MACHINE LEARNING

Project-1

Program for classification of handwritten digits based on the Gaussian naive Bayes approach.

Name: **NIKHIL CHEZIAN**

Matriculation Number: **11027706**

Professor: **Dr. MILAN GNJATOVIC**

Date: **2nd April 2023**

Introduction

Machine learning is a subfield of artificial intelligence that focuses on creating algorithms that can learn from and make predictions or decisions based on data. It involves the use of statistical and computational techniques to build models that can automatically improve their performance over time by learning from new data. Machine learning has become increasingly popular in recent years due to its ability to address a wide range of problems in various domains, such as image and speech recognition, natural language processing, and predictive analytics. The main goal of machine learning is to enable machines to perform tasks without being explicitly programmed to do so, instead relying on the data to guide their decision-making processes.

One of the popular approaches in machine learning is the Naive Bayes algorithm, which is a probabilistic algorithm based on Bayes' theorem. It assumes that the presence or absence of a particular feature in a class is independent of the presence or absence of other features. Naive Bayes is commonly used for text classification tasks, such as spam filtering, sentiment analysis, and topic classification, because of its simplicity, speed, and effectiveness in handling high-dimensional data. It works by calculating the probabilities of each class given the input features and then predicting the most likely class based on these probabilities. Naive Bayes can be easily trained on a large amount of data and can make accurate predictions even with limited data. Therefore, it is a useful and popular machine learning algorithm in various applications, including natural language processing, recommendation systems, and image recognition.

$$\mathrm{P(y|x\_1, x\_2, \ldots, x\_n) \; = \; P(y) \, * \, P(x\_1|y) \, * \, P(x\_2|y) \, * \ldots * \, P(x\_n|y)}$$

Eq.1: Naive Bayes Theorem

where y is the class label, x_1, x_2, ..., x_n are the features, and P(y|x_1, x_2, ..., x_n) is the probability of y given the values of x_1, x_2, ..., x_n. P(y) is the prior probability of y, and P(x_i|y) is the conditional probability of feature x_i given y.

Implementation

In this project, the Python code demonstrates the implementation of the Gaussian Naive Bayes algorithm for multi-class classification on the MNIST dataset. The MNIST dataset is a well-known and popular dataset used in the field of machine learning for solving the problem of handwritten digit classification. It contains a total of 70,000 images of handwritten digits, with 60,000 images used for training and 10,000 images for testing. The applications of these models are widespread, and they can be utilized in different areas such as character recognition, automated document processing, and computer vision. The accuracy of the models trained on the MNIST dataset demonstrates their potential to make significant contributions to various fields. The code uses the 'pandas' library to load the training and testing data from CSV files and pre-processes the data to separate the features and labels. The GaussianNB() function from the 'sklearn' library is then used to train a Naive Bayes model on the training data. The fitted model is used to predict the target variable on the testing data, and the performance of the model is evaluated using the confusion matrix. The code also calculates and prints the precision, recall, F1-score, and support for each class to evaluate the model's performance on individual classes. This code can be used as a reference for implementing the Gaussian Naive Bayes algorithm on other datasets for multi-class classification tasks.

Algorithm

- Load the training and testing data from the given file paths.
- Split the data into features (X) and labels (y).
- Generate a Gaussian Naive Bayes model.
- Train the model using the training data.
- Use the trained model to predict the labels of the testing data.
- Evaluate the model using the Confusion Matrix and store it in the variable cm.
- Print the Confusion Matrix.
- Evaluate the model using Precision, Recall, and F1-score and store the results in precision, recall, f1.
- Print Precision, Recall, F1-score.
- Evaluate the model using Accuracy and store the result in accuracy.
- Print the Accuracy.

Output

The **confusion matrix** is a table that summarizes the predicted class labels and the actual class labels for a set of examples, allowing one to evaluate how well a classification model is performing. It provides a wealth of information about the performance of the model, including measures such as accuracy, precision, recall, and F1-score, which can be calculated from the values in the confusion matrix. The matrix is organized into a grid with two rows and two columns, where the rows represent the predicted class labels (positive and negative) and the columns represent the actual class labels (positive and negative). The diagonal cells of the matrix correspond to the examples that are correctly classified, while the off-diagonal cells correspond to the examples that are misclassified.

Confusion Matrix		Predicted Class									
		0	1	2	3	4	5	6	7	8	9
Actual Class	0	870	0	3	5	2	5	31	1	35	28
	1	0	1079	2	1	0	0	10	0	38	5
	2	79	25	266	91	5	2	269	4	271	20
	3	32	39	6	353	2	3	51	8	409	107
	4	19	2	5	4	168	7	63	7	210	497
	5	71	25	1	20	3	44	40	2	586	100
	6	12	12	3	1	1	7	895	0	26	1
	7	0	15	2	10	5	1	5	280	39	670
	8	13	72	3	7	3	11	12	4	648	201
	9	5	7	3	6	1	0	1	13	18	955

Fig1.1: The confusion Matrix obtained from the code execution.

Precision is an evaluation metric that measures how well the model predicts the positive instances out of all the instances it labelled as positive. It is a way of determining the accuracy of the model's positive predictions. The precision can be computed using the following equation:

$$Precision = \frac{True\ Positive}{True\ Posistive + False\ Positive}$$

where true positive is the number of positive predictions that were correctly predicted by the model, and false positive is the number of negative instances that were inaccurately predicted as positive. The precision score ranges from 0 to 1, with a score of 1 indicating perfect precision (all the model's positive predictions are correct) and a score of 0 indicating no precision (all the model's positive predictions are incorrect). A high precision score implies that the model is making fewer false positive errors, which is important in many applications such as medical diagnosis and fraud detection.

Precision Matrix									
0.790191	0.845611	0.904762	0.708835	0.884211	0.55	0.649964	0.877743	0.284211	0.369582

Fig1.2: Precision Matrix obtained from the code execution.

Recall is a performance metric used in machine learning to evaluate the model's ability to correctly identify the positive instances out of all the actual positive instances. In other words, recall measures how many of the actual positive instances were correctly identified by the model as positive. Recall can be calculated using the formula:

$$Recall = \frac{True\ Positive}{True\ Posistive + False\ Negative}$$

where true positive is the number of correctly predicted positive instances, and false negative is the number of positive instances that were incorrectly predicted as negative. The recall score also ranges from 0 to 1, where a score of 1 indicates perfect recall (all actual positive instances were correctly identified by the model), while a score of 0 indicates no recall (all actual positive instances were incorrectly identified by the model). A high recall score indicates that the model is making fewer false negative errors.

Precision and recall have different evaluation focuses: precision is for the model's positive predictions, while recall is for the actual positive instances in the dataset. A high precision means fewer false positives, and high recall means fewer false negatives. There is usually a trade-off between precision and recall, so finding a balance between them is necessary for the task.

Recall Matrix									
0.887755	0.950661	0.257752	0.349505	0.171079	0.049327	0.934238	0.272639	0.665298	0.946482

Fig1.3: Recall Matrix obtained from the code execution.

F1 score is a widely used evaluation metric in machine learning for assessing the classification model's performance. It is the harmonic mean of precision and recall, where precision measures true positive predictions out of all positive predictions, and recall measures true positive predictions out of all actual positives. The F1 score ranges from 0 to 1, with 1 representing perfect precision and recall, and 0 indicating poor performance. The F1 score is valuable when both precision and recall are essential, and a trade-off between the two is required.

F1 Score									
0.836136	0.895064	0.401207	0.46817	0.286689	0.090535	0.766595	0.416048	0.398279	0.531589

Fig1.4: F-1 Score obtained from the code execution.

The **accuracy score** is another common metric used in machine learning to evaluate the performance of a classification model. It measures the proportion of correct predictions out of all predictions made by the model. The accuracy score ranges from 0 to 1, where a score of 1 indicates perfect performance, while a score of 0 indicates no correct predictions. The accuracy score is useful when the classes are balanced in the dataset, and when false positives and false negatives have similar consequences. However, in imbalanced datasets, accuracy can be misleading and other metrics like precision, recall, and F1 score may be more appropriate.

Accuracy
0.5558555855585559

Fig1.5: Accuracy Score obtained from the code execution.

Additional Questions

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

It is quite common to encounter features in image datasets that exhibit a constant value across all images. Unfortunately, such features do not contribute to the model's performance and may even cause problems during training, such as numerical errors due to division by zero or singular covariance matrices in certain algorithms.

To tackle this issue, there are several approaches that can be adopted. One simple approach is to remove such features altogether from the dataset. This can be done either manually or by using automated feature selection techniques. For example, to remove the constant features: Suppose we have an image dataset containing 10 features, out of which 2 features have the same constant value across all images. We can remove these 2 features from the dataset using pandas in Python by using the following code:

```
import pandas as pd
# Load the dataset
dataset = pd.read_csv("image_dataset.csv")
# Remove the constant features
dataset = dataset.loc[:, (dataset != dataset.iloc[0]).any()]
```

Another approach is to leverage domain knowledge to extract useful information from such features. This is particularly useful in cases where seemingly constant features have some relevance to the underlying problem.

Regularization techniques such as L1 or L2 regularization can also be used to discourage the model from relying on such features, thereby mitigating the problem. Suppose there is an image classification problem where some features have a constant value across all images. L1 or L2 regularization can be used to reduce the impact of such features on the model's performance. For instance, implementing L1 regularization in a logistic regression model may lead to sparse coefficients, implying that some coefficients are set to zero, effectively ignoring the corresponding features.

In situations where the constant features have values close to zero, they can cause issues for algorithms that require feature scaling. In such cases, feature scaling techniques can be used to rescale the features and avoid these issues.

Finally, feature engineering techniques such as PCA or feature aggregation can be used to extract more meaningful features from such datasets. Suppose an image dataset contains a feature that has the same constant value across all images. Feature engineering techniques such as PCA can be used to extract more meaningful features from the dataset. For instance, performing PCA on the dataset and selecting the principal components that capture the most variance in the data can be done. This can help to uncover underlying patterns and correlations that may not be immediately obvious.

```
from sklearn.decomposition import PCA
# Load the dataset
dataset = pd.read_csv("image_dataset.csv")
# Perform PCA on the dataset
pca = PCA(n_components=5)
pca_features = pca.fit_transform(dataset.values[:, :-1])
```

It is worth noting that the choice of approach will depend on the specific dataset and the problem being solved. Evaluating the performance of each approach is important to ensure that the chosen solution leads to the best possible results.

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

Features with a constant value across all images belonging to a given class are known as constant features. These features do not provide any information to the model and can even have a negative impact on the performance of the model. Therefore, it is recommended to remove these constant features from the dataset. The performance of the model can be enhanced and the dataset's dimensionality can be decreased by removing certain features. Calculating the variance of each feature across all samples in the dataset is one technique to spot and eliminate such consistent characteristics. A feature can be eliminated from the dataset if it has a consistent value across all samples and its variance is zero. Libraries such as scikit-learn in Python provide the ‘Variance-Threshold’ method that can be used to remove features with low variance.

Here are some approaches to handle such features:

- **Remove the constant feature**: A dataset of images of fruits has features such as weight, color, and size. If a feature such as "is_fruit" has the same value for all images belonging to the fruit class, it can be removed from the dataset since it does not provide any useful information to the model. For instance, if all images in the dataset belong to the fruit class, the "is_fruit" feature is a constant feature and can be removed.
- **Combine the constant feature with another relevant feature**: In the fruit dataset example, suppose we have a feature such as "has_stem" that has a constant value across all fruit images. To create a more informative feature, "stem_size," this feature can be combined with the "size" feature.
- **Use feature selection techniques**: For instance, in a dataset of images of animals, features such as height, weight, and color are included. If a feature such as "has_tail" has the same value for all images belonging to the mammal class, a feature selection technique such as variance threshold can be used to remove it from the dataset.

- **Analyze the feature's importance**: In a dataset of images of vehicles, features such as engine power, weight, and the number of wheels are included. If a feature such as "is_vehicle" has a constant value across all vehicle images, it may still be crucial for distinguishing between vehicles and non-vehicles. In this scenario, analyzing the feature's importance across all classes before deciding to remove it is critical.

Overall, it is important to evaluate the performance of each approach and select the most appropriate one for the specific dataset and problem being solved.

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

Arithmetic overflow and underflow are common issues that can occur when performing mathematical operations on numbers that are too large or too small to be represented accurately within the available memory or storage space of the computer.

Arithmetic overflow occurs when the result of a mathematical operation exceeds the maximum representable value for a given data type or memory size. For example, if we try to store the result of an addition operation between two large numbers in an 8-bit integer data type, the result may exceed the maximum value that can be stored in 8 bits, resulting in an overflow error.

On the other hand, arithmetic underflow occurs when the result of a mathematical operation is too small to be accurately represented within the available memory or storage space. For example, if we try to store the result of a division operation that results in a very small number in a floating-point data type with limited precision, the result may be rounded to zero, leading to an underflow error.

To prevent arithmetic underflow or overflow, several techniques can be used:

I] Scaling: Scaling the input data to a smaller range can prevent overflow and underflow by reducing the magnitude of the numbers being calculated.

Example- Suppose there is a dataset that contains some very large and very small values. To prevent overflow and underflow, the data can be scaled to a smaller range. One method for achieving this is by normalizing the data through min-max scaling, which scales the values between 0 and 1. An illustration in Python is given below:

```
from sklearn.preprocessing import MinMaxScaler

# Load the data

X = load_data()

# Scale the data

scaler = MinMaxScaler()

X_scaled = scaler.fit_transform(X)
```

II] Logarithmic transformation: Taking the logarithm of the input data can help to reduce the range of values and prevent underflow and overflow.

Example - If a dataset contains values that span several orders of magnitude, taking the logarithm of the values can reduce the range of values and prevent underflow and overflow.

```
import numpy as np

# Load the data

X = load_data()

# Take the logarithm of the data

X_log = np.log(X)
```

III] Numerical precision: Increasing the numerical precision of the computer system or software being used can also help to prevent underflow and overflow.

Example - in Python, we can use the float128 data type to increase the precision of the calculations. Here's an example of how to do this

```
import numpy as np

# Load the data

X = load_data()

# Perform calculations with increased precision

X = np.array(X, dtype=np.float128)
```

IV] If a dataset contains extreme values that cause underflow or overflow, one can clip or truncate the values to a certain range to prevent the issue.

Example -

```
import numpy as np

# Load the data

X = load_data()

# Clip extreme values

X_clipped = np.clip(X, -1000, 1000)
```

Q.4

The low classification accuracy obtained by applying the Gaussian Naive Bayes approach in this particular context could be due to various reasons. One possible reason could be the assumption of independence among the features in the Naive Bayes algorithm, which might not hold true for the MNIST dataset. Another reason could be the limited expressiveness of the model, which might not be able to capture the complexity of the dataset. Additionally, the MNIST dataset contains images of handwritten digits, which can have significant variations in writing style, shape, and size, making it a challenging task for the algorithm to accurately classify them. Overall, the low accuracy score obtained indicates the need for more advanced and sophisticated models for this task.

To build more advanced and sophisticated models for this task, there are several techniques and approaches that can be used:

I] Convolutional Neural Networks (CNNs): CNNs are a powerful deep learning technique that can extract features from images and learn complex patterns. They have been widely used for image classification tasks, including handwritten digit recognition.

II] Transfer learning: Transfer learning involves using a pre-trained model on a large dataset, such as ImageNet, and fine-tuning it on the target dataset. This approach can save significant time and computational resources.

III] Ensemble methods: Ensemble methods involve combining multiple models to improve the overall performance. For example, one can train multiple CNNs with different architectures and combine their predictions to achieve better accuracy.

IV] Data augmentation: Data augmentation involves generating new training samples by applying random transformations, such as rotation, scaling, and flipping, to the existing data. This approach can increase the size of the training set and improve the generalization performance of the model.

V] Hyperparameter tuning: Hyperparameter tuning involves selecting the optimal hyperparameters, such as the learning rate, batch size, and number of layers, for the model. This process can be done using techniques such as grid search or random search.

By using these techniques, it is possible to build more accurate and sophisticated models for handwritten digit recognition.

References

1. http://gnjatovic.info/machinelearning/

2. https://chat.openai.com/

3. https://www.tutorialspoint.com/python/index.htm

4. Input Data Set- https://archive.ics.uci.edu/ml/datasets/sms+spam+collection

5. https://data-flair.training/blogs/python-deep-learning-project-handwritten-digit-recognition/

6. https://scikit-learn.org/stable/tutorial/index.html

7. https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html#training-a-classifier

8. S. Shalev-Shwartz and S. Ben-David, "Understanding Machine Learning: From Theory to Algorithms," Cambridge University Press, 2014. [Online]. Available: https://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/understanding-machine-learning-theory-algorithms.pdf. [Accessed: April 24, 2023].

¶¶¶ Thank You for your Patience ¶¶¶