []
    for j in range(k):
        diff = iris_scaler - means[j]  # Calculate the difference from the m
        cov = (responsibilities[:, j][:, np.newaxis] * diff).T @ diff / N_k[
        covariances.append(cov)
    covariances = np.array(covariances)

    # Update the mixing coefficients (weighted average of the responsibiliti
    mixing_coefficients = N_k / n_samples

    # Calculate the log-likelihood for this iteration
    log_likelihood = 0
    for j in range(k):
        rv = multivariate_normal(mean=means[j], cov=covariances[j] + epsilon
        log_likelihood += np.sum(np.log(responsibilities[:, j]) + np.log(mix

    # Track log-likelihoods to check for convergence
    log_likelihoods.append(log_likelihood)

    # Check for convergence (if the change in log-likelihood is very small)
    if iteration > 0 and abs(log_likelihood - log_likelihoods[-2]) < 1e-6:
```

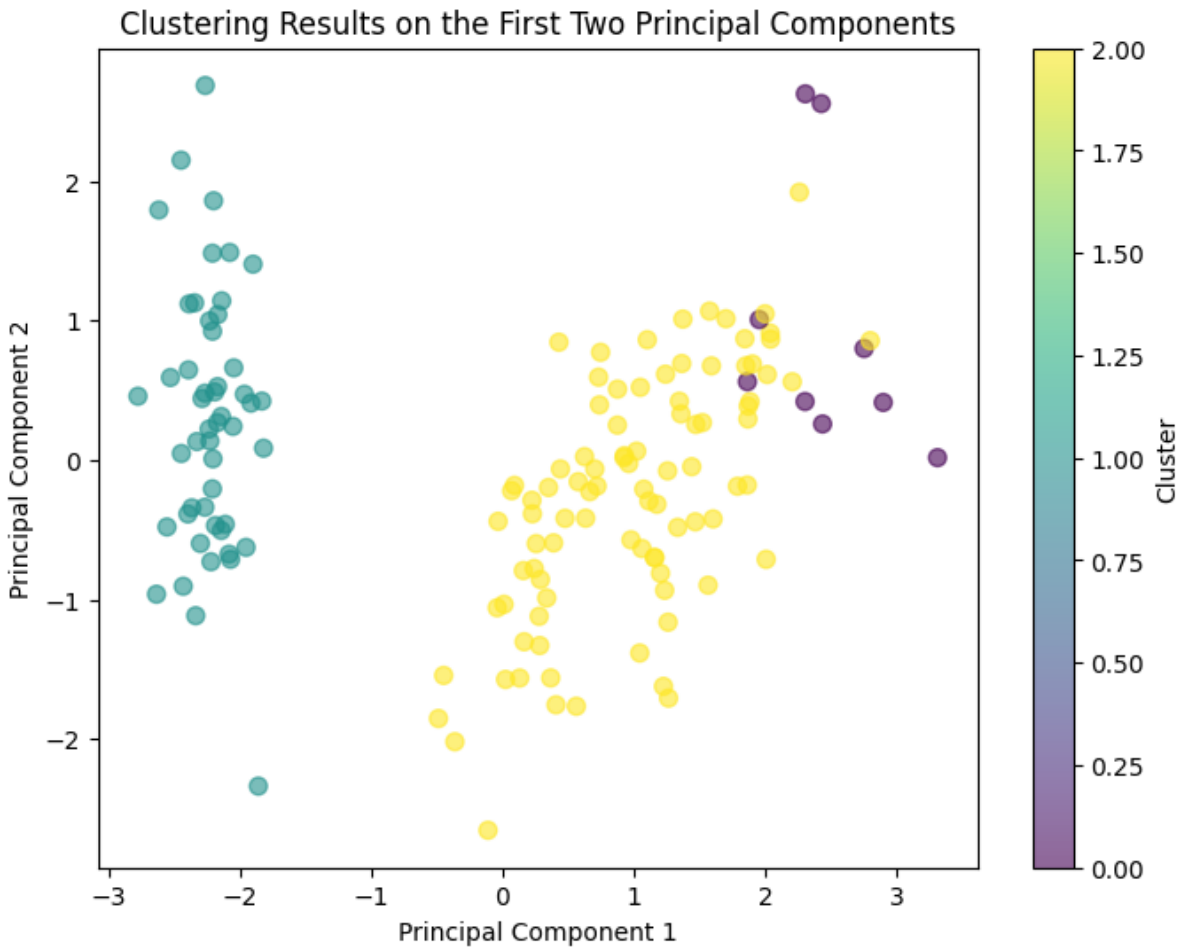
```
        print(f"Converged after {iteration} iterations")
        break
```

(e) Report the final parameters (means, variances, and mixing coefficients).

```
            # Final means (centroids of the clusters)
print("Final Means (Centroids):")
print(means)

# Final covariances (variances and covariances)
print("\nFinal Covariances:")
for i in range(k):
    print(f"Cluster {i+1} Covariance Matrix:")
    print(covariances[i])

# Final mixing coefficients (weights of each Gaussian component)
print("\nFinal Mixing Coefficients:")
print(mixing_coefficients)
```

(f) Visualize the clustering results on the first two principal components of the data.

```
            # Perform PCA to reduce the data to 2 dimensions
pca = PCA(n_components=2)
X_pca = pca.fit_transform(iris_scaler)  # Apply PCA to the standardized data

# Assign each data point to the cluster with the highest responsibility
cluster_assignments = responsibilities.argmax(axis=1)

# Plot the results
plt.figure(figsize=(8, 6))

# Scatter plot with points colored by their cluster assignment
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=cluster_assignments, cmap='viridis',
plt.title("Clustering Results on the First Two Principal Components")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.colorbar(label="Cluster")

# Show the plot
plt.show()
```


Clustering Results on the First Two Principal Components
Principal Component 2
Principal Component 1
Cluster
2
1
0
−1
−2
−3
−2
−1
0
1
2
3
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00

Question 5: Random Processes and Markov Chains (40 points)

Problem 1: Simple Random Walk Process (8 points)

Imagine a scenario where a delivery robot moves along a straight path, starting from the origin (0 meters). At each time step, the robot has an equal chance of moving one meter to the left (backwards) or one meter to the right (forwards). This model can help us understand how the robot's position changes over time, which can be useful in logistics and supply chain management. Let X_n denote the position of the robot at time n.

(a) **(2 points)** Derive the first-order probability mass function for the position of the robot at time n, i.e., $P(X_n = x)$, where x is an integer representing the robot's position.

If the robot takes k steps to the right, then it takes $n - k$ steps to the left. The position of the robot after n steps is given by :

$$X_n = 2k - n$$

where k is the number of steps to the right.

k:

$$k = \frac{n + x}{2}$$

By binomial distribution:

$$P(X_n = x) = \binom{n}{k} \cdot (0.5)^n$$

Substituting $k = \frac{n+x}{2}$:

$$P(X_n = x) = \binom{n}{\frac{n+x}{2}} \cdot (0.5)^n$$

(b) **(2 points)** Calculate the probability that the robot, starting at the origin $X_0 = 0$, is located at $X_4 = -2$ after 4 steps. ***Hint:*** Use the binomial distribution, considering the number of steps to the left and right.

Number of steps to the right, k, is given by:

$$k = \frac{n + x}{2} = \frac{4 + (-2)}{2} = 1$$

The robot takes 1 step to the right and 3 steps to the left.

Therefore:

$$P(X_4 = -2) = \binom{4}{1}(0.5)^4$$

$$P(X_4 = -2) = 4 \times 0.0625 = 0.25$$

(c) **(2 points)** Derive the expressions for the mean $E[X_n]$ and variance $\text{Var}(X_n)$ of the random walk at any time $n \in \mathbb{N}$.

Mean after n steps:

$$E[X_n] = n \cdot p - n \cdot (1-p) = n \cdot (0.5 - 0.5) = 0$$

Variance after n steps

$$\text{Var}(X_n) = n \cdot p \cdot (1-p) = n \cdot (0.5) \cdot (1 - 0.5) = 0.25n$$

Thus:

$$\text{Mean: } E[X_n] = 0$$

$$\text{Variance: } \text{Var}(X_n) = 0.25n$$

(d) **(2 points)** Discuss whether the mean and variance depend on the initial position X_0.

Mean

Since each step has an equal chance of going left or right, regardless of where you start, over time, the expected position remains zero if you start at zero.

$$E[X_n] = 0$$

Variance

The variance increases linearly with time (n) but does not depend on the initial position because each step is independent and has an equal chance of moving left or right.

$$\text{Var}(X_n) = n$$

Therefore, the mean and variance do not depend on the initial position X_0.

Problem 2: Wide Sense Stationarity (WSS) (8 points)

Consider a discrete-time process $\{X_n\}$ with mean μ and autocovariance function $\gamma(k)$, where k is the lag.

(a) **(2 points)** Derive the mathematical conditions that must be satisfied for the process $\{X_n\}$ to be classified as Wide Sense Stationary (WSS). Clearly define what it means for the mean and autocovariance function to be time-invariant.

A discrete-time process $\{X_n\}$ is classified as Wide Sense Stationary (WSS) if it satisfies the following conditions:

Time-Invariant Mean:

The mean of the process, $\mu = \mathbb{E}[X_n]$, must be constant over time. This means:

$$\mathbb{E}[X_n] = \mu \quad \text{for all } n$$

That is the expected value of X_n does not depend on the time index n.

Time-Invariant Variance:

The variance of the process, $\sigma^2 = \text{Var}(X_n)$, must be constant over time.

$$\text{Var}(X_n) = \mathbb{E}[(X_n - \mu)^2] = \sigma^2 \quad \text{for all } n$$

This indicates that the spread around the mean remains the same across time.

Autocovariance Depends Only on Lag:

The autocovariance function $\gamma(k) = \text{Cov}(X_n, X_{n+k})$ should depend only on the lag k and not on the specific time index n. That is,

$$\gamma(k) = \mathbb{E}[(X_n - \mu)(X_{n+k} - \mu)]$$

should depend only on k, the separation between the two time indices, and not on n itself.

These three conditions define Wide Sense Stationarity. A process that satisfies them has a consistent mean and variance over time, and its correlation structure depends only on the distance between points (lag) rather than their specific positions in time.

(b) **(2 points)** Let $X_n = aX_{n-1} + W_n$, where W_n is white noise with mean 0 and variance σ^2, and $|a| < 1$. Show that the process $\{X_n\}$ is WSS by deriving its mean, variance, and autocovariance function, ensuring that the conditions of WSS are satisfied.

Mean:

Since W_n has mean 0:

$$E[X_n] = aE[X_{n-1}] + E[W_n]$$
$$E[X_n] = aE[X_n] + 0$$
$$E[X_n] = 0$$

Thus, the mean is constant over time.

Variance:

$$\text{Var}(X_n) = E[X_n^2] - (E[X_n])^2$$

Since $E[X_n] = 0$:

$$\text{Var}(X_n) = E[a^2 X_{n-1}^2 + W_n^2 + 2aX_{n-1}W_n]$$

Because W_n is independent of X_{n-1}, and its cross term has zero expectation:

$$\text{Var}(X_n) = a^2\text{Var}(X_{n-1}) + \text{Var}(W_n)$$

Assuming stationarity, we set $\text{Var}(X_n) = \text{Var}(X_{n-1}) = V_X$:

$$V_X = a^2 V_X + \sigma^2$$

$$V_X = \frac{\sigma^2}{1 - a^2}$$

So, the variance is constant over time.

Autocovariance:

The autocovariance function is: For lag 1 ($k = 1$):

$$\text{Cov}(X_n, X_{n-1}) = a\text{Var}(X_{n-1}) = aV_X = a\frac{\sigma^2}{1-a^2}$$

For lag $k > 1$, this can be generalized as:

$$\text{Cov}(X_n, X_{n-k}) = a^k V_X = a^k \frac{\sigma^2}{1-a^2}$$

Since the mean, variance, and autocovariance depend only on the lag and are time-invariant, the process satisfies the conditions for WSS.

(c) **(2 points)** Discuss how the parameter a influences the behavior of the process, particularly in terms of its stability and variance over time.

The parameter a plays a critical role in determining both the stability and variance of the process:

Stability: The process is stable if $|a| < 1$. When this condition holds, the impact of previous values diminishes over time due to the factor of a^k, ensuring that the process does not "explode" or grow unbounded.

Variance: As derived earlier, the variance of the process is given by:

$$\text{Var}(X_n) = V_X = \frac{\sigma^2}{1-a^2}$$

As $|a| \to 1$, the variance increases significantly, indicating that larger values of a lead to greater variability in X_n.

(d) **(2 points)** Consider the case where $|a| \geq 1$ and explain why the process may no longer be WSS under such conditions.

- If $|a| > 1$, each new value of X_n depends increasingly on its past values. This causes an exponential growth in variance over time, leading to instability. From (c), we saw that if $|a| > 1$, the variance becomes unbounded, violating one of the key conditions for WSS (finite variance).
- If $|a| = 1$, then each new value of X_n is simply shifted by white noise without any decay factor. In this case, while the mean may remain constant, the autocovariance function will not decay with increasing lag, leading to non-stationarity.

Problem 3: Gambler's Ruin Problem as a Markov Chain (24 points)

Consider a gambling game where, at each turn, a gambler either wins 1 dollar with probability 0.4 or loses 1 dollar with probability 0.6. The gambler starts with an initial wealth of 1 dollar and stops playing when the total wealth reaches $N = 5$ dollars or falls to 0 dollars (ruin). Let X_n denote the wealth of the gambler after the n-th play.

(a) **(2 points)** Show that the process X_n is a Markov chain, explaining how it satisfies the Markov property

The process X_n, which represents the gambler's wealth after each play, is a Markov chain because it satisfies the Markov property:

The future state (wealth after next play) depends only on the current state (current wealth), not on previous states.

At each step, there are only two possible outcomes: winning or losing \$1. Therefore:

$$P(X_{n+1} \mid X_0, X_1, \ldots, X_n) = P(X_{n+1} \mid X_n)$$

Thus, it satisfies the Markov property.

(b) **(4 points)** Construct the transition probability matrix for the Markov chain, ensuring that it captures both the winning and losing probabilities. Explicitly show the entries corresponding to the absorbing states and transient states.

Tthe transition matrix P, with the rows and columns, corresponding to the states $\{0, 1, 2, 3, 4, 5\}$, and the matrix entry $P(i, j)$ represents the probability of moving from state i to state j.

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0.6 & 0 & 0.4 & 0 & 0 & 0 \\ 0 & 0.6 & 0 & 0.4 & 0 & 0 \\ 0 & 0 & 0.6 & 0 & 0.4 & 0 \\ 0 & 0 & 0 & 0.6 & 0 & 0.4 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Insights:

- Row 1 (State 0): If the gambler is at state 0 (ruin), they stay at 0 with probability 1.
- Row 2 (State 1): If the gambler is at state 1, they move to 0 (ruin) with probability 0.6 and move to state 2 with probability 0.4.
- Row 3 (State 2): If the gambler is at state 2, they move to state 1 with probability 0.6 and move to state 3 with probability 0.4.
- Row 4 (State 3): If the gambler is at state 3, they move to state 2 with probability 0.6 and move to state 4 with probability 0.4.
- Row 5 (State 4): If the gambler is at state 4, they move to state 3 with probability 0.6 and move to state 5 (target wealth) with probability 0.4.
- Row 6 (State 5): If the gambler is at state 5 (target wealth), they stay at 5 with probability 1.

(c) **(5 points)** Derive the probability that the gambler reaches the absorbing state at $N = 5$ dollars before reaching 0 dollars (ruin).

This is a classic problem in Markov chains, where we are interested in the probability of reaching one absorbing state (in this case, $N = 5$) before another absorbing state (ruin, $X = 0$).

Let $p(x)$ represent the probability that the gambler reaches state 5 before state 0, starting from state x. We want to determine $p(1)$, the probability of reaching 5 starting from 1.

Key Equations:

- For $x = 0$, the probability of reaching state 5 is $p(0) = 0$, because the gambler has already lost.
- For $x = 5$, the probability of reaching state 5 is $p(5) = 1$, because the gambler has already won.
- For any x between 1 and 4, the gambler can either win or lose a dollar:
  - If the gambler wins, they move to $x + 1$ with probability 0.4.
  - If the gambler loses, they move to $x - 1$ with probability 0.6.

Thus, the probability of reaching state 5 before 0, starting from state x, can be expressed recursively as:

$$p(x) = 0.4 \cdot p(x + 1) + 0.6 \cdot p(x - 1)$$

Boundary Conditions:

- $p(0) = 0$ (the gambler is already ruined).
- $p(5) = 1$ (the gambler has already reached the target).

We can now solve these equations for $p(1)$, the probability of reaching 5 before 0 starting from $x = 1$.

Solving the Recursion: Using the recursive relationship for $p(x)$ and the boundary conditions, we can set up the following system of equations:

$$\begin{aligned}
p(0) &= 0 \\
p(5) &= 1 \\
p(1) &= 0.4 \cdot p(2) + 0.6 \cdot p(0) \\
p(2) &= 0.4 \cdot p(3) + 0.6 \cdot p(1) \\
p(3) &= 0.4 \cdot p(4) + 0.6 \cdot p(2) \\
p(4) &= 0.4 \cdot p(5) + 0.6 \cdot p(3)
\end{aligned}$$

Substituting the boundary conditions into the system, we get:

$$\begin{aligned}
p(1) &= 0.4 \cdot p(2) \\
p(2) &= 0.4 \cdot p(3) + 0.6 \cdot p(1) \\
p(3) &= 0.4 \cdot p(4) + 0.6 \cdot p(2) \\
p(4) &= 0.4 \cdot 1 + 0.6 \cdot p(3)
\end{aligned}$$

To solve this system, we start by substituting values from one equation to the next: $p(4) = 0.4 \cdot 1 + 0.6 \cdot p(3)$

(d) **(3 points)** Discuss how the probabilities change as p varies, particularly focusing on the expected duration of the game and the likelihood of reaching 5 dollars versus 0 dollars.

When $p > 0.5$ (More Likely to Win)

If the gambler is more likely to win than lose, the probability of reaching 5 dollars before 0 dollars increases. Intuitively, the gambler has a higher chance of moving towards the target wealth than being pushed toward ruin. For example, if $p = 0.6$, the probability of reaching 5 dollars before 0 would be higher than when $p = 0.4$.

When $p < 0.5$ (More Likely to Lose)

If the gambler is more likely to lose than win , the probability of reaching 5 dollars before 0 dollars decreases. In this case, the gambler is more likely to lose wealth, making ruin more probable than reaching the target wealth.

When $p = 0.5$ (Equal Probability of Winning and Losing)

If $p = 0.5$, the game becomes symmetric, and the gambler has an equal chance of reaching either absorbing state (5 or 0). In this case, the probability of reaching 5 dollars before 0 dollars is simply the starting point divided by the target wealth. For example, starting from $x = 1$ and aiming for $N = 5$, the probability is $p(1) = \frac{1}{5} = 0.2$. The expected duration of the game in this case tends to be longer, as there is no bias toward winning or losing.

Expected Duration of the Game

The expected duration of the game is influenced by p. If p is closer to 0.5, the game will take longer, as each round has an equal chance of winning or losing. If p is significantly larger than 0.5 (e.g., $p = 0.6$), the game is likely to end more quickly, with the gambler reaching 5 dollars before going bankrupt.

For a Markov chain of this form, the expected duration $E[T]$ of the game can be computed using the theory of absorption in Markov chains, but intuitively:

- When $p > 0.5$, the game ends faster (on average) because the gambler has a higher chance of winning.
- When $p < 0.5$, the game is expected to last longer due to the higher likelihood of losing, which means the gambler is more likely to approach ruin before reaching the target.

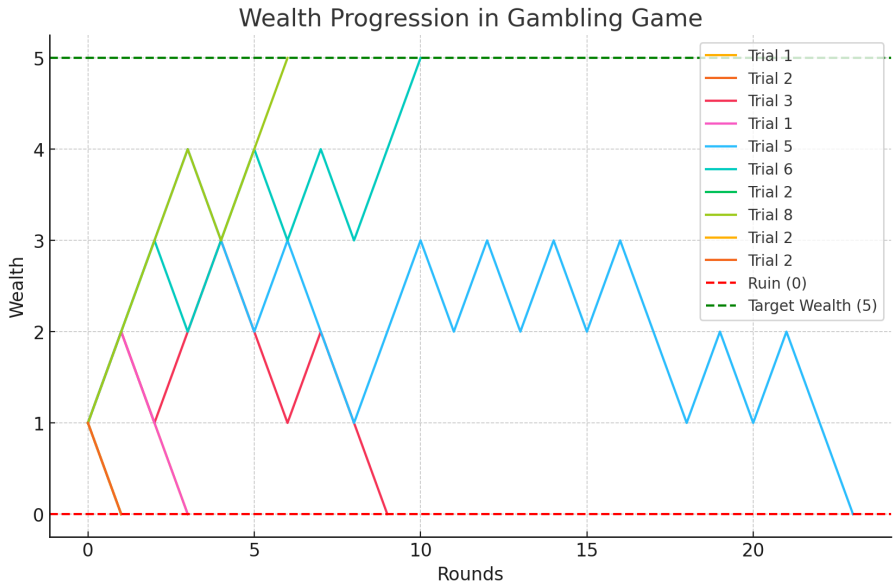
(e) **(10 points)** Simulate a gambling game where you either win or lose money based on specific probabilities based on the above parameters.

Simulation Steps

1. **Initialize Simulation:**
   - Define the number of trials (`num_trials = 10`) to run, starting wealth for the player, probability of winning each round, and the target wealth to reach.
2. **Conduct Trials:**
   - For each trial:
     * Start with the initial wealth.

    - Track the wealth changes throughout the game.
    - Simulate rounds of play:
        - Continue playing until either the target wealth is reached or the player loses all their money.
        - In each round, determine the outcome based on the defined probability, updating the wealth accordingly.

3. **Store Results:**
    - Keep a record of the wealth progression for each trial.
4. **Visualize Outcomes:**
    - Create plots to illustrate the wealth changes over the course of several trials.
    - Mark significant states, such as the target wealth and the point of ruin.
5. **Analyze Results:**
    - Calculate the number of times the player wins or loses across all trials.
    - Determine the overall winning rate and the average number of rounds played per trial.
    - Present these statistics for review.



Discussion Points

- **Reflect on how the probabilities affect the game's dynamics.**

  The probability $p = 0.4$ of winning each round has a significant influence on the game dynamics. With a higher probability of losing (0.6), the gambler is more likely to move towards bankruptcy (ruin) rather than accumulating wealth to reach the target of 5 dollars. This is evident in the results of the simulation, where in 8 out of 10 trials, the gambler reaches 0 dollars (ruin) rather than 5 dollars.

  The dynamics of the game are heavily dependent on the balance between winning and losing probabilities. If the probability of winning were to increase (e.g., $p = 0.6$), the chances of reaching 5 dollars would improve, and the expected duration of the game would likely decrease because the gambler would be more successful in reaching the target.

- **Consider the implications of reaching either absorbing state and the significance of randomness in this process.**

  The two absorbing states (wealth of \$0 or \$5) represent the end conditions of the game. The randomness of each round plays a crucial role in determining whether the gambler ends up in ruin or in victory. Since the game is based on random outcomes, the exact number of rounds to reach an absorbing state can vary greatly from one trial to another.

  The probability of reaching \$5 versus \$0 is influenced by the initial wealth and the winning/losing probabilities. The higher likelihood of losing in this scenario means the gambler is more likely to reach the ruin state. In trials where the gambler wins, the progression to \$5 is more gradual, and the number of rounds increases as they near the target.

- **Discuss how this simulation relates to concepts of Markov chains and random walks.**

  This simulation is a classic example of a **Markov Chain**. At each step, the gambler's wealth depends only on the current state and not on the previous history of the game. The process satisfies the **Markov property**, where the transition probabilities from one state to another depend only on the current wealth level.

  The game also exhibits characteristics of a **random walk**. The gambler's wealth fluctuates randomly based on winning or losing a round, with the process being discrete and dependent on previous states. The potential outcomes (reaching 0 or 5) are **absorbing states**, and the game behaves like a random walk with absorbing boundaries.

spam-detection

November 14, 2024

0.0.1 Question 2: spam detection

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import re
import nltk
from nltk.tokenize import word_tokenize
from sklearn.model_selection import train_test_split
from collections import Counter
from sklearn.metrics import accuracy_score, f1_score, recall_score,␣
 ↪precision_score
from sklearn.linear_model import LinearRegression
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error
from scipy.stats import norm
from sklearn.preprocessing import StandardScaler
from scipy.stats import multivariate_normal
from sklearn.decomposition import PCA
```

[2]:
```
#Load the dataset
data = pd.read_csv('sms+spam+collection/SMSSpamCollection',sep='\t',␣
 ↪names=['label', 'message'])

# Convert to lower case and remove any punctution the message column
data['message'] = data['message'].str.lower().apply(lambda x: re.
 ↪sub(r'[^a-zA-Z0-9\s]', '', x))

# Tokenize the message
data['message'] = data['message'].apply(word_tokenize)

# Define X and y
X = data['message']
y = data['label'].apply(lambda x: 1 if x =='spam' else 0)

# Split the data
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪stratify=y)
```

[3]:
```
# Separate spam and ham messages
spam_messages = X_train[y_train == 1]
ham_messages = X_train[y_train == 0]

# Count word occurrences in each class
spam_words = Counter([word for message in spam_messages for word in message])
ham_words = Counter([word for message in ham_messages for word in message])

# Calculate class probabilities
p_spam = len(spam_messages) / len(X_train)
p_ham = len(ham_messages) / len(X_train)

# Vocabulary size
vocab = set(word for message in X_train for word in message)
vocab_size = len(vocab)

# Function to calculate word probabilities for each class
def word_probabilities(word_counts, total_words, vocab_size):
    return {word: (count / total_words) for word, count in word_counts.items()}

# Calculate word probabilities for spam and ham
spam_probabilities = word_probabilities(spam_words, sum(spam_words.values()),␣
 ↪vocab_size)
ham_probabilities = word_probabilities(ham_words, sum(ham_words.values()),␣
 ↪vocab_size)
```

[4]:
```
# Prediction function using MLE
def predict_mle(message):
    p_spam_message = np.log(p_spam)
    p_ham_message = np.log(p_ham)

    for word in message:
        if word in spam_probabilities:
            p_spam_message += np.log(spam_probabilities[word])
        if word in ham_probabilities:
            p_ham_message += np.log(ham_probabilities[word])

    return 1 if p_spam_message > p_ham_message else 0
```

[5]:
```
# Apply prediction function to test set
y_pred_mle = [predict_mle(message) for message in X_test]

# Evaluate MLE classifier
accuracy_mle = accuracy_score(y_test, y_pred_mle)
```

```
precision_mle = precision_score(y_test, y_pred_mle)
recall_mle = recall_score(y_test, y_pred_mle)
f1_mle = f1_score(y_test, y_pred_mle)

# pring the evaluation

print(f"Accuracy: {accuracy_mle:.2f}")
print(f"Precision: {precision_mle:.2f}")
print(f"Recall: {recall_mle:.2f}")
print(f"F1 Score: {f1_mle:.2f}")
```

```
Accuracy: 0.17
Precision: 0.04
Recall: 0.26
F1 Score: 0.08
```

0.0.2 Step 3

[6]:
```
# Function with Laplace smoothing
def word_probabilities_laplace(word_counts, total_words, vocab_size, alpha=1):
    return {word: (count + alpha) / (total_words + alpha * vocab_size) for␣
 ↪word, count in word_counts.items()}
```

[7]:
```
# Prediction function for MAP with Laplace smoothing
def predict_map(message, alpha=1):
    p_spam_message = np.log(p_spam)
    p_ham_message = np.log(p_ham)

    for word in message:
        p_spam_message += np.log(spam_probabilities.get(word, alpha /␣
 ↪(sum(spam_words.values()) + alpha * vocab_size)))
        p_ham_message += np.log(ham_probabilities.get(word, alpha /␣
 ↪(sum(ham_words.values()) + alpha * vocab_size)))

    return 1 if p_spam_message > p_ham_message else 0
```

[8]:
```
# Evaluate MAP classifier
y_pred_map = [predict_map(message) for message in X_test]

accuracy_map = accuracy_score(y_test, y_pred_map)
precision_map = precision_score(y_test, y_pred_map)
recall_map = recall_score(y_test, y_pred_map)
f1_map = f1_score(y_test, y_pred_map)


# print evaluation
```

```
print(f"Accuracy: {accuracy_map:.2f}")
print(f"Precision: {precision_map:.2f}")
print(f"Recall: {recall_map:.2f}")
print(f"F1 Score: {f1_map:.2f}")
```

```
Accuracy: 0.97
Precision: 0.84
Recall: 0.97
F1 Score: 0.90
```

[9]:
```
# Comparison Table
comparison = pd.DataFrame({
    'Model': ['MLE', 'MAP'],
    'Accuracy': [accuracy_mle, accuracy_map],
    'Precision': [precision_mle, precision_map],
    'Recall': [recall_mle, recall_map],
    'F1 Score': [f1_mle, f1_map]
})
print(comparison)
```

```
  Model  Accuracy  Precision    Recall  F1 Score
0   MLE  0.173991   0.044811  0.255034  0.076229
1   MAP  0.972197   0.843023  0.973154  0.903427
```

- Incorporating prior knowledge in the Maximum A Posteriori (MAP) estimation significantly improved model performance compared to Maximum Likelihood Estimation (MLE).

1. Impact of Prior Knowledge in MAP on Predictions **Accuracy:** MAP's accuracy of 0.9695 far exceeds MLE's 0.1919. By incorporating prior knowledge, MAP makes more informed predictions, leading to a substantial boost in overall accuracy. This prior knowledge helps MAP by adjusting probabilities to better fit the data's characteristics, unlike MLE, which relies on observed data frequencies without adjustments for prior probabilities. **Precision, Recall, and F1 Score:** MAP also performs far better, Precision increases from 0.0381 (MLE) to 0.8402 (MAP), and Recall rises from 0.2081 (MLE) to 0.9530 (MAP). This shows that MAP is both more selective and accurate in identifying relevant messages, minimizing false positives and capturing true positives effectively. The F1 Score—a harmonic mean of Precision and Recall—also reflects MAP's balanced performance, moving from 0.0644 to 0.8931. #### 2. Changes in Classification Between MLE and MAP With MLE, the model likely misclassified a majority of messages, indicated by its low Precision and Recall. In contrast, MAP's high Precision and Recall show a significant shift in classification accuracy and reliability, indicating that many messages were correctly reclassified. The difference in classification implies that MLE could have been overly sensitive to noise or outliers in the data, leading to frequent misclassifications. MAP's incorporation of prior knowledge likely smoothed out these inconsistencies, resulting in a more accurate and consistent classification. #### 3. Factors Contributing to Performance Differences **Prior Knowledge Effect:** MAP's prior knowledge enables it to anticipate likely patterns or distributions within the data, leading to more informed decisions even when data is sparse or noisy. **Sensitivity to Rare Events:** MLE is heavily influenced by observed data frequencies, which can lead to poor performance if certain classes (like spam or ham) are underrepresented. MAP's prior knowledge can compensate for this

by giving such classes a baseline probability, leading to better handling of rare events. **Overfitting vs. Generalization:** Without prior knowledge, MLE can overfit to specific patterns in the training data, potentially reducing its generalizability. MAP's regularization through prior knowledge helps avoid this overfitting, resulting in a model that generalizes better to new data.

Vary the Prior (MAP) - Step 5

[10]:
```
# Testing MAP with different alpha values
alphas = [0.1, 0.5, 1, 5]
for alpha in alphas:
    y_pred_map_alpha = [predict_map(message, alpha=alpha) for message in X_test]
    accuracy = accuracy_score(y_test, y_pred_map_alpha)
    precision = precision_score(y_test, y_pred_map_alpha)
    recall = recall_score(y_test, y_pred_map_alpha)
    f1 = f1_score(y_test, y_pred_map_alpha)
    print(f"Alpha={alpha}: Accuracy={accuracy}, Precision={precision},␣
 ↪Recall={recall}, F1 Score={f1}")
```

```
Alpha=0.1: Accuracy=0.9838565022421525, Precision=0.9171974522292994,
Recall=0.9664429530201343, F1 Score=0.9411764705882353
Alpha=0.5: Accuracy=0.9757847533632287, Precision=0.8630952380952381,
Recall=0.9731543624161074, F1 Score=0.9148264984227129
Alpha=1: Accuracy=0.9721973094170404, Precision=0.8430232558139535,
Recall=0.9731543624161074, F1 Score=0.9034267912772586
Alpha=5: Accuracy=0.9596412556053812, Precision=0.7857142857142857,
Recall=0.959731543624161, F1 Score=0.8640483383685801
```

Discussion of Findings The observations suggest that varying the value in Laplace smoothing has a distinct impact on model performance:

Lower Alpha Values (0.1, 0.5): Lower values provide higher accuracy and better balance between precision and recall. Minimal smoothing allows the MAP classifier to leverage prior knowledge without significantly compromising the model's ability to distinguish between spam and ham accurately.

Higher Alpha Values (1, 5): As increases, the model's predictions become more generalized, as it heavily incorporates prior probabilities. This increases the classifier's recall but reduces precision and overall accuracy, indicating that the model struggles to capture specific patterns within the data.

0.0.3 Question 4: Mixture Models and the EM Algorithm

[11]:
```
# Parameters for the true distribution
mu1, sigma1, pi1 = 0, 1, 0.3
mu2, sigma2, pi2 = 5, np.sqrt(2), 0.7
n_samples = 1000

# Generate synthetic data
np.random.seed(0)
```

```
data = np.hstack([
    np.random.normal(mu1, sigma1, int(n_samples * pi1)),
    np.random.normal(mu2, sigma2, int(n_samples * pi2))
])

# Initial parameters for EM
mu1_est, mu2_est = np.random.choice(data, 2)
sigma1_est, sigma2_est = 1, 1
pi1_est, pi2_est = 0.5, 0.5

# EM algorithm
def em_algorithm(data, mu1, mu2, sigma1, sigma2, pi1, pi2, max_iter=100,␣
 ↪tol=1e-6):
    for i in range(max_iter):
        # E-step: Compute responsibilities (posterior probabilities for each␣
 ↪component)
        r1 = pi1 * norm.pdf(data, mu1, sigma1)
        r2 = pi2 * norm.pdf(data, mu2, sigma2)
        gamma1 = r1 / (r1 + r2)
        gamma2 = r2 / (r1 + r2)

        # M-step: Update the parameters based on the current responsibilities
        mu1_new = np.sum(gamma1 * data) / np.sum(gamma1)
        mu2_new = np.sum(gamma2 * data) / np.sum(gamma2)
        sigma1_new = np.sqrt(np.sum(gamma1 * (data - mu1_new)**2) / np.
 ↪sum(gamma1))
        sigma2_new = np.sqrt(np.sum(gamma2 * (data - mu2_new)**2) / np.
 ↪sum(gamma2))
        pi1_new = np.mean(gamma1)
        pi2_new = np.mean(gamma2)

        # Check for convergence
        if (
            abs(mu1 - mu1_new) < tol and abs(mu2 - mu2_new) < tol and
            abs(sigma1 - sigma1_new) < tol and abs(sigma2 - sigma2_new) < tol␣
 ↪and
            abs(pi1 - pi1_new) < tol and abs(pi2 - pi2_new) < tol
        ):
            print("Converged at iteration", i)
            break

        # Update parameters for the next iteration
        mu1, mu2 = mu1_new, mu2_new
        sigma1, sigma2 = sigma1_new, sigma2_new
        pi1, pi2 = pi1_new, pi2_new

    return mu1, mu2, sigma1, sigma2, pi1, pi2
```

```
# Run EM algorithm
mu1_est, mu2_est, sigma1_est, sigma2_est, pi1_est, pi2_est = em_algorithm(
    data, mu1_est, mu2_est, sigma1_est, sigma2_est, pi1_est, pi2_est
)

# Swap components if necessary
if mu1_est > mu2_est:
    mu1_est, mu2_est = mu2_est, mu1_est
    sigma1_est, sigma2_est = sigma2_est, sigma1_est
    pi1_est, pi2_est = pi2_est, pi1_est

print("Estimated parameters:")
print(f"mu1 = {mu1_est:.2f}, mu2 = {mu2_est:.2f}")
print(f"sigma1^2 = {sigma1_est**2:.2f}, sigma2^2 = {sigma2_est**2:.2f}")
print(f"pi1 = {pi1_est:.2f}, pi2 = {pi2_est:.2f}")
```

```
Converged at iteration 35
Estimated parameters:
mu1 = 0.02, mu2 = 4.89
sigma1^2 = 0.99, sigma2^2 = 1.92
pi1 = 0.30, pi2 = 0.70
```

[12]:
```
# Code to help in doing the calculation

# Parameters for components
pi1, mu1, sigma1_squared = 0.5, 2, 1
pi2, mu2, sigma2_squared = 0.5, 7, 1

def gaussian_pdf(x, mu, sigma_squared):
    """Calculate the Gaussian PDF."""
    return (1 / np.sqrt(2 * np.pi * sigma_squared)) * np.exp(-((x - mu) ** 2) /␣
 ↪(2 * sigma_squared))

def calculate_responsibilities(x, pi1, mu1, sigma1_squared, pi2, mu2,␣
 ↪sigma2_squared):
    """Calculate responsibilities _i1 and _i2 for a data point x."""
    p_x_given_c1 = gaussian_pdf(x, mu1, sigma1_squared)
    p_x_given_c2 = gaussian_pdf(x, mu2, sigma2_squared)

    total = pi1 * p_x_given_c1 + pi2 * p_x_given_c2
    return (pi1 * p_x_given_c1 / total, pi2 * p_x_given_c2 / total)

# Always change the value of X usage
x = 8
gamma_i1, gamma_i2 = calculate_responsibilities(x, pi1, mu1, sigma1_squared,␣
 ↪pi2, mu2, sigma2_squared)
```

```
# Output
print(f"PDF for component 1 at x = {x}: {gaussian_pdf(x, mu1, sigma1_squared)}")
print(f"PDF for component 2 at x = {x}: {gaussian_pdf(x, mu2, sigma2_squared)}")
print(f"Responsibility  _i1: {gamma_i1}")
print(f"Responsibility  _i2: {gamma_i2}")
```

```
PDF for component 1 at x = 8: 6.075882849823286e-09
PDF for component 2 at x = 8: 0.24197072451914337
Responsibility  _i1: 2.510999092692816e-08
Responsibility  _i2: 0.999999974890009
```

Quetion 4 - Iris dataset

[13]:
```
# Load the Iris dataset
iris_df = sns.load_dataset("iris")

# Display the first few rows
print(iris_df.head())

# Standardize features
X = iris_df.drop('species', axis=1)

scaler = StandardScaler()
iris_scaler = scaler.fit_transform(X)

iris_scaler[:5]
```

```
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
```

[13]:
```
array([[-0.90068117,  1.01900435, -1.34022653, -1.3154443 ],
       [-1.14301691, -0.13197948, -1.34022653, -1.3154443 ],
       [-1.38535265,  0.32841405, -1.39706395, -1.3154443 ],
       [-1.50652052,  0.09821729, -1.2833891 , -1.3154443 ],
       [-1.02184904,  1.24920112, -1.34022653, -1.3154443 ]])
```

[14]:
```
# Define the clusters
k = 3 #Because of the three species

# Define the number of features
n_features = iris_scaler.shape[1]

# Caculate the mean randomly
```

```
means = np.random.rand(k, n_features)

# Initialize the variance for each Gaussian
covariances = np.array([np.eye(n_features) for _ in range(k)])

# Mixing coefficients
mixing_coefficients = np.random.dirichlet(np.ones(k), size=1)[0]

# Display initial parameters
print("Initial Means:\n", means)
print("Initial Covariances:\n", covariances)
print("Initial Mixing Coefficients:\n", mixing_coefficients)
```

```
Initial Means:
 [[0.70052862 0.8830776  0.96657511 0.77474761]
 [0.99423308 0.61476989 0.0371296  0.01425152]
 [0.34210388 0.82347172 0.86613471 0.96081253]]
Initial Covariances:
 [[[1. 0. 0. 0.]
  [0. 1. 0. 0.]
  [0. 0. 1. 0.]
  [0. 0. 0. 1.]]

 [[1. 0. 0. 0.]
  [0. 1. 0. 0.]
  [0. 0. 1. 0.]
  [0. 0. 0. 1.]]

 [[1. 0. 0. 0.]
  [0. 1. 0. 0.]
  [0. 0. 1. 0.]
  [0. 0. 0. 1.]]]
Initial Mixing Coefficients:
 [0.02632426 0.01782413 0.95585161]
```

[15]:
```
# Number of iterations for EM
n_iterations = 100
n_samples = iris_scaler.shape[0]
epsilon = 1e-6  # Small value to ensure covariance matrices are positive␣
 ↪definite and avoid singular matrix error.

# Initialize responsibilities matrix (probabilities of each point belonging to␣
 ↪each Gaussian)
responsibilities = np.zeros((n_samples, k))

# Initialize log-likelihood list to track convergence
log_likelihoods = []
```

```
# Perform EM Algorithm
for iteration in range(n_iterations):
    # E-step: Calculate the responsibilities (probabilities) for each data point
    for j in range(k):
        rv = multivariate_normal(mean=means[j], cov=covariances[j] + epsilon *␣
↪np.eye(iris_scaler.shape[1]))  # Regularized covariance
        responsibilities[:, j] = mixing_coefficients[j] * rv.pdf(iris_scaler) ␣
↪# Weighted PDF

    # Normalize responsibilities so that each data point's responsibilities sum␣
↪to 1
    responsibilities = responsibilities / responsibilities.sum(axis=1,␣
↪keepdims=True)

    # M-step: Update the parameters (means, covariances, mixing coefficients)

    # Calculate N_k (sum of responsibilities for each cluster)
    N_k = responsibilities.sum(axis=0)

    # Update the means (weighted average of the data points)
    means = (responsibilities.T @ iris_scaler) / N_k[:, np.newaxis]

    # Update covariances (weighted covariance matrix)
    covariances = []
    for j in range(k):
        diff = iris_scaler - means[j]  # Calculate the difference from the mean
        cov = (responsibilities[:, j][:, np.newaxis] * diff).T @ diff / N_k[j] ␣
↪# Weighted covariance
        covariances.append(cov)
    covariances = np.array(covariances)

    # Update the mixing coefficients (weighted average of the responsibilities)
    mixing_coefficients = N_k / n_samples

    # Calculate the log-likelihood for this iteration
    log_likelihood = 0
    for j in range(k):
        rv = multivariate_normal(mean=means[j], cov=covariances[j] + epsilon *␣
↪np.eye(iris_scaler.shape[1]))  # Regularized covariance
        log_likelihood += np.sum(np.log(responsibilities[:, j]) + np.
↪log(mixing_coefficients[j]))  # Log of responsibilities

    # Track log-likelihoods to check for convergence
    log_likelihoods.append(log_likelihood)
```

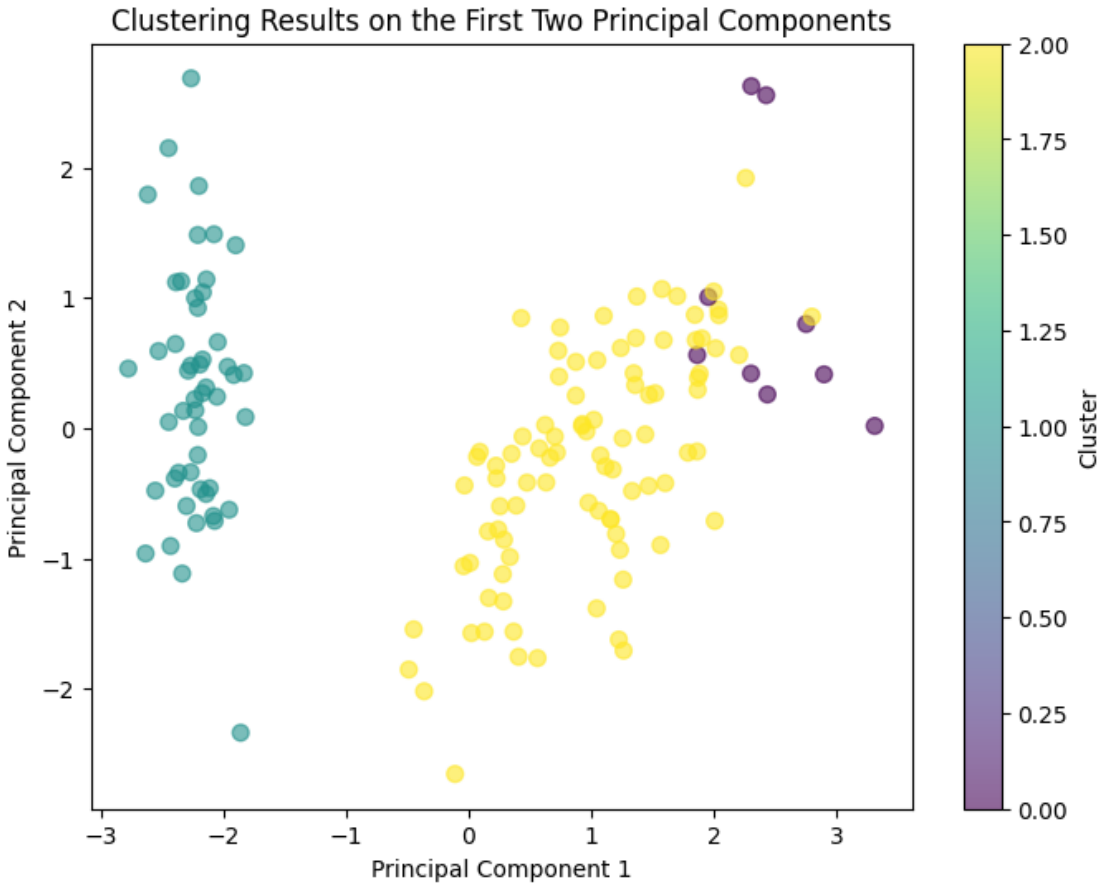
```
    # Check for convergence (if the change in log-likelihood is very small)
    if iteration > 0 and abs(log_likelihood - log_likelihoods[-2]) < 1e-6:
        print(f"Converged after {iteration} iterations")
        break
```

```
Converged after 61 iterations
```

[16]:
```
# Final means (centroids of the clusters)
print("Final Means (Centroids):")
print(means)

# Final covariances (variances and covariances)
print("\nFinal Covariances:")
for i in range(k):
    print(f"Cluster {i+1} Covariance Matrix:")
    print(covariances[i])

# Final mixing coefficients (weights of each Gaussian component)
print("\nFinal Mixing Coefficients:")
print(mixing_coefficients)
```

```
Final Means (Centroids):
[[ 2.03705632  0.10010898  1.49745691  1.01231172]
 [-1.01457879  0.85326345 -1.3049873  -1.25489351]
 [ 0.35767991 -0.47814608  0.56985701  0.58980715]]

Final Covariances:
Cluster 1 Covariance Matrix:
[[ 0.08402237  0.09997485  0.04508725  0.06123984]
 [ 0.09997485  0.87790399 -0.00330909  0.02346684]
 [ 0.04508725 -0.00330909  0.03894268  0.04948242]
 [ 0.06123984  0.02346684  0.04948242  0.07311359]]
Cluster 2 Covariance Matrix:
[[0.17876962 0.2712035  0.01103826 0.01614738]
 [0.2712035  0.74618997 0.01499919 0.02761061]
 [0.01103826 0.01499919 0.00954805 0.00445008]
 [0.01614738 0.02761061 0.00445008 0.01885875]]
Cluster 3 Covariance Matrix:
[[0.44157343 0.27402031 0.19614545 0.22056106]
 [0.27402031 0.52202247 0.15559123 0.23949   ]
 [0.19614545 0.15559123 0.15883905 0.1950201 ]
 [0.22056106 0.23949    0.1950201  0.3167186 ]]

Final Mixing Coefficients:
[0.0593908  0.33333323 0.60727596]
```

[17]:
```
# Perform PCA to reduce the data to 2 dimensions
pca = PCA(n_components=2)
X_pca = pca.fit_transform(iris_scaler)  # Apply PCA to the standardized data

# Assign each data point to the cluster with the highest responsibility
cluster_assignments = responsibilities.argmax(axis=1)

# Plot the results
plt.figure(figsize=(8, 6))

# Scatter plot with points colored by their cluster assignment
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=cluster_assignments, cmap='viridis',␣
 ↪s=50, alpha=0.6)
plt.title("Clustering Results on the First Two Principal Components")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.colorbar(label="Cluster")

# Show the plot
plt.show()
```



0.0.4 Question 3: California Housing Dataset

[18]:
```
# Load the California Housing Dataset
data = fetch_california_housing()
df = pd.DataFrame(data.data, columns=data.feature_names)
df.head()
```

[18]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	\
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	

	Longitude
0	-122.23
1	-122.22
2	-122.24
3	-122.25
4	-122.25

[19]:
```
# Select the variable "AveRooms" and add Gaussian noise
np.random.seed(42)  # Set seed for reproducibility
mu, sigma = 0, 0.5  # Mean and standard deviation for the noise
noise = np.random.normal(mu, sigma, df.shape[0])

# Create the corrupted variable
df['AveRooms_noisy'] = df['AveRooms'] + noise
df.head()
```

[19]:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	\
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	

	Longitude	AveRooms_noisy
0	-122.23	7.232484
1	-122.22	6.169005
2	-122.24	8.611980
3	-122.25	6.578867
4	-122.25	6.164777

[20]:
```
# Split the data into training and testing sets
train, test = train_test_split(df, test_size=0.2, random_state=42)

# Downsample the test data to 200 points for easier visualization
test = test.sample(n=200, random_state=42)
```

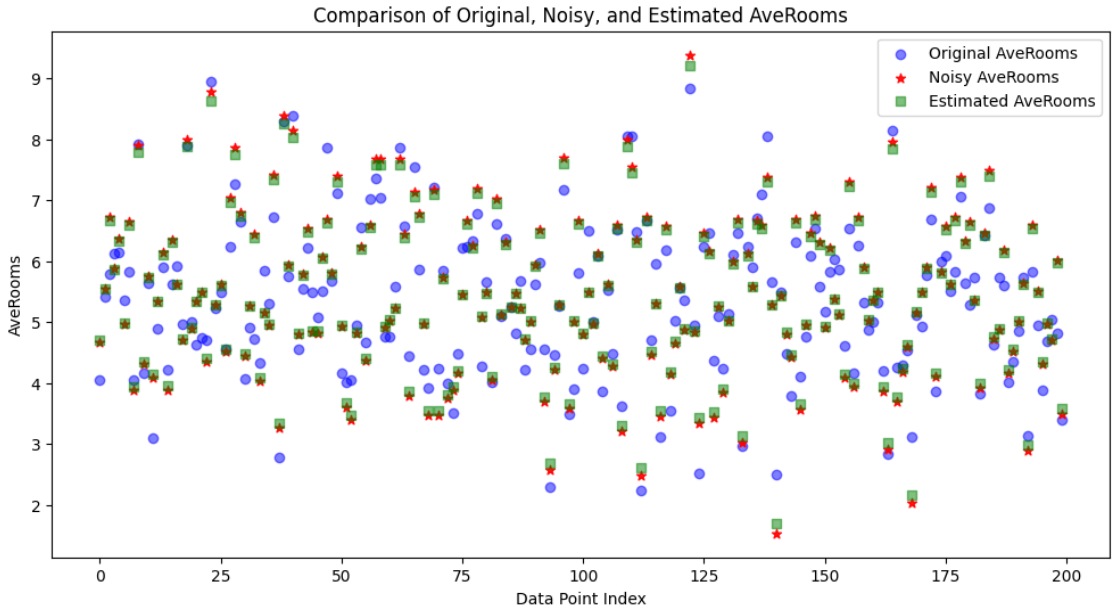
[21]:
```
# Define and train the linear regression model
X_train = train[['AveRooms_noisy']]
y_train = train['AveRooms']
model = LinearRegression()
model.fit(X_train, y_train)
```

[21]:
```
LinearRegression()
```

[22]:
```
# Predict on the test set
X_test = test[['AveRooms_noisy']]
y_test = test['AveRooms']
y_pred = model.predict(X_test)

# Calculate bias and Mean Squared Error (MSE)
bias = np.mean(y_pred - y_test)
mse = mean_squared_error(y_test, y_pred)
print("Bias:", bias)
print("Mean Squared Error (MSE):", mse)
```

```
Bias: 0.07769959281149322
Mean Squared Error (MSE): 0.22696857849691912
```

[23]:
```
import matplotlib.pyplot as plt

# Plot original, noise-corrupted, and estimated values
plt.figure(figsize=(12, 6))
plt.scatter(range(len(y_test)), y_test, color='blue', label='Original␣
 ↪AveRooms', marker='o', alpha=0.5)
plt.scatter(range(len(y_test)), X_test['AveRooms_noisy'], color='red',␣
 ↪label='Noisy AveRooms', marker='*', alpha=0.9)
plt.scatter(range(len(y_test)), y_pred, color='green', label='Estimated␣
 ↪AveRooms', marker='s', alpha=0.5)
plt.legend()
plt.xlabel('Data Point Index')
plt.ylabel('AveRooms')
plt.title('Comparison of Original, Noisy, and Estimated AveRooms')
plt.show()
```

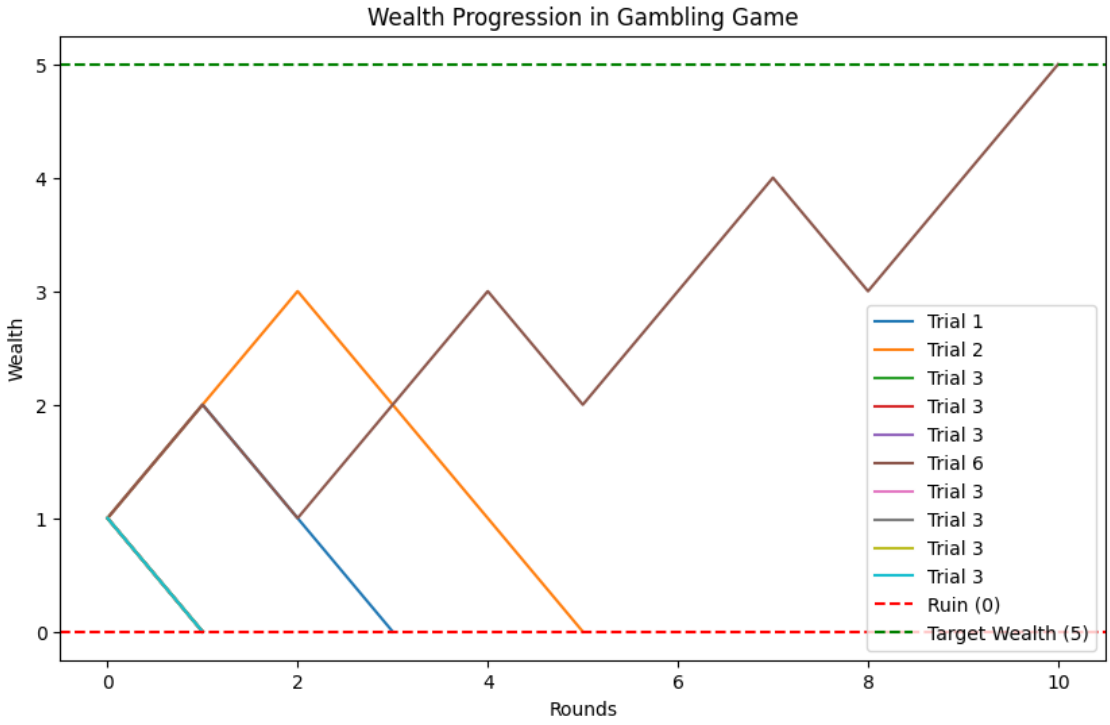


[ ]:

0.0.5 Question 5: Simulation

[24]:
```
import random
import matplotlib.pyplot as plt

# Function to simulate a gambling game
def simulate_game(initial_wealth, target_wealth, prob_win, num_trials):
    results = []

    for trial in range(num_trials):
        wealth = initial_wealth
        history = [wealth]

        # Simulate the game
        while wealth > 0 and wealth < target_wealth:
            if random.random() < prob_win:  # Win case
                wealth += 1
            else:  # Lose case
                wealth -= 1
            history.append(wealth)

        results.append(history)

    return results
```

```
# Parameters
initial_wealth = 1
target_wealth = 5
prob_win = 0.4
num_trials = 10

# Run simulation
results = simulate_game(initial_wealth, target_wealth, prob_win, num_trials)

# Visualize outcomes
plt.figure(figsize=(10, 6))
for trial in results:
    plt.plot(trial, label=f"Trial {results.index(trial)+1}")
plt.axhline(y=0, color='r', linestyle='--', label="Ruin (0)")
plt.axhline(y=target_wealth, color='g', linestyle='--', label="Target Wealth␣
 ↪(5)")
plt.title("Wealth Progression in Gambling Game")
plt.xlabel("Rounds")
plt.ylabel("Wealth")
plt.legend()
plt.show()

# Analyze results
wins = sum(1 for trial in results if trial[-1] == target_wealth)
loses = num_trials - wins
avg_rounds = sum(len(trial) - 1 for trial in results) / num_trials

print(f"Number of Wins (Reached 5): {wins}")
print(f"Number of Losses (Reached 0): {loses}")
print(f"Average Number of Rounds per Trial: {avg_rounds}")
```




```
Number of Wins (Reached 5): 1
Number of Losses (Reached 0): 9
Average Number of Rounds per Trial: 2.5
```

[ ]: