

✓ The MNIST Dataset: Handwritten Digit Recognition with a Naive Bayes Classifier

✓ 1. Importing necessary libraries

First, we need to import the libraries required for our analysis: `numpy` and `pandas` for data manipulation, `tensorflow` for loading the MNIST dataset, `sklearn` for building the Naive Bayes Classifier and evaluating its performance, and `matplotlib` and `seaborn` for creating a confusion matrix heatmap.

```
import numpy as np
import pandas as pd
from tensorflow.keras.datasets import mnist
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import seaborn as sns
```

✓ 2. Loading the dataset

We use the `keras` library to load the MNIST dataset, which is a well-known collection of 70,000 handwritten digits (0-9). The dataset is split into training and testing sets, with the training set containing images and their corresponding labels.


```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

 Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>
11490434/11490434 ————— 0s 0us/step

To speed up processing, we select only the first 500 samples from the training dataset. This allows us to work with a manageable amount of data while still maintaining enough variety for training.

```
X_train_small = X_train[:500]
y_train_small = y_train[:500]

# Displaying the shape of the data to confirm the size
print(f'Training Features shape: {X_train_small.shape}')
print(f'Training Labels shape: {y_train_small.shape}')
```

 Training Features shape: (500, 28, 28)
Training Labels shape: (500,)

✓ 3. Preprocessing the data

The images in the dataset are initially in a 3D format (number of samples, height, width). We flatten each image from 28×28 pixels to a 1D array of 784 pixels, making it suitable for input into the Naive Bayes Classifier.

```
X_train_small_flat = X_train_small.reshape(-1, 28 * 28)
X_test_flat = X_test.reshape(-1, 28 * 28)
```

✓ 4. Splitting the data into training and testing sets



We further split our smaller training set into a new training set and a validation set (20%) using `train_test_split`. This allows us to evaluate the model's performance on data it hasn't seen during training.

```
X_train_final, X_val, y_train_final, y_val = train_test_split(X_train_small_flat, y_train_small, test_size=0.2, random_state=42)
```

✓ 5. Creating and training the Naive Bayes Classifier

In this section, we set up a Gaussian Naive Bayes Classifier, which is effective for continuous feature data like pixel values. The model is trained using the new training set.

```
gnb = GaussianNB()
gnb.fit(X_train_final, y_train_final)
```

  GaussianNB ⓘ ?
GaussianNB()

6. Making predictions

After training, we use the model to predict labels for the validation set. This gives us an idea of how well the model is performing.

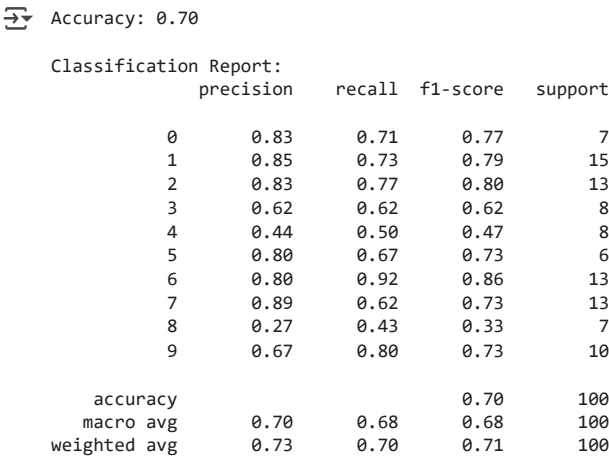
```
y_pred = gnb.predict(X_val)
```

7. Evaluating the model

Now, we calculate the accuracy of the model and print a detailed classification report, which includes precision, recall, and F1-score for each digit class, providing insight into the model's performance.

```
accuracy = accuracy_score(y_val, y_pred)
print(f'Accuracy: {accuracy:.2f}')

print("\nClassification Report:")
print(classification_report(y_val, y_pred))
```



8. Generating the confusion matrix

Finally, we visualise the confusion matrix, which shows how many instances of each digit were correctly or incorrectly classified. This helps us understand specific areas where the model may be confusing certain digits.

```
conf_matrix = confusion_matrix(y_val, y_pred)

plt.figure(figsize=(4,4))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=range(10), yticklabels=range(10))
plt.title('Confusion Matrix Heatmap')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

