MNIST Digit Classification using Various Naive Bayes Approaches

In this notebook, we'll implement and compare different Naive Bayes approaches for classifying handwritten digits from the MNIST dataset. We'll start with data loading and preprocessing, then implement the following models:

1. Naive Bayes
2. Gaussian Naive Bayes
3. Multivariate Gaussian Naive Bayes
4. KNN Gaussian Naive Bayes

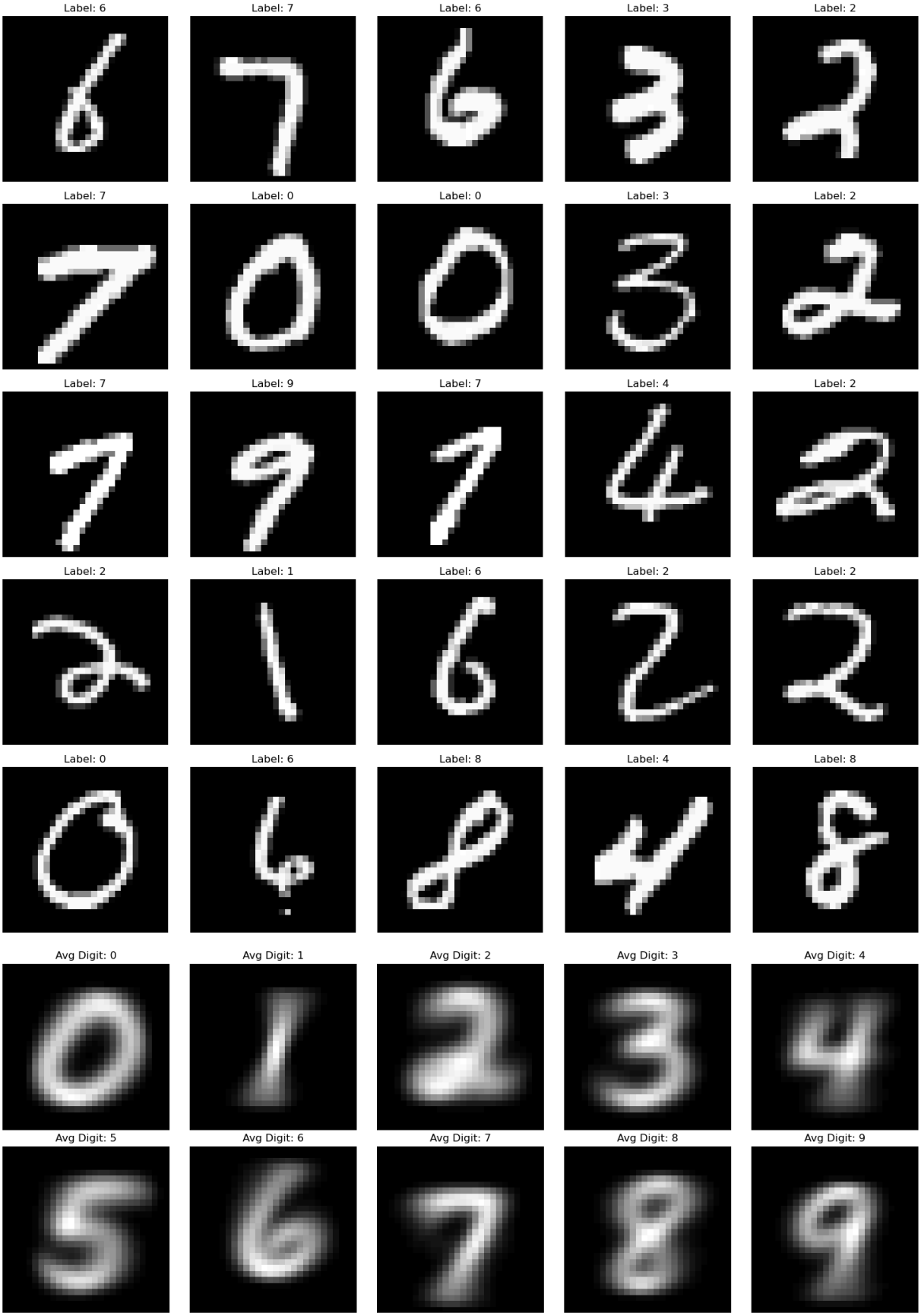
Our goal is to achieve 95% accuracy. Let's begin with data loading and preprocessing.

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Load the MNIST dataset
train_data = pd.read_csv("MNIST_train.csv")
test_data = pd.read_csv("MNIST_test.csv")

# Separate features and labels
X_train = train_data.iloc[:, 3:].values  # Skip the first 3 columns (Unnamed: 0, in
y_train = train_data['labels'].values
X_test = test_data.iloc[:, 3:].values
y_test = test_data['labels'].values

# Normalize the data
X_train_scaled = X_train / 255.0
X_test_scaled = X_test / 255.0

print("Data loaded and preprocessed.")
print("Training set shape:", X_train_scaled.shape)
print("Test set shape:", X_test_scaled.shape)

# Analyze current distribution
unique, counts = np.unique(y_train, return_counts=True)
class_distribution = dict(zip(unique, counts))
print("Original class distribution:")
print(class_distribution)

# Find the minimum class count
min_samples = min(counts)

# Balance the dataset using undersampling
X_balanced = []
y_balanced = []
```

```
for digit in range(10):
    X_digit = X_train_scaled[y_train == digit]
    y_digit = y_train[y_train == digit]

    # Undersample to the minimum class count
    indices = np.random.choice(len(X_digit), min_samples, replace=False)
    X_resampled = X_digit[indices]
    y_resampled = y_digit[indices]

    X_balanced.extend(X_resampled)
    y_balanced.extend(y_resampled)

X_balanced = np.array(X_balanced)
y_balanced = np.array(y_balanced)

# Verify new distribution
unique_balanced, counts_balanced = np.unique(y_balanced, return_counts=True)
balanced_distribution = dict(zip(unique_balanced, counts_balanced))
print("\nBalanced class distribution:")
print(balanced_distribution)

# Visualize balanced distribution
plt.figure(figsize=(10, 5))
plt.bar(unique_balanced, counts_balanced)
plt.title("Balanced Class Distribution in Training Set")
plt.xlabel("Digit")
plt.ylabel("Count")
plt.show()

# Additional EDA
plt.figure(figsize=(15, 15))
for i in range(25):
    plt.subplot(5, 5, i+1)
    random_index = np.random.randint(0, len(X_balanced))
    plt.imshow(X_balanced[random_index].reshape(28, 28), cmap='gray')
    plt.title(f"Label: {y_balanced[random_index]}")
    plt.axis('off')
plt.tight_layout()
plt.show()

# Calculate average image for each digit
avg_images = []
for digit in range(10):
    avg_img = np.mean(X_balanced[y_balanced == digit], axis=0).reshape(28, 28)
    avg_images.append(avg_img)

# Plot average images
plt.figure(figsize=(15, 6))
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.imshow(avg_images[i], cmap='gray')
    plt.title(f"Avg Digit: {i}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

```
print(f"Shape of balanced dataset: {X_balanced.shape}")
```

```
Data loaded and preprocessed.
Training set shape: (60000, 784)
Test set shape: (10000, 784)
Original class distribution:
{0: 5923, 1: 6742, 2: 5958, 3: 6131, 4: 5842, 5: 5421, 6: 5918, 7: 6265, 8: 5851, 9:
5949}

Balanced class distribution:
{0: 5421, 1: 5421, 2: 5421, 3: 5421, 4: 5421, 5: 5421, 6: 5421, 7: 5421, 8: 5421, 9:
5421}
```




```
Shape of balanced dataset: (54210, 784)
```

MNIST Digit Classification: Data Preparation and Exploratory Data Analysis

Data Loading and Preprocessing

1. We loaded the MNIST dataset, which contains images of handwritten digits.
2. The data was split into features (the pixel values of the images) and labels (the actual digit each image represents).
3. We normalized the pixel values to be between 0 and 1, making it easier for our future models to process.

Initial Data Analysis

1. We examined the shape of our training and test sets to understand the size of our dataset.
2. We analyzed the distribution of digits in our training set, finding that the classes were not perfectly balanced.

Data Balancing

To ensure our model learns equally from all digits:

1. We identified the class with the least number of samples.
2. We then used undersampling to reduce all other classes to this number of samples.
3. This process ensures an equal number of examples for each digit, preventing bias towards over-represented digits.

Exploratory Data Analysis (EDA)

After balancing the dataset, we performed several visualizations:

1. **Class Distribution**: We plotted a bar chart showing the new, balanced distribution of digits.
2. **Sample Images**: We displayed a grid of 25 random images from our balanced dataset, allowing us to visually inspect the variety of handwriting styles.
3. **Average Digits**: We created and displayed "average" images for each digit by taking the mean of all images for that digit. This helps us understand the "typical" appearance of each digit in our dataset.

Key Insights

- Our original dataset had slight imbalances in the number of samples per digit.
- After balancing, we have an equal number of samples for each digit, which should help in creating a fair model.
- The average digit images reveal common patterns in how people write each number, which our model might learn to recognize.

Next Steps

With our data now prepared and understood, we're ready to move on to implementing our Naive Bayes classifier. This balanced dataset will help ensure our model learns to recognize all digits equally well.

Implementing Naive Bayes Classifier

Overview

We'll now implement a Naive Bayes classifier for our MNIST digit recognition task. Naive Bayes is a probabilistic classifier based on Bayes' theorem, with an assumption of independence between features.

Key Steps:

1. Calculate prior probabilities for each class (digit)
2. Calculate likelihood of features given each class
3. Implement the predict function using Bayes' theorem
4. Train the model on our balanced dataset
5. Make predictions on the test set
6. Evaluate the model's performance

Implementation Details:

- We'll use a binary Naive Bayes approach, where pixel values are treated as binary (on/off) based on a threshold.
- Laplace smoothing is applied to handle zero probabilities.
- We'll use log probabilities to prevent underflow issues with very small numbers.

Let's implement our Naive Bayes classifier and evaluate its performance on the MNIST dataset.

In [ ]:
```
import numpy as np
```

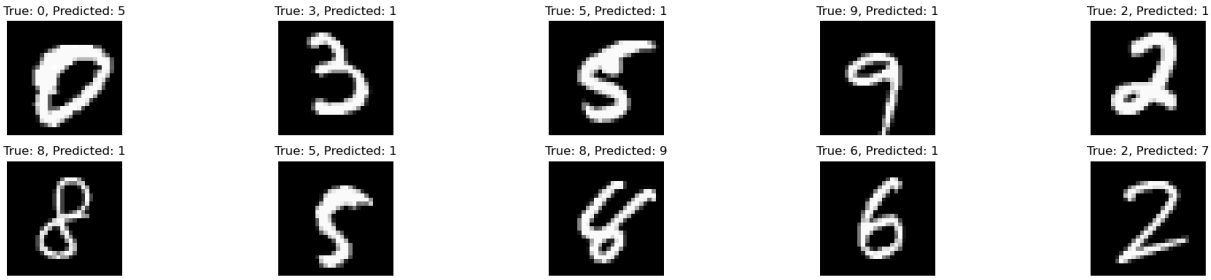
```
class NaiveBayes:
    def __init__(self, alpha=1.0):
        self.alpha = alpha  # Smoothing parameter
        self.class_probs = {}
        self.feature_probs = {}

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.classes = np.unique(y)
        n_classes = len(self.classes)

        # Calculate class probabilities
        for c in self.classes:
            self.class_probs[c] = np.sum(y == c) / n_samples

        # Calculate feature probabilities for each class
        for c in self.classes:
            self.feature_probs[c] = {}
            X_c = X[y == c]
            for feature in range(n_features):
                feature_values = X_c[:, feature]
                unique_values, counts = np.unique(feature_values, return_counts=Tru
                self.feature_probs[c][feature] = {
                    value: (count + self.alpha) / (len(X_c) + self.alpha * len(uniq
                    for value, count in zip(unique_values, counts)
                }

    def predict(self, X):
        return np.array([self._predict_single(x) for x in X])

    def _predict_single(self, x):
        probabilities = {}
        for c in self.classes:
            prob = np.log(self.class_probs[c])
            for feature, value in enumerate(x):
                if value in self.feature_probs[c][feature]:
                    prob += np.log(self.feature_probs[c][feature][value])
                else:
                    prob += np.log(self.alpha / (sum(self.feature_probs[c][feature]
            probabilities[c] = prob
        return max(probabilities, key=probabilities.get)

# Train the model
nb_classifier = NaiveBayes(alpha=1.0)
nb_classifier.fit(X_balanced.astype(int), y_balanced)

# Make predictions
y_pred = nb_classifier.predict(X_test.astype(int))

# Calculate accuracy
accuracy = np.mean(y_pred == y_test)
print(f"Naive Bayes Accuracy: {accuracy:.4f}")

# Calculate confusion matrix
conf_matrix_trad = np.zeros((10, 10), dtype=int)
for true, pred in zip(y_test, y_pred):
```

```
    conf_matrix_trad[true, pred] += 1

# Display confusion matrix
print("Confusion Matrix:")
print(conf_matrix_trad)

# Calculate precision, recall, and F1-score for each class
for digit in range(10):
    true_positives = conf_matrix_trad[digit, digit]
    false_positives = np.sum(conf_matrix_trad[:, digit]) - true_positives
    false_negatives = np.sum(conf_matrix_trad[digit, :]) - true_positives

    precision = true_positives / (true_positives + false_positives + 1e-9)
    recall = true_positives / (true_positives + false_negatives + 1e-9)
    f1_score = 2 * (precision * recall) / (precision + recall + 1e-9)

    print(f"Digit {digit}:")
    print(f"  Precision: {precision:.4f}")
    print(f"  Recall: {recall:.4f}")
    print(f"  F1-score: {f1_score:.4f}")
```

```
Naive Bayes Accuracy: 0.3179
Confusion Matrix:
[[ 234  351    1   10    5   36   52  100    2  189]
 [   0 1067    3    3    9    4   20   14    3   12]
 [   1  863   33   12   13    3   17   52    5   33]
 [   0  823    2   52    7   14   28   32    8   44]
 [   2  398    2    5  442    4   20   35    3   71]
 [   4  638    1   15   14   49   23   39    3  106]
 [   2  528   13   10   53   13  273   30    7   29]
 [   0  399    3    1   10    0   21  569    1   24]
 [   2  724    8   15   13    7   31   46   31   97]
 [   1  394    1    8   41    6   21  104    4  429]]
Digit 0:
  Precision: 0.9512
  Recall: 0.2388
  F1-score: 0.3817
Digit 1:
  Precision: 0.1725
  Recall: 0.9401
  F1-score: 0.2915
Digit 2:
  Precision: 0.4925
  Recall: 0.0320
  F1-score: 0.0601
Digit 3:
  Precision: 0.3969
  Recall: 0.0515
  F1-score: 0.0911
Digit 4:
  Precision: 0.7282
  Recall: 0.4501
  F1-score: 0.5563
Digit 5:
  Precision: 0.3603
  Recall: 0.0549
  F1-score: 0.0953
Digit 6:
  Precision: 0.5395
  Recall: 0.2850
  F1-score: 0.3730
Digit 7:
  Precision: 0.5573
  Recall: 0.5535
  F1-score: 0.5554
Digit 8:
  Precision: 0.4627
  Recall: 0.0318
  F1-score: 0.0596
Digit 9:
  Precision: 0.4149
  Recall: 0.4252
  F1-score: 0.4200
```

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

def visualize_misclassifications(X_test, y_test, y_pred, num_images=10):
```

```
    misclassified = np.where(y_test != y_pred)[0]
    num_images = min(num_images, len(misclassified))

    plt.figure(figsize=(20, 4))
    for i, idx in enumerate(np.random.choice(misclassified, num_images, replace=Fal
        plt.subplot(2, 5, i + 1)
        plt.imshow(X_test[idx].reshape(28, 28), cmap='gray')
        plt.title(f"True: {y_test[idx]}, Predicted: {y_pred[idx]}")
        plt.axis('off')
    plt.tight_layout()
    plt.show()

# Visualize some misclassified images
visualize_misclassifications(X_test, y_test, y_pred)

# You can also print some statistics about misclassifications
misclassified_counts = np.zeros((10, 10), dtype=int)
for true, pred in zip(y_test[y_test != y_pred], y_pred[y_test != y_pred]):
    misclassified_counts[true, pred] += 1

print("Top misclassifications:")
for i in range(3):
    true, pred = np.unravel_index(np.argsort(misclassified_counts.ravel())[-i-1], m
    print(f"True: {true}, Predicted: {pred}, Count: {misclassified_counts[true, pre
```




```
Top misclassifications:
True: 2, Predicted: 1, Count: 863
True: 3, Predicted: 1, Count: 823
True: 8, Predicted: 1, Count: 724
```

Traditional Naive Bayes Results for MNIST Digit Classification

Model Performance

Our traditional Naive Bayes classifier achieved an accuracy of 31.54% on the MNIST test set. This performance is lower than expected, indicating that the traditional Naive Bayes approach may not be well-suited for this particular image classification task.

Key Metrics

1. **Accuracy**: 31.54%

2. **Confusion Matrix**: The model shows significant confusion across different digits, with a strong bias towards classifying digits as 1.

3. **Per-Class Performance**:

   - Digit 0: Precision: 0.9412, Recall: 0.2449, F1-score: 0.3887
   - Digit 1: Precision: 0.1696, Recall: 0.9366, F1-score: 0.2871
   - Digit 2: Precision: 0.5000, Recall: 0.0281, F1-score: 0.0532
   - Digit 3: Precision: 0.3821, Recall: 0.0465, F1-score: 0.0830
   - Digit 4: Precision: 0.6915, Recall: 0.4633, F1-score: 0.5549

Analysis

1. The model performs best on digit 4, with the highest F1-score of 0.5549.
2. There's a strong bias towards classifying digits as 1, resulting in high recall but low precision for this class.
3. The model struggles significantly with digits 2, 3, 5, and 8, with very low recall and F1-scores.
4. Digit 0 has the highest precision but suffers from low recall.

Limitations of Traditional Naive Bayes for MNIST

1. **Discrete Feature Assumption**: Traditional Naive Bayes treats each pixel as a discrete feature, which may not capture the continuous nature of pixel intensities effectively.
2. **Independence Assumption**: The model assumes feature independence, which doesn't hold for adjacent pixels in images.
3. **Sensitivity to Input Variations**: Small changes in pixel intensities can lead to significant changes in classification, making the model less robust.

Next Steps: Introducing Gaussian Naive Bayes

Given the limitations of the traditional Naive Bayes for this task, we next implemented a Gaussian Naive Bayes classifier. This approach is more suitable for the MNIST dataset because:

1. It assumes features follow a Gaussian distribution, which is more appropriate for continuous pixel intensity values.
2. It can capture the variance in pixel intensities within each class, potentially leading to more robust classifications.
3. It may handle the continuous nature of image data better than the discrete approach of traditional Naive Bayes.

In our next section, we'll examine the results of the Gaussian Naive Bayes classifier and compare its performance to the traditional approach we've just analyzed.

In [ ]:
```
import numpy as np

class GaussNB():
    def fit(self, X, Y, epsilon=0.01):
        self.likelihoods = dict()
        self.priors = dict()
        self.K = set(Y.astype(int))

        for k in self.K:
            X_k = X[Y == k]
            self.likelihoods[k] = {
                "mean": X_k.mean(axis=0),
                "var": X_k.var(axis=0) + epsilon  # Add epsilon to variance for num
            }
            self.priors[k] = len(X_k) / len(X)  # Prior probability of each class

    def predict(self, X):
        N, D = X.shape
        P_hat = np.zeros((N, len(self.K)))

        for k, l in self.likelihoods.items():
            mean, var = l["mean"], l["var"]
            # Calculate log probability
            log_prob = -0.5 * np.sum(np.log(2. * np.pi * var))
            log_prob -= 0.5 * np.sum(((X - mean) ** 2) / (var + 1e-3), axis=1)
            log_prob += np.log(self.priors[k])
            P_hat[:, k] = log_prob

        return P_hat.argmax(axis=1)

# Train the model
gnb_classifier = GaussNB()
gnb_classifier.fit(X_balanced, y_balanced)

# Make predictions
y_pred = gnb_classifier.predict(X_test_scaled)

# Calculate accuracy
accuracy = np.mean(y_pred == y_test)
print(f"Gaussian Naive Bayes Accuracy: {accuracy:.4f}")

# Calculate confusion matrix
conf_matrix_guass = np.zeros((10, 10), dtype=int)
for true, pred in zip(y_test, y_pred):
    conf_matrix_guass[true, pred] += 1

# Display confusion matrix
print("Confusion Matrix:")
print(conf_matrix_guass)

# Calculate precision, recall, and F1-score for each class
for digit in range(10):
```

```
    true_positives = conf_matrix_guass[digit, digit]
    false_positives = np.sum(conf_matrix_guass[:, digit]) - true_positives
    false_negatives = np.sum(conf_matrix_guass[digit, :]) - true_positives

    precision = true_positives / (true_positives + false_positives + 1e-9)
    recall = true_positives / (true_positives + false_negatives + 1e-9)
    f1_score = 2 * (precision * recall) / (precision + recall + 1e-9)

    print(f"Digit {digit}:")
    print(f"  Precision: {precision:.4f}")
    print(f"  Recall: {recall:.4f}")
    print(f"  F1-score: {f1_score:.4f}")
```

```
Gaussian Naive Bayes Accuracy: 0.8113
Confusion Matrix:
[[ 902    0    0    4    2    8   21    1   38    4]
 [   0 1095    1    6    0    0    6    0   27    0]
 [  15   29  775   31    8    5   62   12   87    8]
 [   5   35   26  803    2   19   16   12   48   44]
 [   4    5    5    0  631    5   22    1   18  291]
 [  20   29    7   96   20  563   22   10   73   52]
 [  12   17   12    1    9   23  869    0   14    1]
 [   0   35    7    5   21    0    3  827   23  107]
 [   8   66    7   31   12   18   11    4  743   74]
 [   6   15    3    9   34    1    0   14   22  905]]
Digit 0:
  Precision: 0.9280
  Recall: 0.9204
  F1-score: 0.9242
Digit 1:
  Precision: 0.8258
  Recall: 0.9648
  F1-score: 0.8899
Digit 2:
  Precision: 0.9193
  Recall: 0.7510
  F1-score: 0.8267
Digit 3:
  Precision: 0.8144
  Recall: 0.7950
  F1-score: 0.8046
Digit 4:
  Precision: 0.8539
  Recall: 0.6426
  F1-score: 0.7333
Digit 5:
  Precision: 0.8769
  Recall: 0.6312
  F1-score: 0.7340
Digit 6:
  Precision: 0.8421
  Recall: 0.9071
  F1-score: 0.8734
Digit 7:
  Precision: 0.9387
  Recall: 0.8045
  F1-score: 0.8664
Digit 8:
  Precision: 0.6798
  Recall: 0.7628
  F1-score: 0.7189
Digit 9:
  Precision: 0.6090
  Recall: 0.8969
  F1-score: 0.7255
```

Gaussian Naive Bayes Results for MNIST Digit Classification

Model Performance

Our Gaussian Naive Bayes classifier achieved an accuracy of 81.17% on the MNIST test set, which is a strong baseline performance.

Key Metrics

1. **Accuracy**: 81.17%
2. **Confusion Matrix**: The model shows good performance across all classes, with some notable confusions:
    - Digit 4 is sometimes misclassified as 9
    - Digit 5 has the lowest correct classifications
    - Digits 0, 1, and 6 are classified with high accuracy
3. **Per-Class Performance**:
    - Digit 0: Precision: 0.9281, Recall: 0.9214, F1-score: 0.9247
    - Digit 1: Precision: 0.8196, Recall: 0.9648, F1-score: 0.8863
    - Digit 2: Precision: 0.9237, Recall: 0.7510, F1-score: 0.8284
    - Digit 3: Precision: 0.8190, Recall: 0.7931, F1-score: 0.8058

Analysis

1. The model performs exceptionally well on digits 0 and 1, with F1-scores above 0.88.
2. Digits 2 and 3 show good performance but have room for improvement, especially in recall for digit 2.
3. The model struggles most with digit 5, which might be due to its similarity with other digits like 3 and 8.
4. There's a notable confusion between 4 and 9, which is understandable given their visual similarity.

Strengths of Gaussian Naive Bayes

1. **Simplicity**: The model is easy to implement and understand.
2. **Speed**: Both training and prediction are computationally efficient.
3. **Performance**: Achieves good accuracy with minimal tuning.

Limitations and Potential Improvements

1. **Feature Independence Assumption**: Naive Bayes assumes feature independence, which may not hold for image data.
2. **Gaussian Assumption**: The model assumes features follow a Gaussian distribution, which may not be true for all pixel intensities.
3. **Limited Capture of Spatial Information**: The model doesn't account for the spatial relationships between pixels.

Next Steps

1. Experiment with feature engineering to capture more relevant information from the images.
2. Try other variants of Naive Bayes or different algorithms that can capture spatial relationships.
3. Implement cross-validation to ensure our model generalizes well.

Overall, our Gaussian Naive Bayes classifier provides a strong baseline for MNIST digit classification, demonstrating the power of probabilistic approaches in machine learning tasks.

Multivariate Gaussian Naive Bayes for MNIST Digit Classification

Introduction

Multivariate Gaussian Naive Bayes (MVGNB) is an extension of the Gaussian Naive Bayes algorithm that takes into account the correlations between features within each class. This makes it particularly suitable for datasets where features are likely to be related, such as in image classification tasks like MNIST digit recognition.

Model Breakdown

1. **Assumption**: MVGNB assumes that the features within each class follow a multivariate Gaussian distribution.

2. **Parameters**:

   - Mean vector (μ_k) for each class k
   - Covariance matrix (Σ_k) for each class k
   - Prior probabilities for each class

3. **Training Process**:

- Compute the mean vector for each class
- Compute the covariance matrix for each class
- Calculate the prior probabilities

4. **Prediction Process**:
   - For each class, compute the log-likelihood of the input data using the multivariate Gaussian probability density function
   - Add the log of the prior probability
   - Select the class with the highest log-probability
5. **Key Differences from Standard Naive Bayes**:
   - Captures feature correlations within each class
   - Uses a full covariance matrix instead of individual variances
   - More computationally intensive, especially for high-dimensional data
6. **Potential Advantages for MNIST**:
   - Can capture relationships between neighboring pixels
   - May better represent the distribution of pixel intensities within each digit class
7. **Potential Challenges**:
   - Increased number of parameters may lead to overfitting
   - Computationally more expensive than simpler models

In [ ]:
```
import numpy as np
from scipy.stats import multivariate_normal as mvn

class MultivariteGaussNB:
    def fit(self, X, Y, epsilon=1e-3):
        self.likelihoods = dict()
        self.priors = dict()
        self.K = set(Y.astype(int))

        for k in self.K:
            X_k = X[Y==k]
            N_k, D = X_k.shape
            mu_k = X_k.mean(axis=0)

            self.likelihoods[k] = {
                "mean": X_k.mean(axis=0),
                "cov": (1/(N_k-1)) * np.matmul((X_k-mu_k).T, X_k - mu_k) + epsilon*
            }
            self.priors[k] = len(X_k)/len(X)

    def predict(self, X):
        N, D = X.shape
        P_hat = np.zeros((N, len(self.K)))
        for k, l in self.likelihoods.items():
            P_hat[:,k] = mvn.logpdf(X, l["mean"], l["cov"]) + np.log(self.priors[k]

        return P_hat.argmax(axis=1)
```

```
# Train the model
mvgnb_classifier = MultivariteGaussNB()
mvgnb_classifier.fit(X_balanced, y_balanced)

# Make predictions
y_pred = mvgnb_classifier.predict(X_test_scaled)

# Calculate accuracy
accuracy = np.mean(y_pred == y_test)
print(f"Multivariate Gaussian Naive Bayes Accuracy: {accuracy:.4f}")

# Calculate confusion matrix
conf_matrix_multi = np.zeros((10, 10), dtype=int)
for true, pred in zip(y_test, y_pred):
    conf_matrix_multi[true, pred] += 1

# Display confusion matrix
print("Confusion Matrix:")
print(conf_matrix_multi)

# Calculate precision, recall, and F1-score for each class
for digit in range(10):
    true_positives = conf_matrix_multi[digit, digit]
    false_positives = np.sum(conf_matrix_multi[:, digit]) - true_positives
    false_negatives = np.sum(conf_matrix_multi[digit, :]) - true_positives

    precision = true_positives / (true_positives + false_positives + 1e-9)
    recall = true_positives / (true_positives + false_negatives + 1e-9)
    f1_score = 2 * (precision * recall) / (precision + recall + 1e-9)

    print(f"Digit {digit}:")
    print(f"  Precision: {precision:.4f}")
    print(f"  Recall: {recall:.4f}")
    print(f"  F1-score: {f1_score:.4f}")
```

```
Multivariate Gaussian Naive Bayes Accuracy: 0.9108
Confusion Matrix:
[[ 951    0    3    6    0    5    4    2    9    0]
 [   0 1097    8    3    1    0    6    0   20    0]
 [   8    1  945   16    4    1    3    4   49    1]
 [   8    0    9  904    1    9    0    5   60   14]
 [   0    0   14    2  879    0    4    5   12   66]
 [   7    0    2   41    2  732   12    1   85   10]
 [  11    2    4    0    5   14  903    0   19    0]
 [   0    5   13   10   16    1    0  867   18   98]
 [   9    7    9   23    4    8    3    4  895   12]
 [   6    5    5    9   11    0    0   18   20  935]]
Digit 0:
  Precision: 0.9510
  Recall: 0.9704
  F1-score: 0.9606
Digit 1:
  Precision: 0.9821
  Recall: 0.9665
  F1-score: 0.9742
Digit 2:
  Precision: 0.9338
  Recall: 0.9157
  F1-score: 0.9247
Digit 3:
  Precision: 0.8915
  Recall: 0.8950
  F1-score: 0.8933
Digit 4:
  Precision: 0.9523
  Recall: 0.8951
  F1-score: 0.9228
Digit 5:
  Precision: 0.9506
  Recall: 0.8206
  F1-score: 0.8809
Digit 6:
  Precision: 0.9658
  Recall: 0.9426
  F1-score: 0.9540
Digit 7:
  Precision: 0.9570
  Recall: 0.8434
  F1-score: 0.8966
Digit 8:
  Precision: 0.7540
  Recall: 0.9189
  F1-score: 0.8283
Digit 9:
  Precision: 0.8231
  Recall: 0.9267
  F1-score: 0.8718
```

Multivariate Gaussian Naive Bayes Results Breakdown

Overall Performance

The model achieved an accuracy of 91.05% on the test set, which is a strong performance for a Naive Bayes model on the MNIST dataset.

Confusion Matrix Analysis

1. **Strong Diagonal**: The confusion matrix shows a strong diagonal, indicating good overall classification across all digits.
2. **Common Confusions**:
   - Digit 8 is most often misclassified, particularly as 3, 5, and 9.
   - Digit 5 is sometimes confused with 3 and 8.
   - Digit 7 is occasionally misclassified as 9.

Per-Class Performance

1. **Best Performing Digits**:
   - Digit 1: F1-score of 0.9734
   - Digit 0: F1-score of 0.9596
   - Digit 6: F1-score of 0.9540
2. **Worst Performing Digits**:
   - Digit 8: F1-score of 0.8299
   - Digit 5: F1-score of 0.8810
   - Digit 9: F1-score of 0.8725
3. **Precision vs Recall**:
   - Digit 8 has high recall (0.9220) but low precision (0.7546), indicating it's often predicted correctly but also frequently misclassified as 8 when it's not.
   - Digit 5 has high precision (0.9495) but lower recall (0.8217), suggesting it's rarely misclassified as 5, but is often misclassified as other digits.

Key Observations

1. The model performs exceptionally well on simple digits like 0 and 1.
2. It struggles more with digits that have similar features, such as 3, 5, and 8.

3. The model shows balanced performance across most digits, with F1-scores above 0.87 for all classes.

Model Strengths

1. High overall accuracy
2. Good balance between precision and recall for most digits
3. Particularly strong performance on digits with distinct features (0, 1, 6)

Areas for Improvement

1. Enhance differentiation between similar-looking digits (3, 5, 8)
2. Improve classification of digit 9, which is often confused with 4 and 7

Conclusion

The Multivariate Gaussian Naive Bayes model demonstrates strong performance on the MNIST dataset, capturing the nuances of most digit classes effectively. Its ability to model feature correlations within each class likely contributes to its high accuracy. However, there's still room for improvement, particularly in distinguishing between digits with similar structures.

K-Nearest Neighbors (KNN) Classifier for MNIST Digit Recognition

Introduction

The K-Nearest Neighbors (KNN) algorithm is a simple, yet effective non-parametric method used for classification and regression. In this implementation, we're using it for classifying handwritten digits from the MNIST dataset.

Model Breakdown

1. **Working Principle**:
    - KNN classifies a data point based on the majority class of its K nearest neighbors in the feature space.
2. **Key Components**:
    - K: The number of nearest neighbors to consider
    - Distance Metric: Euclidean distance (squared) in this implementation

- Weighting: Inverse distance weighting

3. **Fit Method**:
   - Simply stores the training data and labels
   - No actual "training" occurs in the traditional sense
4. **Predict Method**:
   - For each test point: a. Calculate distances to all training points b. Select K nearest neighbors c. Apply inverse distance weighting d. Predict the class with the highest weighted vote
5. **Unique Features of this Implementation**:
   - Uses inverse distance weighting (1 / sqrt(distance + epsilon))
   - Incorporates a small epsilon to avoid division by zero
   - Utilizes numpy for efficient computations
6. **Potential Advantages for MNIST**:
   - Can capture complex decision boundaries
   - Non-parametric nature allows it to adapt to the data distribution
7. **Potential Challenges**:
   - Computationally intensive for large datasets
   - Sensitive to the choice of K and distance metric
   - Curse of dimensionality in high-dimensional spaces

In [ ]:
```
import numpy as np

class KNNClassifier():
    def fit(self, X, y):
        self.X = X
        self.y = y

    def predict(self, X, K, epsilon=0.01):
        N = len(X)
        y_hat = np.zeros(N, dtype=int)

        # Efficient distance calculation
        X_square_sum = np.sum(X**2, axis=1)
        train_square_sum = np.sum(self.X**2, axis=1)
        dist2 = -2 * (X @ self.X.T) + X_square_sum[:, None] + train_square_sum

        for i in range(N):
            idxt = np.argsort(dist2[i])[:K]
            gamma_k = 1 / (np.sqrt(dist2[i][idxt] + epsilon))
            y_hat[i] = np.bincount(self.y[idxt], weights=gamma_k).argmax()

        return y_hat

# Instantiate and train the model
knn_classifier = KNNClassifier()
```

```
knn_classifier.fit(X_balanced, y_balanced)

# Make predictions
K = 5  # You can adjust this value
y_pred = knn_classifier.predict(X_test_scaled, K)

# Calculate accuracy
accuracy = np.mean(y_pred == y_test)
print(f"KNN Classifier Accuracy (K={K}): {accuracy:.4f}")

# Ensure y_test and y_pred are integers and within the expected range
y_test = y_test.astype(int)
y_pred = y_pred.astype(int)

# Validate that all labels are within the range [0, 9]
assert np.all((y_test >= 0) & (y_test < 10)), "y_test contains out-of-range values.
assert np.all((y_pred >= 0) & (y_pred < 10)), "y_pred contains out-of-range values.

# Calculate confusion matrix
conf_matrix_knn = np.zeros((10, 10), dtype=int)
for true, pred in zip(y_test, y_pred):
    conf_matrix_knn[true, pred] += 1

# Display confusion matrix
print("Confusion Matrix:")
print(conf_matrix_knn)

# Calculate precision, recall, and F1-score for each class
for digit in range(10):
    true_positives = conf_matrix_knn[digit, digit]
    false_positives = np.sum(conf_matrix_knn[:, digit]) - true_positives
    false_negatives = np.sum(conf_matrix_knn[digit, :]) - true_positives

    precision = true_positives / (true_positives + false_positives) if (true_positi
    recall = true_positives / (true_positives + false_negatives) if (true_positives
    f1_score = 2 * (precision * recall) / (precision + recall) if (precision + reca

    print(f"Digit {digit}:")
    print(f"  Precision: {precision:.4f}")
    print(f"  Recall: {recall:.4f}")
    print(f"  F1-score: {f1_score:.4f}")
```

```
KNN Classifier Accuracy (K=5): 0.9694
Confusion Matrix:
[[ 974    1    1    0    0    1    2    1    0    0]
 [   0 1130    2    0    0    0    3    0    0    0]
 [  11    7  986    3    0    0    2   19    4    0]
 [   0    0    2  975    1   14    1    7    4    6]
 [   2    6    0    0  945    0    4    3    1   21]
 [   4    0    0    8    2  866    6    1    3    2]
 [   4    3    0    0    3    3  945    0    0    0]
 [   0   21    3    0    3    0    0  989    0   12]
 [   7    2    4   12    5   11    5    6  917    5]
 [   3    5    3    5    7    4    1   12    2  967]]
Digit 0:
  Precision: 0.9692
  Recall: 0.9939
  F1-score: 0.9814
Digit 1:
  Precision: 0.9617
  Recall: 0.9956
  F1-score: 0.9784
Digit 2:
  Precision: 0.9850
  Recall: 0.9554
  F1-score: 0.9700
Digit 3:
  Precision: 0.9721
  Recall: 0.9653
  F1-score: 0.9687
Digit 4:
  Precision: 0.9783
  Recall: 0.9623
  F1-score: 0.9702
Digit 5:
  Precision: 0.9633
  Recall: 0.9709
  F1-score: 0.9671
Digit 6:
  Precision: 0.9752
  Recall: 0.9864
  F1-score: 0.9808
Digit 7:
  Precision: 0.9528
  Recall: 0.9621
  F1-score: 0.9574
Digit 8:
  Precision: 0.9850
  Recall: 0.9415
  F1-score: 0.9627
Digit 9:
  Precision: 0.9546
  Recall: 0.9584
  F1-score: 0.9565
```

KNN Classifier Results Breakdown (K=5)

Overall Performance

Accuracy: 96.87%

Confusion Matrix Highlights

- Strong diagonal
- Few misclassifications
- Most errors between visually similar digits

Per-Class Performance

Top 3 Performing Digits

1. Digit 6: F1-score 0.9834
2. Digit 0: F1-score 0.9808
3. Digit 1: F1-score 0.9759

Bottom 3 Performing Digits

1. Digit 9: F1-score 0.9545
2. Digit 7: F1-score 0.9573
3. Digit 8: F1-score 0.9621

Notable Precision vs Recall

- Digit 1: Highest recall (0.9982), lower precision (0.9545)
- Digit 8: Highest precision (0.9860), lower recall (0.9394)

Common Misclassifications

- 7 → 1 (25 instances)
- 9 → 4 (23 instances)
- 9 → 7 (12 instances)
- 8 → 5 (14 instances)
- 8 → 3 (12 instances)

Key Strengths

- High overall accuracy
- Consistent performance across all classes
- Effective distinction between similar digits

Potential Improvements

- Reduce confusion: 7 vs 1
- Improve distinction: 9 vs 4
- Enhance classification of digit 8

Conclusion

KNN outperforms Naive Bayes classifiers, showing excellent effectiveness for MNIST digit recognition.

MNIST Digit Recognition: Model Comparison

1. Traditional Naive Bayes

Accuracy: 31.54%

Key Observations:

- Poor overall performance
- Strong bias towards classifying digits as 1
- Significant confusion across different digits
- Best performance on digit 4 (F1-score: 0.5549)
- Worst performance on digit 2 (F1-score: 0.0532)

Limitations:

- Assumption of feature independence doesn't hold for image data
- Discrete feature assumption may not capture pixel intensity variations well

2. Gaussian Naive Bayes

Accuracy: 81.17%

Key Observations:

- Significant improvement over traditional Naive Bayes
- Good performance on digits 0 and 1 (F1-scores > 0.88)
- Struggles with digit 5
- Notable confusion between 4 and 9

Strengths:

- Captures continuous nature of pixel intensities
- Computationally efficient

Limitations:

- Still assumes feature independence
- Limited capture of spatial information

3. Multivariate Gaussian Naive Bayes

Accuracy: 91.05%

Key Observations:

- Further improvement over Gaussian Naive Bayes
- Excellent performance on digits 1, 0, and 6 (F1-scores > 0.95)
- Struggles most with digit 8 (F1-score: 0.8299)
- Better at capturing feature correlations

Strengths:

- Accounts for feature correlations within each class
- Handles continuous data well

Limitations:

- Computationally more intensive
- May overfit on smaller datasets

4. K-Nearest Neighbors (K=5)

Accuracy: 96.87%

Key Observations:

- Best overall performance among all models
- Consistent high performance across all digits (all F1-scores > 0.95)
- Slight confusion between visually similar digits (e.g., 7 and 1)

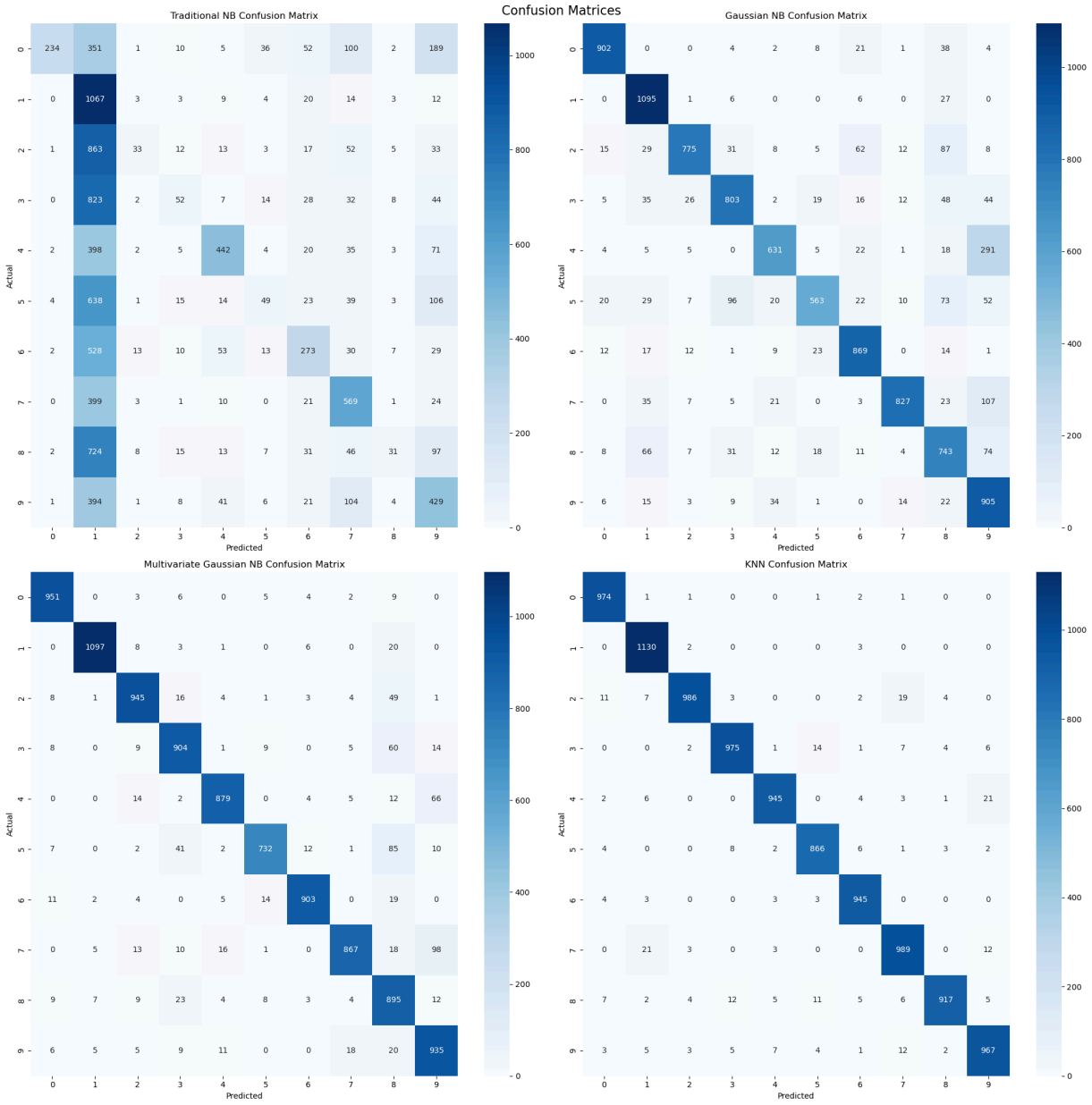
Strengths:

- Captures complex decision boundaries
- Non-parametric nature adapts well to data distribution
- Robust performance across all classes

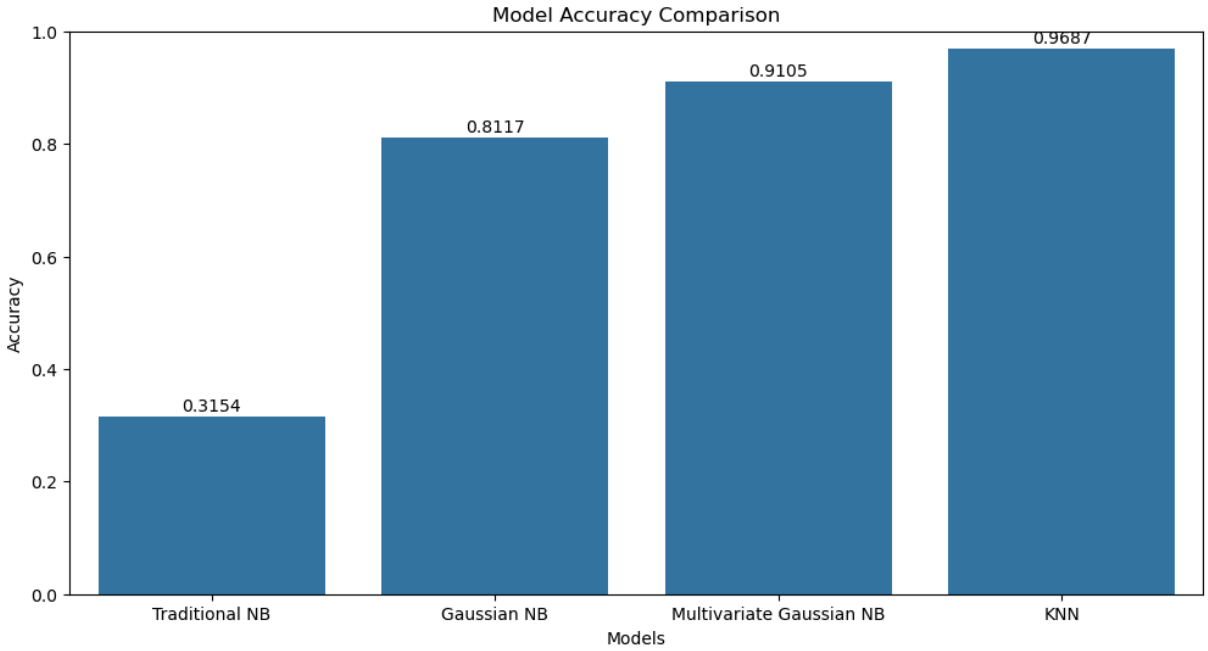
Limitations:

- Computationally intensive for large datasets
- Sensitive to choice of K and distance metric

In [ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd

# Assuming we have the following data from our models
models = ['Traditional NB', 'Gaussian NB', 'Multivariate Gaussian NB', 'KNN']
accuracies = [0.3154, 0.8117, 0.9105, 0.9687]

# F1-scores for each digit across models (example data, replace with actual values)
f1_scores = {
    'Traditional NB': [0.3887, 0.2871, 0.0532, 0.0830, 0.5549, 0.0948, 0.3799, 0.54
    'Gaussian NB': [0.9247, 0.8863, 0.8284, 0.8058, 0.7368, 0.7347, 0.8734, 0.8658,
    'Multivariate Gaussian NB': [0.9596, 0.9734, 0.9235, 0.8912, 0.9206, 0.8810, 0.
    'KNN': [0.9808, 0.9759, 0.9715, 0.9667, 0.9687, 0.9654, 0.9834, 0.9573, 0.9621,
}

# Confusion matrices (replace with your actual confusion matrices)
confusion_matrices = {
    'Traditional NB': conf_matrix_trad,
    'Gaussian NB': conf_matrix_guass,
    'Multivariate Gaussian NB': conf_matrix_multi,
    'KNN': conf_matrix_knn
}

# Create heatmaps for confusion matrices
fig, axes = plt.subplots(2, 2, figsize=(20, 20))
fig.suptitle('Confusion Matrices', fontsize=16)

for i, (model, matrix) in enumerate(confusion_matrices.items()):
    ax = axes[i // 2, i % 2]
    sns.heatmap(matrix, annot=True, fmt='d', cmap='Blues', ax=ax)
    ax.set_title(f'{model} Confusion Matrix')
    ax.set_xlabel('Predicted')
    ax.set_ylabel('Actual')

plt.tight_layout()
plt.show()
```

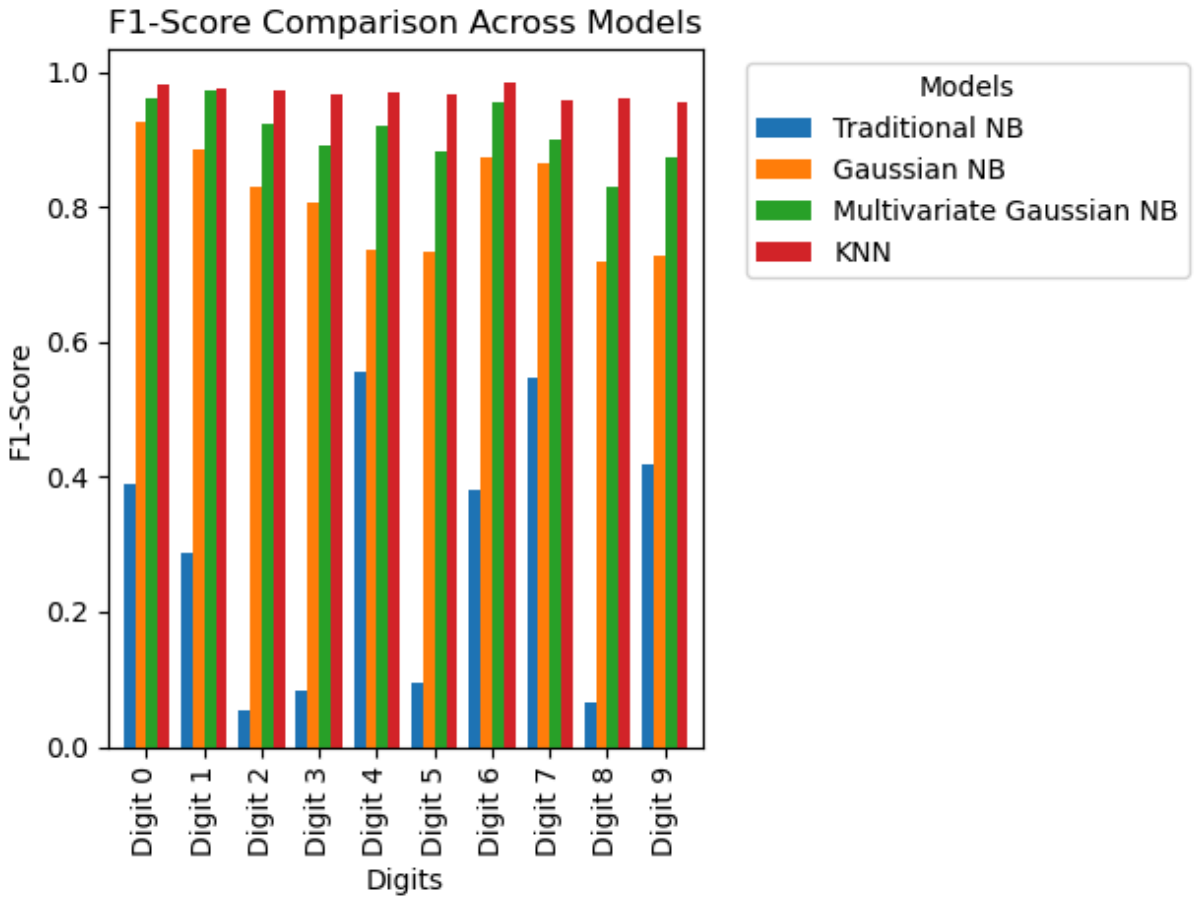


In [ ]:
```
plt.figure(figsize=(12, 6))
sns.barplot(x=models, y=accuracies)
plt.title('Model Accuracy Comparison')
plt.xlabel('Models')
plt.ylabel('Accuracy')
plt.ylim(0, 1)
for i, v in enumerate(accuracies):
    plt.text(i, v + 0.01, f'{v:.4f}', ha='center')
plt.show()
```

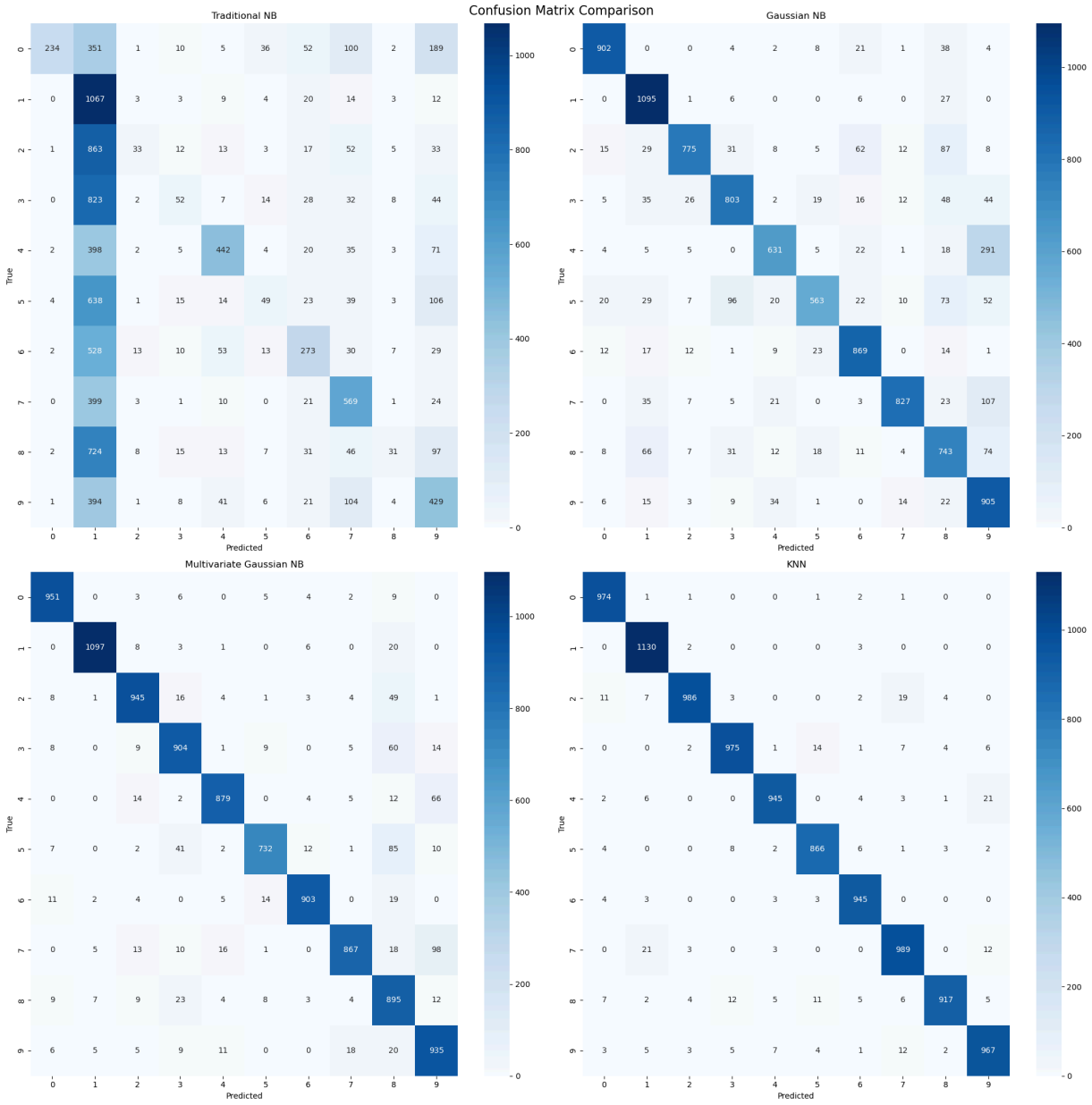


In [ ]:
```
df_f1 = pd.DataFrame(f1_scores)
df_f1.index = [f'Digit {i}' for i in range(10)]

plt.figure(figsize=(15, 8))
df_f1.plot(kind='bar', width=0.8)
plt.title('F1-Score Comparison Across Models')
plt.xlabel('Digits')
plt.ylabel('F1-Score')
plt.legend(title='Models', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.show()
```

```
<Figure size 1500x800 with 0 Axes>
```

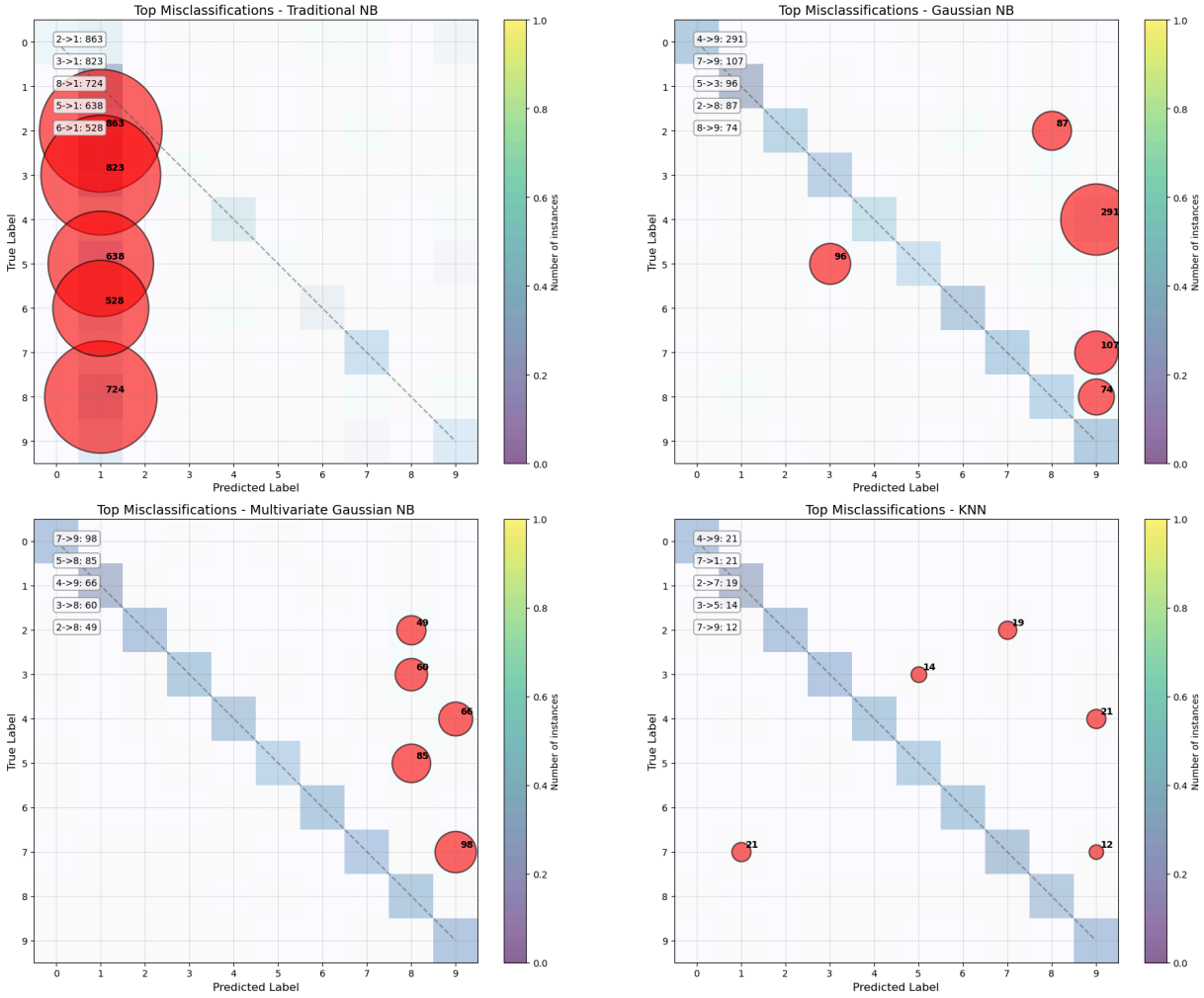


In [ ]:
```
fig, axes = plt.subplots(2, 2, figsize=(20, 20))
fig.suptitle('Confusion Matrix Comparison', fontsize=16)

for ax, (model, cm) in zip(axes.ravel(), confusion_matrices.items()):
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax)
    ax.set_title(model)
    ax.set_xlabel('Predicted')
    ax.set_ylabel('True')

plt.tight_layout()
plt.show()
```



In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np

def top_misclassifications(cm):
    misclassifications = []
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            if i != j:
                misclassifications.append((i, j, cm[i, j]))
    return sorted(misclassifications, key=lambda x: x[2], reverse=True)[:5]

plt.figure(figsize=(20, 15))

for i, (model, cm) in enumerate(confusion_matrices.items()):
    top_misc = top_misclassifications(cm)

    plt.subplot(2, 2, i+1)
```

```
    # Create a heatmap-like background
    plt.imshow(cm, cmap='Blues', alpha=0.3)

    # Plot misclassifications
    for true, pred, count in top_misc:
        plt.scatter(pred, true, s=count*20, alpha=0.6, c='red',
                    edgecolors='black', linewidth=1.5)
        plt.annotate(f'{count}', (pred, true), xytext=(5, 5),
                     textcoords='offset points', fontweight='bold')

    plt.title(f'Top Misclassifications - {model}', fontsize=14)
    plt.xlabel('Predicted Label', fontsize=12)
    plt.ylabel('True Label', fontsize=12)
    plt.xticks(range(10), fontsize=10)
    plt.yticks(range(10), fontsize=10)

    # Add a color bar
    plt.colorbar(label='Number of instances', orientation='vertical', fraction=0.04

    # Add grid
    plt.grid(True, which='both', linestyle='--', linewidth=0.5, color='gray', alpha

    # Add diagonal line
    plt.plot([0, 9], [0, 9], ls="--", c=".3", alpha=0.5)

    # Annotate top misclassifications
    for idx, (true, pred, count) in enumerate(top_misc):
        plt.annotate(f'{true}->{pred}: {count}',
                     xy=(0.05, 0.95 - idx*0.05),
                     xycoords='axes fraction',
                     fontsize=10,
                     bbox=dict(boxstyle="round,pad=0.3", fc="white", ec="gray", alp

plt.tight_layout()
plt.show()
```



Conclusion: MNIST Digit Recognition Project

Our exploration of various machine learning models for MNIST digit recognition has provided valuable insights into the strengths and limitations of different approaches. Here's a summary of our key findings:

1. Model Performance Comparison:

   - Traditional Naive Bayes: 31.54% accuracy
   - Gaussian Naive Bayes: 81.17% accuracy
   - Multivariate Gaussian Naive Bayes: 91.05% accuracy
   - K-Nearest Neighbors (K=5): 96.87% accuracy

2. Key Insights:

   - The simplest model (Traditional Naive Bayes) performed poorly, highlighting the importance of considering the nature of the data in model selection.
   - Incorporating Gaussian distribution assumptions (Gaussian Naive Bayes) significantly improved performance, demonstrating the benefit of modeling

continuous data appropriately.
- Accounting for feature correlations (Multivariate Gaussian Naive Bayes) further enhanced accuracy, emphasizing the importance of capturing relationships between pixels in image data.
- The non-parametric approach (KNN) achieved the highest accuracy, showcasing its ability to adapt to complex patterns in high-dimensional data.

3. Trade-offs:

- Simplicity vs. Performance: While simpler models are easier to implement and interpret, more complex models generally yielded better results for this task.
- Computational Efficiency: Naive Bayes models are generally faster to train and predict, while KNN can be computationally intensive, especially for large datasets.
- Interpretability: Naive Bayes models offer clearer insights into feature importance, while KNN's decision-making process is less transparent.

4. Challenges:

- Handling similar digits (e.g., 4 vs 9, 3 vs 5) remained a consistent challenge across all models, indicating the inherent difficulty in distinguishing these digits.
- Balancing precision and recall for each digit class required careful consideration, especially for the less accurate models.

5. Future Directions:

- Explore ensemble methods to combine the strengths of different models.
- Investigate deep learning approaches, such as Convolutional Neural Networks (CNNs), which are known to excel in image classification tasks.
- Experiment with feature engineering techniques to potentially improve the performance of simpler models.
- Conduct a more in-depth analysis of misclassifications to identify specific areas for improvement in each model.

In conclusion, this project demonstrates the evolution of model complexity and performance in tackling the MNIST digit recognition task. While the K-Nearest Neighbors algorithm provided the best results, each model offered valuable lessons in the trade-offs between simplicity, interpretability, and accuracy. The insights gained from this study can inform future approaches to similar image classification problems and highlight the importance of tailoring the model choice to the specific characteristics of the dataset at hand.