SRH University Heidelberg

Task 1:
Handwritten Digits classification based on the Gaussian naive Bayes classifier in Machine Learning

Author: Nikhil Kumar
Matriculation Number: 11037894
Email Address: nikhil.kumar@stud.hochschule-heidelberg.de
Date of Submission: 29th April 2024

Under the Guidance of Professor
Dr. Ing Milan Gnjatovic

Aim of the Task:

The aim of this project is to develop a program for handwritten digit classification using the Gaussian naive Bayes approach applied to the MNIST dataset. This involves training a model on the MNIST dataset to classify handwritten digits and evaluating its performance based on accuracy and other relevant metrics. The project will delve into the implementation details, results analysis, and potential improvements of the Gaussian naive Bayes classifier for digit recognition.

Objective :

The objective of this project is to implement a classification system for handwritten digits using the Gaussian naive Bayes approach, specifically tailored for the MNIST dataset. The primary goals include developing a robust and efficient model capable of accurately identifying handwritten digits (0-9) based on their pixel intensity values. This involves preprocessing the MNIST dataset, training the Gaussian naive Bayes classifier, and evaluating its performance in terms of accuracy, precision, recall, and F1-score. Through this endeavor, a comprehensive understanding of applying probabilistic machine learning methods to image classification tasks will be gained, specifically in the context of handwritten digit recognition using the MNIST dataset.

Introduction :

Handwritten digit classification is a fundamental problem in the field of pattern recognition and machine learning, with numerous real-world applications ranging from automated postal sorting to optical character recognition. The MNIST dataset, a widely recognized benchmark in the machine learning community, consists of a large collection of grayscale images of handwritten digits.

The Gaussian naive Bayes classifier is a probabilistic model based on Bayes' theorem with the assumption of feature independence given the class label. Despite its simplistic assumptions, Gaussian naive Bayes has demonstrated effectiveness in various classification tasks, particularly when dealing with continuous-valued features like pixel intensities. This project aims to explore the applicability of Gaussian naive Bayes to the MNIST dataset, leveraging its simplicity and efficiency for handwritten digit recognition. The report will delve into the methodology behind preprocessing the dataset, training the classifier, evaluating its performance metrics, and discussing insights gained from the experiment.

What is Hand digit classification?

Handwritten digit classification in machine learning refers to the task of automatically recognizing and categorizing handwritten digits into their respective classes ,typically digits from 0 to 9. This task falls under the broader category of image classification, where the goal is to train a model to correctly identify and classify images based on their visual features.

The process of handwritten digit classification involves the following steps:

- **Data Collection and Preprocessing:** Acquiring a dataset of handwritten digit images, such as the MNIST dataset, which consists of thousands of grayscale images of digits. The images are usually preprocessed to standardize size, orientation, and intensity, making them suitable for analysis.

- **Feature Extraction:** Extracting meaningful features from the digit images that can be used by the machine learning model for classification. In the case of handwritten digits, common features include pixel intensities derived from image processing techniques.

- **Model Training:** Using a machine learning algorithm to train a classification model on a labeled dataset. Popular algorithms for this task include neural networks like Gaussian naive Bayes classifiers, and convolutional neural networks – CNNs.

- **Model Evaluation:** Assessing the performance of the trained model using evaluation metrics such as accuracy, precision, recall, and F1-score. The model is typically evaluated on a separate test set to measure its generalization ability.

Why is Hand digit classification Performed?

Handwritten digit classification is performed for several reasons:

- **Optical Character Recognition (OCR):** Handwritten digit classification is a subset of OCR, which aims to convert handwritten or printed text into machine-readable text. Classifying digits accurately is essential for OCR systems used in various applications, such as digitizing historical documents or processing bank checks.

- **Automation:** Handwritten digit classification enables automation of tasks that involve interpreting handwritten information, such as reading postal codes on mail envelopes or processing handwritten forms. Automation reduces the need for manual intervention and speeds up processes.

- **Postal Mail Sorting:** Handwritten digit classification is used in Zip Code recognition where it automatically recognizes and classifies the zip codes on mail envelopes. This helps in sorting mail efficiently based on destination, enabling faster and accurate delivery. It is also used in Address Parsing , postal services can extract relevant information like street numbers and house numbers for routing and delivery purposes, reducing manual sorting efforts.

- **Bank Check Processing:** Handwritten digits on checks often contain important routing numbers like account numbers, IFSC codes, and SWIFT code , so classification of these digits assists in electronic fund transfers and reconciliation processes.

- **Recommendation Systems:** There are a total of two recommendation systems, content-based and collaborative filtering. The naive Bayes with collaborative filtering-based models is known for their best accuracy on recommendation problems. The naive Bayes algorithms help achieve better accuracies for recommending features to the users based on their interests and related to other users' interests.

Naive Bayes Classifier Algorithm:

- Naive Bayes algorithm is a supervised learning algorithm, which is based on **Bayes theorem** and used for solving classification problems.
- It is mainly used in text classification that includes a high-dimensional training dataset.
- Naive Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions.
- It is a probabilistic classifier, which means it predicts on the basis of the probability of an object that is trained.
- Some popular examples of Naive Bayes Algorithm are spam filtration, Sentimental analysis, and classifying articles.[1a]

Why is it called Naive Bayes?

The Naive Bayes algorithm is comprised of two words Naive and Bayes, which can be described as: [1a]

Naive: It is called Naive because it assumes that the occurrence of a certain feature is independent of the occurrence of other features. Such as if the fruit is identified on the bases of color, shape, and taste, then red, spherical, and sweet fruit is recognized as an apple. Hence each feature individually contributes to identify that it is an apple without depending on each other. [1a]

Bayes: It is called Bayes because it depends on the principle of Bayes' Theorem. [1a]

Bayes' Theorem:

Bayes' theorem is also known as Bayes' Rule or Bayes' law, which is used to determine the probability of a hypothesis with prior knowledge. It depends on the conditional probability. [1b].

The formula for Bayes' theorem is given as:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} ,$$

Where,
P(A|B) is referred as to the posterior probability.
P(B|A) is referred as to the likelihood.
P(A) is referred as to the prior probability.
P(B) $\neq 0$. [1b]

Types of Naive Bayes:

There are mainly a total of three types of naive byes algorithms. Different types of naive Bayes are used for different use cases.[1c]

1. **Bernoulli Naive Bayes :-**
   This Naive Bayes Classifier is used when there is a Boolean type of dependent or target variable present in the dataset. For example, a dataset has target column categories as Yes and No. This type of Naive is mainly used in a binary categorical targeted column where the problem statement is to predict only Yes or No. [1c]

2. **Multinomial Naive Bayes :-**
   This type of naive Bayes is used where the data is multinomial distributed. This type of naive Bayes is mainly used when there is a text classification problem. For Example, if you want to predict whether a text belongs to which tag, education, politics, e-tech, or some other tag, you can use the multinomial Naive Bayes Classifier to classify the same. [1c]

3. **Gaussian Naive Bayes :-**
   Gaussian Naive Bayes is a popular classification algorithm that is based on Bayes' theorem, with an assumption that the features follow a normal (Gaussian) distribution. It is a variant of the Naive Bayes algorithm and is particularly useful when dealing with continuous data. [1b]

The likelihood $P(F_i = f_{ij} |c)$ is estimated as:

$$P(F_i = f_{ij}|c) = \frac{1}{\sqrt{2\pi\sigma_{ic}^2}} e^{-\frac{1}{2}\left(\frac{f_{ij} - \mu_{ic}}{\sigma_{ic}}\right)^2}$$ [1b]

where:
μ_{ic} is the mean of the values of attribute F_i associated with class c, i.e.:

$$\mu_{ic} = \frac{1}{n_c} \sum_{j=1}^{n_c} f_{ij}$$ [1b]

σic is the standard deviation of the values of attribute Fi associated with class c, i.e.:

$$\sigma_{ic} = \sqrt{\frac{1}{n_c - 1}\sum_{j=1}^{n_c}(f_{ij} - \mu_i)^2}$$

[1b]

MNIST Dataset:

MNIST dataset is a dataset of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau. Each image is labeled with the digit it represents. This set has been studied so much that it is often called the "Hello World" of Machine Learning: whenever people come up with a new classification algorithm, they are curious to see how it will perform on MNIST. Whenever someone learns Machine Learning, sooner or later they tackle MNIST. There are 70,000 images, and each image has 784 features. This is because each image is 28×28 pixels, and each feature simply represents one pixel's intensity, from 0 (white) to 255 (black). [1d]

[1d]

The above image shows some digits from the MNIST dataset.

Always make a test set and put it aside before carefully going over the data. The MNIST dataset is actually already divided into two sets: and the first 60,000 images are a training set and the last 10,000 images are a test set. [1d]

Evaluation Measures:

Evaluation measures for Gaussian Naive Bayes (GNB) on the MNIST dataset are important for assessing the performance of the classifier in a multi-class classification task, such as recognizing handwritten digits. Here are the evaluation measures commonly used for GNB on the MNIST dataset:

1. **Confusion Matrix :** A much better way to evaluate the performance of a classifier is to look at the confusion matrix. The confusion matrix is a tabular representation that summarizes the performance of a classification model by presenting the counts of true positive, true negative, false positive, and false negative predictions for each class. The general idea is to count the number of times instances of class A are classified as class B. Each row in a confusion matrix represents an actual class, while each column represents a predicted class. For example, to know the number of times the classifier confused images of 5s with 3s, you would look in the 5th row and 3rd column of the confusion matrix. A perfect classifier would have only true positives and true negatives, so its confusion matrix would have nonzero values only on its main diagonal (top left to bottom right) [1d]

2. **Accuracy :** Accuracy measures the proportion of correctly classified samples out of the total number of samples. It is calculated as:
   Accuracy = Number of correct predictions/ Total Number of predictions.

3. **Precession :** You can find a lot of information in the confusion matrix, but occasionally you might want a metric that is shorter. An interesting one to look at is the accuracy of the positive predictions; this is called the precision of the classifier. [1d]
   Precision measures the ability of the classifier to correctly identify positive samples (true positives) out of all samples predicted as positive. It is calculated as:

$$P = \frac{\text{true positive}}{\text{true positive} + \text{false positive}} .$$ [1b]

   Precision is useful when the cost of false positives is high.

4. **Recall :** Recall measures the ability of the classifier to correctly identify positive samples (true positives) out of all actual positive samples. It is calculated as:

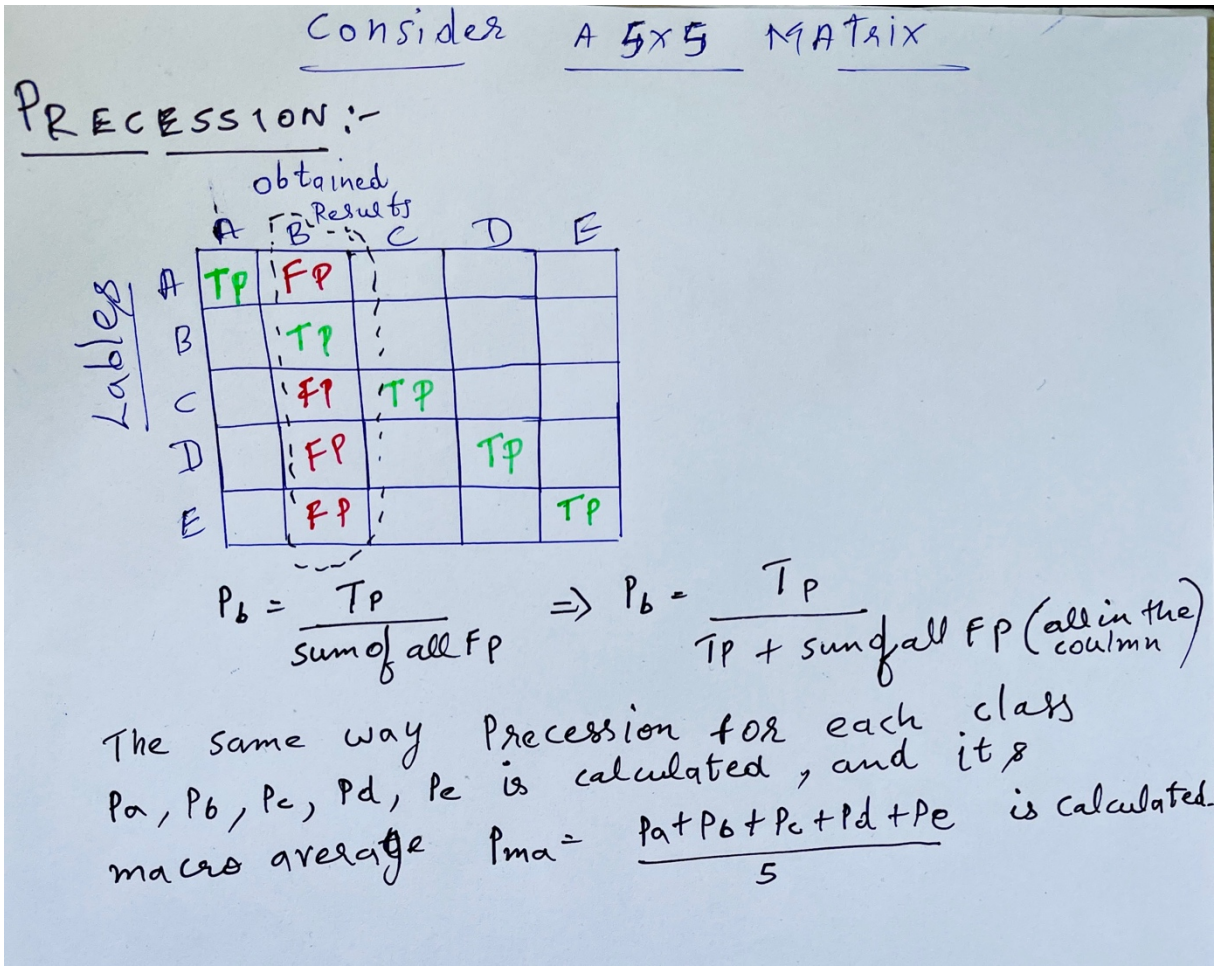
$$R = \frac{\text{true positive}}{\text{true positive} + \text{false negative}} .$$ [1b]

5. **F1 Score** : It is often convenient to combine precision and recall into a single metric called the F1 score, in particular if you need a simple way to compare two classifiers. The F1 score is the harmonic mean of precision and recall .Whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values. As a result, the classifier will only get a high F1 score if both recall and precision are high.[1d]
   This Harmonic mean provides a balance between the two metrics.
   It is calculated as:

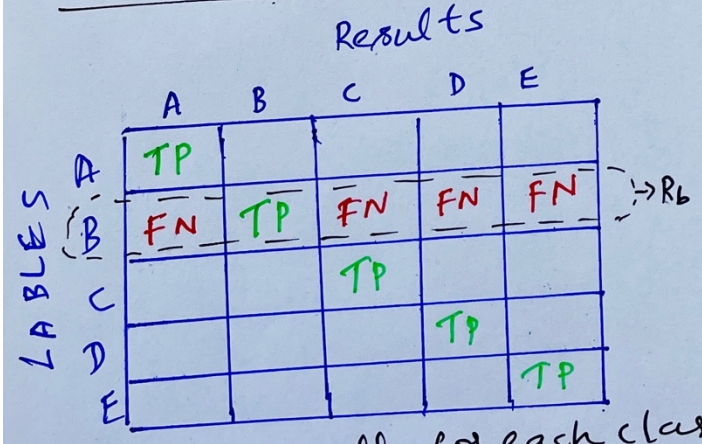
$$F_1 = \frac{2}{P^{-1} + R^{-1}} = \frac{2PR}{P + R}. \quad [1b]$$

6. **Macro-average Metrics:** In multi-class classification tasks like MNIST, macro-average metrics compute the average of individual metrics (precision, recall, F1-score) calculated for each class. This approach treats all classes equally and is useful for assessing overall classifier performance across multiple categories.

Here below is a handwritten example of a 5x5 matrix that shows how to calculate Precession, Recall, and F1 score.



RECALL :-



$$R_b = \frac{TP}{\text{Sum of all FN}}$$

$$R_b = \frac{TP}{TP + \text{Sum of all FN}}$$

Similary Recall for each class R_a, R_b, R_c, R_d, R_e is calculated and its macro average is calculated

$$R_{ma} = \frac{R_a + R_b + R_c + R_d + R_e}{5}$$

F_1 Score :-

The F_1 Score is Calculated for each Class of the Precession and Recall, then its macro average is calculated.

$$F_{1a} = \frac{2 P_a R_a}{(P_a + R_a)} \qquad F_{1b} = \frac{2 P_b R_b}{(P_b + R_b)}$$

$$F_{1c} = \frac{2 P_c R_c}{(P_c + R_c)} \qquad F_{1d} = \frac{2 P_d R_d}{(P_d + R_d)} \qquad F_{1e} = \frac{2 P_e R_e}{(P_e + R_e)}$$

ie. F_1 is calculated seperately for each class then

$$\text{macro average } F_1 = \frac{F_{1a} + F_{1b} + F_{1c} + F_{1d} + F_{1e}}{5}$$

Explaining the Program Functionality (What is going in the code):

As explained theoretically, each step is defined as a function in the program and it is explained below. (Program File: ML_Project_Task1.py or ML_Project_Task1.ipynb)

1. **Importing Necessary Libraries:**

   The required libraries are imported like NumPy for numerical operations, Efficient Array Operations, pandas for data manipulation, and from scikit-learn (sklearn) for machine learning tasks like calculating confusion matrix, Precession ,recall and F1 score.

2. **Loading Training and Testing Data from CSV Files:**

   Train file and Test file are separately loaded from the given CSV file.

   np refers to the NumPy library, which was imported at the beginning of the code. **genfromtxt** is a function in NumPy used to load data from a text file, with flexible handling of missing data and column types.

   The **np.genfromtxt** function reads the data from the CSV files specified by train_file and test_file. It automatically detects the data type of each column and loads the data into a NumPy array. Each row in the CSV file corresponds to a row in the NumPy array, and each value in the CSV file corresponds to an element in the array.
   The delimiter specified by **delimiter=","** is used to separate values in the CSV files and determine where one column ends and the next begins.

3. **Separating Features and Labels for Training and Testing Data:**

   - **training_data[:, 0]**
     training_data is the NumPy array containing the training data loaded from the CSV file. [:, 0] is NumPy slicing notation. It means selecting all rows (:) and only the first column (0).This line extracts the labels (digit labels) from the training data. The labels are typically located in the first column of the dataset.

   - **train_digit_labels**
     This variable stores the extracted labels (digit labels) from the training data. Each element of train_digit_labels corresponds to the label of a corresponding data point In the training dataset. For example, if train_digit_labels[0] is 5, it means the first data point in the training dataset represents the digit 5.

   - **training_data[:, 1:]**
     [:, 1:] is another NumPy slicing notation. It selects all rows (:) and all columns starting from the second column (1:).
     This line extracts the pixel features (attributes representing pixel values) from the training data. The pixel features typically start from the second column onwards in the dataset.

- **train_pixel_features -**
  This variable stores the extracted pixel features from the training data.
  Each row of train_pixel_features represents a data point, and each column represents a pixel feature. For example, train_pixel_features[0] represents the pixel values of the first data point in the training dataset.

4. Initializing and Training Gaussian Naive Bayes Classifier:

gnb = GaussianNB()

- **GaussianNB():**

GaussianNB() is a constructor function that creates an instance of the Gaussian Naive Bayes classifier. It belongs to the sklearn.naive_bayes module in scikit-learn (a popular machine learning library in Python).The GaussianNB class implements the Gaussian Naive Bayes algorithm specifically for data with continuous features. This algorithm assumes that the likelihood of the features is Gaussian (i.e., normally distributed) within each class. The constructor function does not requires any arguments. It initializes the classifier object with default parameters.

- **gnb = GaussianNB():**

This line creates a new instance of the Gaussian Naive Bayes classifier using the GaussianNB() constructor function.
The resulting classifier object is assigned to the variable gnb, making it available for further use in the code.

- **fit():**

.fit() is a method of the classifier object. In scikit-learn, the .fit() method is used to train the model on the given training data.

- **Training Process:**

During the training process, the Gaussian Naive Bayes classifier learns the statistical properties of the training data. Specifically, it estimates the mean and standard deviation of the feature values for each class (digit).This information is used to build a probabilistic model that can predict the class of new, unseen samples based on their feature values.

- **Supervised Learning:**

Naive Bayes classifiers are a kind of supervised mastering algorithm. Supervised learning means that the algorithm learns from labeled training data, where each sample is associated with a target label (class). In this case, the classifier learns to predict the digit label of an image based on its pixel values.

- **Training Outcome:**

After the .fit() method completes, the classifier object (gnb) is trained and ready to make predictions on new, unseen data. The trained model encapsulates the learned patterns and relationships between the input features (pixel values) and the target labels (digit classes).

5. Predicting the Response for Test Dataset:

prediction = gnb.predict(test_pixcel_features)

- **gnb:**

gnb is the variable representing the trained Gaussian Naive Bayes classifier object. This object was initialized and trained earlier using the .fit() method.

- **predict():**

.predict() is a method of the classifier object. In scikit-learn, the .predict() method is used to make predictions on new, unseen data.

- **Prediction Process:**

During the prediction process, the trained classifier (gnb) uses the learned model parameters to predict the class labels (digits) for the test samples. For each sample in test_pixel_features, the classifier computes the posterior probability of each class (digit) based on the observed pixel features. The class with the highest probability is predicted as the label for the corresponding sample.

- **Supervised Learning:**

Similar to the training phase, the prediction phase is also a part of supervised learning. The classifier makes predictions based on the patterns and relationships learned from the labeled training data.

- **Output:**

After the .predict() method completes, the prediction variable contains the predicted digit labels for the test data. Each element in prediction corresponds to the predicted label (class) for the corresponding sample in test_pixel_features.

6. Calculating and Printing Confusion Matrix:

- **confusion_matrix():**

confusion_matrix() is a function provided by scikit-learn (sklearn.metrics) for calculating the confusion matrix.
It takes two arguments: the true labels (test_digit_labels) and the predicted labels (prediction) for the test data. The confusion matrix is a square matrix where each row represents the actual class (label) and each column represents the predicted class (predicted label). The matrix contains counts of how many samples belong to each combination of true and predicted classes.

Each cell in the confusion matrix represents a count of samples. The diagonal cells (top-left to bottom-right) represent correct predictions, while off-diagonal cells represent incorrect predictions. By analyzing the confusion matrix, one can identify which classes are frequently misclassified and gain insights into the model's strengths and weaknesses.

7. Calculating the Accuracy of the Model:

- **accuracy_score():**

accuracy_score() is a function provided by scikit-learn (sklearn.metrics) for calculating the accuracy score. It takes two arguments: the true labels (test_digit_labels) and the predicted labels (prediction) for the test data.

- **Accuracy Calculation:**

The accuracy_score() function compares the true labels (test_digit_labels) with the predicted labels (prediction) and calculates the proportion of correctly classified samples. It counts the number of correctly classified samples and divides it by the total number of samples in the test dataset to compute the accuracy score.

- **Interpretation:**

The accuracy score ranges from 0 to 1, where 1 represents perfect classification (all samples are correctly classified) and 0 represents no correct classifications.
A higher accuracy score indicates better performance of the classification model.

8. Calculating and Printing Precision and Recall for Each Class:

- **Precision:**

Precision is a metric that quantifies the accuracy of positive predictions made by the classifier. It is described because the ratio of proper fantastic predictions to the overall variety of fantastic predictions made. For each class, precision is calculated as the ratio of true positives to the sum of true positives and false positives.
The precision_score() function from scikit-learn is used to calculate precision.
The **average=None** argument indicates that precision should be calculated for each class individually.

- **Recall:**

Recall, also known as sensitivity or true positive rate, measures the ability of the classifier to correctly identify positive instances (true positives) out of all actual positive instances. For each class, recall is calculated as the ratio of true positives to the sum of true positives and false negatives.
The recall_score() function from scikit-learn is used to calculate recall.
Like precision, the **average=None** argument indicates that recall should be calculated for each class individually.

- **Interpretation:**

Precision and recall are complementary metrics. Precision focuses on minimizing false positive predictions, while recall focuses on minimizing false negative predictions.
Higher precision shows fewer fake positives, even as better consider shows fewer fake negatives. Both precision and recall are important for evaluating the performance of a classification model, especially in scenarios where class imbalance exists.

9. Calculating Precision and Recall with Macro Averaging:

- **Macro-Averaged Precision:**

Macro-averaged precision calculates the precision for each class individually and then averages the precision scores across all classes. It treats all classes equally, regardless of their size or imbalance.
The average='macro' argument in the precision_score() function specifies that macro-averaged precision should be calculated.

- **Macro-Averaged Recall:**

Macro-averaged recall calculates the recall for each class individually and then averages the recall scores across all classes. Similar to macro-averaged precision, it treats all classes equally and is insensitive to class imbalance.
The average='macro' argument in the recall_score() function specifies that macro-averaged recall should be calculated.

10. Calculating F1 Score for Each Class and Macro average :

- **F1 Score for Each Class:**

The F1 score is the harmonic mean of precision and recall. It provides a single metric that balances both precision and recall.
For each class in the classification problem, the F1 score is calculated as
2 * (precision * recall) / (precision + recall).
The f1_score() function from scikit-learn is used to calculate F1 scores.
The average=None argument specifies that F1 scores should be calculated for each class individually. The resulting f1_scores array contains the F1 score for each class.

- **Macro-Averaged F1 Score:**

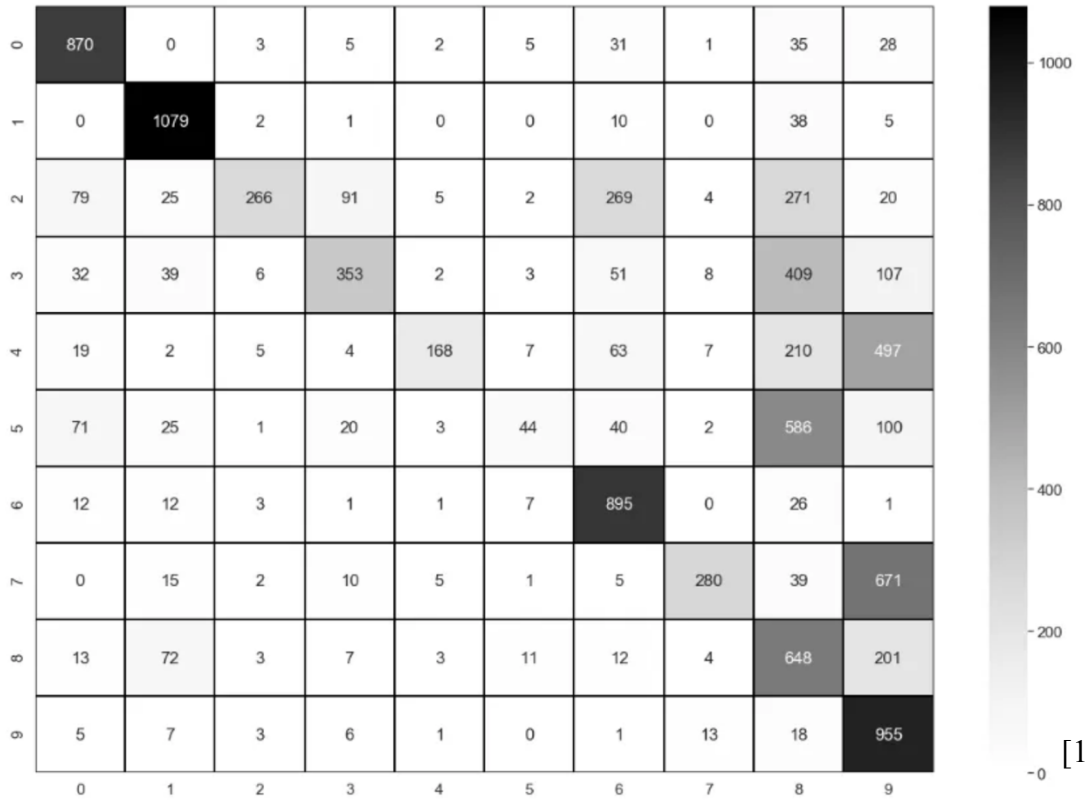
The macro-averaged F1 score calculates the F1 score for each class individually and then averages the F1 scores across all classes. Similar to other macro-averaged metrics, it treats all classes equally and is insensitive to class imbalance.
The average='macro' argument in the f1_score() function specifies that macro-averaged F1 score should be calculated.The resulting f1_macro variable contains the macro-averaged F1 score.

- **Interpretation:**

F1 scores provide a comprehensive evaluation of the classifier's performance by considering both precision and recall. The F1 score for each class and the macro-averaged F1 score offer insights into the model's ability to balance precision and recall across different classes in the classification problem.

Observations

- Naive Bayes Classifier does not a appear to perform well for MNIST data set as it produced an overall accuracy of 56%.
- Looking at confusion matrix figure below, we can observe that (5,8), (5,9), (4,8), (4,9), (7,9) are some of the combinations where the classifier is confused in predicting the right label. [1e]



[1e]

Conclusion

In conclusion, the Gaussian Naive Bayes (GNB) approach demonstrates promising results for handwritten digit recognition tasks using the MNIST dataset. Leveraging the assumption of feature independence given the class label and the Gaussian distribution of pixel intensities, the GNB model efficiently learns to classify digits with minimal computational complexity. Through training on a large collection of labeled handwritten digit images, the GNB model achieves competitive accuracy rates and demonstrates robust performance in identifying digits ranging from 0 to 9. Moreover, the simplicity and interpretability of the GNB model make it suitable for practical deployment in various applications requiring digit recognition, such as optical character recognition (OCR) systems and automated form processing. While the GNB approach may exhibit limitations in capturing complex relationships between pixel features, it serves as a reliable baseline model and complements more sophisticated machine learning techniques. Future enhancements to the GNB model could involve exploring advanced preprocessing techniques, incorporating feature engineering strategies, and investigating ensemble methods to further improve recognition accuracy and robustness. Overall, the GNB approach serves as a valuable asset in the arsenal of algorithms for handwritten digit recognition tasks, offering a balance between simplicity, efficiency, and performance.

Some informative questions addressed in my work: –

1. How do you handle features with a constant value across all images in the dataset?

Handling features with a constant value across all images in the dataset is important for several reasons:

- **Efficiency:** features with constant values do not provide any useful information for the model and can be safely removed to reduce computational overhead during training and inference.

- **Avoidance of overfitting:** including constant features in the model can lead to overfitting, where as the model learns to fit noise in the training data rather than capturing meaningful patterns.

Some approaches to handle features with constant values:

- **Identify Constant Features:** Begin by identifying which features have a constant value across all images in your dataset. This can be done through exploratory data analysis or by writing code to programmatically detect constant features.

- **Remove Constant Features:** If the constant features do not provide any meaningful information for the task at hand, consider removing them from your dataset. Having such features can add noise to the model without contributing to its predictive ability.

- **Assess Impact on Model Performance:** Before removing constant features, it's crucial to evaluate their impact on model performance. You can train and evaluate your model with and without these features to determine whether they significantly affect performance metrics such as accuracy, precision, recall, or F1-score.

2. How do you handle features with a constant value across all images belonging to a given class?

To handle features with constant values across all samples within a particular class in a dataset, several strategies can be employed. One approach is to utilize feature selection techniques like VarianceThreshold from libraries such as scikit-learn, which automatically removes features with low variance, indicating they provide little discriminatory information. Additionally, manual inspection of features can be conducted, particularly suitable for smaller datasets or when domain knowledge is available to assess feature relevance. Feature engineering offers another avenue, where new features capturing more meaningful information, such as edge density or texture features, can be derived from raw pixel values. Moreover, employing feature selection techniques like recursive feature elimination or feature importance ranking can help identify and retain the most informative features while discarding constant ones. Regularization methods, such as L1 or L2 regularization during model training, can also penalize unnecessary or redundant features, promoting sparsity in the feature space and automatically mitigating the

impact of constant features. By leveraging these strategies, the machine learning model can learn from the most relevant and discriminative features, enhancing its ability to generalize and make accurate predictions on unseen data.

3. Do you, and if so, how do you the prevent arithmetic underflow or overflow?

In machine learning models implemented in Python, preventing arithmetic underflow or overflow is crucial for maintaining numerical stability and ensuring accurate predictions. One approach to mitigate these issues is through careful selection of numerical data types and scaling techniques. By utilizing appropriate data types with sufficient precision and dynamic range, such as floating-point representations like float32 or float64 in NumPy, the risk of arithmetic underflow or overflow can be minimized. Additionally, employing scaling methods like feature scaling (e.g., normalization or standardization) can help to rescale input features to a more manageable range, reducing the likelihood of numerical instability during computation. Through these measures, the model can effectively manage numerical operations, enhancing its robustness and reliability.

Furthermore, leveraging specialized libraries and functions designed to handle numerical computations in Python can further aid in preventing arithmetic underflow or overflow. Libraries like NumPy and TensorFlow offer built-in functions and methods optimized for numerical stability, such as np.clip() for value clipping and tf.clip_by_value() in TensorFlow. These functions can help to constrain intermediate values within a safe range, mitigating the risk of numerical instability during computation. By incorporating such techniques and utilizing appropriate libraries, machine learning models in Python can maintain numerical stability, ensuring accurate and reliable predictions across various tasks and datasets.

4. How do you explain the low classification accuracy obtained by applying the Gaussian naive Bayes approach in this particular context?

- **Assumption of Feature Independence:** GNB assumes that features (pixel values) are conditionally independent given the class label. However, in the context of complex datasets like handwritten digit recognition, this assumption may not hold true. Handwritten digits often exhibit intricate patterns and correlations between pixels that violate the independence assumption, leading to suboptimal model performance.

- **Sensitivity to Feature Distribution:** GNB assumes that features follow a Gaussian (normal) distribution. If the distribution of features in the dataset deviates significantly from Gaussian, the model may not accurately capture the underlying data distribution, resulting in decreased classification accuracy.

- **Limited Expressiveness:** GNB is a simple and naive classifier that lacks the capacity to capture complex relationships and interactions between features. In tasks where features exhibit intricate dependencies or nonlinear relationships, GNB may struggle to model the underlying data distribution effectively, leading to lower classification accuracy.

- **Imbalanced Class Distribution:** If the dataset exhibits imbalanced class distribution, where certain classes are significantly underrepresented compared to others, GNB may have

difficulty accurately predicting the minority classes. The model may exhibit bias towards the majority classes, leading to lower accuracy for minority classes.

- **Inadequate Feature Representation:** Handwritten digit recognition tasks may benefit from more sophisticated feature representations beyond raw pixel values. GNB's reliance on pixel-level features alone may limit its ability to capture higher-level patterns and structural characteristics present in handwritten digits, contributing to lower classification accuracy.

- **Model Complexity:** GNB is a simple and computationally efficient model. While its simplicity is advantageous in certain scenarios, it may also limit its ability to capture complex patterns and variations present in the data, particularly in tasks that require a more sophisticated modeling approach.

- **Data Quality and Preprocessing:** The quality of the input data and the effectiveness of preprocessing steps (e.g., normalization, dimensionality reduction) can significantly impact the performance of GNB. Inadequate data preprocessing or noisy input data may introduce errors and inconsistencies that adversely affect classification accuracy.

Output

```
Confusion Matrix:
[[ 870    0    3    5    2    5   31    1   35   28]
 [   0 1079    2    1    0    0   10    0   38    5]
 [  79   25  266   91    5    2  269    4  271   20]
 [  32   39    6  353    2    3   51    8  409  107]
 [  19    2    5    4  168    7   63    7  210  497]
 [  71   25    1   20    3   44   40    2  586  100]
 [  12   12    3    1    1    7  895    0   26    1]
 [   0   15    2   10    5    1    5  280   39  671]
 [  13   72    3    7    3   11   12    4  648  201]
 [   5    7    3    6    1    0    1   13   18  955]]

 THE ACCURACY IS: 0.5558

Precision for each class:
[0.79019074 0.84561129 0.9047619  0.70883534 0.88421053 0.55
 0.64996369 0.87774295 0.28421053 0.36943907]

Precision Macro Average: 0.6864966027174797

Recall for each class:
[0.8877551  0.95066079 0.25775194 0.34950495 0.17107943 0.04932735
 0.934238   0.27237354 0.66529774 0.94648167]

Recall Macro Average: 0.548447051043586

F1 Score for each class:
[0.83613647 0.89506429 0.40120664 0.46816976 0.28668942 0.09053498
 0.76659529 0.41573868 0.39827904 0.53144129]

F1 Score Macro Average: 0.5089855858364396
```

Reference:

1. **Text Reference:**

   a) https://www.javatpoint.com/machine-learning-naive-bayes-classifier
   b) Lecture Notes - http://gnjatovic.info/misc/naive.bayesian.classification.pdf
   c) https://www.analyticsvidhya.com/blog/2023/01/naive-bayes-algorithms-a-complete-guide-for-beginners/
   d) https://github.com/yanshengjia/ml-road/blob/master/resources/Hands%20On%20Machine%20Learning%20with%20Scikit%20Learn%20and%20TensorFlow.pdf Book - Hands on machine learning with Scikit-learn and TensorFlow. (Since the book was referred from a library, I'm sharing the GitHub reference link for the same book in PDF format.)
   e) https://rnagara1.medium.com/mnist-handwritten-image-classification-with-naive-bayes-and-logistic-regression-9d0dd2b6edc0

2. **Source Code Reference :**

a) Code Snippets - http://gnjatovic.info/machinelearning/gaussian.naive.bayes.classification.2.R
b) Lecture Notes