Handwritten Digit Classification Using Gaussian Naive Bayes Approach

Anjali Pankan, Matriculation Number: 11016821, Date of Submission: 01/02/2022, Task Number: 1, Instructor: Prof. Dr.-Ing. Milan Gnjatović, E-mail: anjali.pankan@gmail.com

Abstract

In this project, a program for the classification of handwritten digits based on Gaussian naive Bayes approach is explained. In order to implement the program, python programming language is used. For training and testing the classifier the program used MNIST dataset. It implemented the training procedure by calculating mean and standard deviations of all pixels (attributes) of all classes (digits) from the MNIST training dataset. The testing procedure used MNIST test dataset and could correctly predict the digits up to an extent.

Index Terms

Gaussian naive Bayes Classification, MNIST dataset.

I. Introduction

Machine learning is a well known field in scientific world in which computers are enabled to learn from past data. The term machine learning was first put in place by Arthur Samuel in 1959. There are many machine learning algorithms available to generate mathematical models to create predictions based on historical data or information. Machine learning has many applications such as speech recognition, image recognition, email filtering, recommender system etc.. In order to construct mathematical models, the machine learning algorithms uses sample historical data, which is referred to as training data. As more and more information is provided to the model, it produces higher performance [1]. This helps making predictions and decisions without the need of explicit programming. A well known algorithm in machine learning is the Gaussian naive Bayes Classifier, which is a simple algorithm that builds fast machine learning models that can help make quick predictions.

One of the most commonly used machine learning datasets is the MNIST dataset. It is a large dataset consisting of handwritten digits from 0 to 9. The handwritten digit images are stored as grayscale images having dimension 28×28 pixels [2]. The main aim of this project is to classify handwritten digits using Gaussian naive Bayes algorithm by utilizing MNIST dataset.

II. Gaussian naive Bayes Classification

In the field of probability theory, Bayes theorem is found as an important mathematical formula to determine conditional probability. Bayes theorem gives a mathematical explanation for describing the probability of an event based on the previous condition, that may related to the event. It is also named as Bayes rule, Bayes law and recently named as Price theorem. Given two events A and B, Bayes theorem can be mathematically stated as shown in Equation 1 [8].

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{1}$$

Where:

- P(A|B) is referred as the posterior probability.
- P(B|A) is referred as the likelihood.
- P(A) is referred as the prior probability.
- and P(B) $\neq$ 0.

Naive Bayes classifiers are a set of algorithms that supports the Bayes theorem. In other words, its a family of algorithms where all of them share a standard principle. Naive Bayes classifiers are based on two assumptions. The first assumption says, a text document can be represented as a bag-of-words such that it is a set of words that belongs to a vocabulary V [8]. The second assumption is that, the presence or absence of a feature does not influence the presence or absence of any other feature. Which means all the features of a class $c \in C$ are independent to each other. Hence, Naive Bayes classifiers assumes conditional independence and is called naive Bayes assumption.

In probability theory and statistics, the Gaussian Naive Bayes classifier is one of the probabilistic classifiers to deal with data of numerical features. This classifier is an extension of Naive Bayes classifier and deals with the strong assumption of independence of features and Bayes theorem. The main assumption that Gaussian Naive Bayes classification depends upon is the normal distribution of values of features in class $c \in C$.

III. Data pre-processing

In this section, the data pre-processing for the handwritten digit classification program using Gaussian Naive Bayes approach is explained. In order to train and test the Gaussian naive Bayes Classifier program, the MNIST dataset has been used [4]. The acronym MNIST stands for Modified National Institute of Standards and Technology [5]. The MNIST dataset is available as comma-separated values (csv) format. It contains separate dataset for training and testing. The training dataset contains 60,000 samples whereas the test dataset contains 10,000 samples. These samples are images of handwritten digits and, using the image processing technique, these images are converted to pixels of size 28×28. It is to be noted that these images were converted into grayscale before determining the values of the pixels. All pixels contains values between 0 to 255. The 28×28 pixels of an image are arranged into a single row consisting of 784 columns. In the MNIST dataset, the first column represents the digit value and the remaining 784 columns represents the actual image corresponding to the digit value in the first column [6].

For loading MNIST dataset into the program, "Pandas" library is bing used. The Pandas is a python library, which is used for data analysis.The Pandas library is built on top of two core Python libraries: Matplotlib for data visualization and NumPy for mathematical operations [7].

Listing 1: Code for loading MNIST dataset using Pandas library

```
import pandas as pd

# Load both training and test datasets.
data_train = pd.read_csv("./mnist_train.csv", header=None)
data_test = pd.read_csv("./mnist_test.csv", header=None)
```

Listing 1 illustrates importing of Pandas library into the program and loading of MNIST dataset using it's read_csv() function. The read_csv() function takes as an argument the path to the dataset file to be loaded as well as other optional parameters. In this project, the program expects the dataset to be kept in the same directory as that of the program. An additional parameter called "header" is set to $None$ to indicate that there is no header for the dataset. In this way all the rows of the loaded dataset are kept part of actual processing data. It is important to note that the read_csv() function loads data in a columnar fashion. That means, the loaded data can be accessed as column rather than rows. For example, data_train[0] gives the first column of training data which contains all 60,000 digit values whose actual pixel values are stored in the remaining 784 columns.

IV. Training procedure

Since the MNIST dataset contains numerical values and satisfy normal distribution of values of features for each class, it is possible to apply Gaussian naive Bayes Classification for this dataset. For training the system, the complete training dataset is being used. That means, the training data used in the program consists of 60,000 rows consisting of 785 columns including the first column containing digit values. Here, the classes represents the digits from 0 to 9, and hence there will be 10 classes. The pixels represents the attributes and there are 784 attributes. These values are respectively saved into variables *num_classes* and *num_attributes* as shown in Listing 2. The program uses python library called NumPy to manipulate dataset arrays. This library is very useful for processing array type objects in python. The num_classes is obtained by finding length of the array returned by NumPy's built in function unique() on the data_train[0] column. Which returns all the unique elements in the data_train[0] column. As the first column (data_train[0]) contains digit values, the number of attributes can be obtained by subtracting 1 from the total number of columns, obtained by calculating data_train.shape[1].

Listing 2: Code for determining number of classes and attributes

```
import numpy as np

# Determine number of classes and attributes from training dataset.
num_classes = len(np.unique(data_train[0]))
num_attributes = data_train.shape[1] - 1
```

The training procedure carried out in this program finds mean and standard deviation of the values of each attribute for each class. Let Fi be an attribute that takes values f_{i1}, f_{i2}, . . . , f_{in_c} in all instances of class c. Then the mean μ_{ic} of the attribute is the average of the values of the attribute calculated by adding all the data points and dividing it with the number of data points. Mathematically it can be represented using the Equation 2 [8].

$$\mu_{ic} = \frac{1}{n_c} \sum_{j=1}^{n_c} f_{ij} \tag{2}$$

Similarly, the standard deviation σ_{ic}, which is a measure of how distributed the data points are in relation to the mean, can be calculated for the attributes using the Equation 3.

$$\sigma_{ic} = \sqrt{\frac{1}{n_c - 1}\sum_{j=1}^{n_c}(f_{ij} - \mu_i)^2} \tag{3}$$

Using the above formulas the mean and standard deviations of the attributes are calculated for each class, using the function naive_bayes(), as showed in the Listing 3.

Listing 3: Code for creating naive byes model by calculating mean and standard deviation of attributes of all classes

```
# Calculate the mean and standard deviation of all classes for
# all attributes. Store these values in separate matrices with
# same dimentions: num_classes x num_attributes.
def naive_bayes():
  means = np.zeros((num_classes, num_attributes), dtype=np.float64)
  stdevs = np.zeros((num_classes, num_attributes), dtype=np.float64)

  digit_array = np.array(data_train[0])
  # Loop through each class in the training dataset.
  for class_num, class_name in enumerate(np.unique(data_train[0])):
    # For the current class, find the indices in the training dataset.
    indices = np.where(class_name == digit_array)
    class_total_instances = len(indices[0])

    # For the current class, determine mean for each attribute.
    for col in range(num_attributes):
      for row in indices[0]:
        means[class_num][col] = means[class_num][col] + data_train[col+1][row]

      means[class_num][col] = means[class_num][col] / class_total_instances

    # For the current class, determine standard deviation for each attribute.
    for col in range(num_attributes):
      mean_diff_sqr = 0.0
      for row in indices[0]:
        mean_diff_sqr = mean_diff_sqr + ((data_train[col+1][row] - means[class_num][col]) **
            2)

      stdevs[class_num][col] = math.sqrt(mean_diff_sqr / (class_total_instances - 1))

  return means, stdevs
```

In this function, two matrices (means and stdevs) with dimension *num_classes* $\times$ *num_attributes* are created to store means and standard deviations respectively. The function then finds the classes from data_train[0] using NumPy's unique() function. It then uses the enumerate() function to get a list of class and its index from the returned list. After that, the function uses a loop to iterate through (index, class) tuple list. For a given class the function then finds the occurrence of it in the data_train[0] column using NumPy's where() function. The where() function returns the list of indices where that particular class is present. Now, since the list of indices are available, it is straight forward to calculate the mean and standard deviation. In order to calculate the mean of an attribute, the values are added first and finally divided with the total indices of the class. This process is repeated for each attribute of the class. The intermediate and final values generated during the calculation are stored in the corresponding position on the mean matrix. In a similar fashion, the standard deviation of all attributes of the class are calculated and stored at the corresponding position of the stdevs matrix. This process is repeated for each class and the mean and standard deviation of all attributes of all classes are determined. At the end, the function returns the means and stdevs matrices. Tables I, II and III shows the mean and standard deviations calculated by the naive_bayes() function for attributes 778, 779 and 780 respectively for each class.

Finally, the prior probabilities of the classes are calculated from the training dataset. The prior probability for an uncertain quantity is the probability distribution that would express one's beliefs about this quantity before some evidence is taken into account [9]. Here, it is the fraction of the images compared to all images that belongs to a particular class from the training set. Listing 4 shows the code that calculates prior probability.

Listing 4: Code for determining prior probabilities of classes

```
# Calculate prior probabilities from the training dataset.
_, fractions = np.unique(data_train[0], return_counts=True)
pri_probs = fractions / len(data_train[0])
```

TABLE I: Means and Standard Deviations of attribute 778

778		
	mean	standard deviation
0	0.0	0.0
1	0.0	0.0
2	0.0	0.0
3	0.0	0.0
4	0.0	0.0
5	0.0	0.0
6	0.0	0.0
7	0.10518754988	3.65287552288
8	0.0	0.0
9	0.0837115481594	3.82529347233

TABLE II: Means and Standard Deviations of attribute 779

779		
	mean	standard deviation
0	0.0	0.0
1	0.0	0.0
2	0.0	0.0
3	0.0	0.0
4	0.0	0.0
5	0.0	0.0
6	0.0	0.0
7	0.112210694334	4.77317757045
8	0.0	0.0
9	0.0342914775593	2.09913216363

V. Testing procedure

Testing the model created during the training procedure is carried out using the MNIST test dataset. For testing the model, the complete test dataset consisting of 10,000 images are used. Like the training dataset the test dataset also contains 785 columns in which, the first column contains the digit values corresponding to the image and the remaining 784 columns contains the pixels of the image. For testing the model, each test image data excluding the first column value is read row by row into an array and passed to the predict function.

The Multinomial naive Bayes classifier determines the maximum posterior probability using the Formula 4 [8]. Here P(c) is the prior probability of a class c and P(f_i|c) is the maximum likelihood estimate of a given value of an attributes f_i against class c. A given test data is assigned to one of the classes (digits) if the calculated posterior probability is highest compared to other classes. As already explained, the prior probability is calculated during the training procedure.

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} P(c) \prod_{i=1}^{n} P(f_i|c) \tag{4}$$

A problem with multiplication of a sequence of probabilities is that it can lead to arithmetic underflow [8]. That means, when the likelihood product is calculated against all attribute values, the final value obtained can be very small that a computer cannot represent it with the given number of bits. In order to eliminate this issue, the log probability can be used. Equation 5 shows the log probability representation of the above equation.

TABLE III: Means and Standard Deviations of attribute 780

780		
	mean	standard deviation
0	0.0	0.0
1	0.0	0.0
2	0.0	0.0
3	0.0	0.0
4	0.0	0.0
5	0.0	0.0
6	0.0	0.0
7	0.0191540303272	1.07253694161
8	0.0	0.0
9	0.0	0.0

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \left(\log P(c) + \sum_{i=1}^{n} \log P(f_i|c) \right) \tag{5}$$

The likelihood estimation P($F_i = f_{ij}$|c) for Gaussian naive Bayes Classification can be determined using Formula 6. Here, f_{ij} is the value of an attribute whose likelihood is to be calculated against a class and, F_i is the attribute whose mean and standard deviation are respectively μ_{ic} and σ_{ic} for a class c.

$$P(F_i = f_{ij}|c) = \frac{1}{\sqrt{2\pi\sigma_{ic}^2}} e^{-\frac{1}{2}\left(\frac{f_{ij} - \mu_{ic}}{\sigma_{ic}}\right)^2} \tag{6}$$

Using the log probability, the above equation can be rewritten as shown in Equation 7 [3].

$$P(F_i = f_{ij}|c) = -(\log \sqrt{2\pi}\sigma_{ic} + \frac{1}{2}\left(\frac{f_{ij} - \mu_{ic}}{\sigma_{ic}}\right)^2) \tag{7}$$

Applying the Equation 7 to Equation 5, it can be rewritten as in Equation 8.

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \left(\log P(c) - \left(\sum_{i=1}^{n} \log \sqrt{2\pi}\sigma_{ic} + \frac{1}{2} \sum_{i=1}^{n} \left(\frac{f_i - \mu_{ic}}{\sigma_{ic}}\right)^2 \right) \right) \tag{8}$$

One observation of the standard deviation matrix that is created during training procedure is that it contains zeros at many attribute places. As a result, such standard deviation values cannot be applied in the Equation 7 as it create unexpected results. Also, since the likelihood estimation does not yield any fruitful information in cases of standard deviation is zero, these values are skipped from calculating likelihood. An important thing to be noted here is that the values at the same attribute positions needs to be also skipped from mean and test data array. Listing 5 shows the calculation of Gaussian likelihood in function log_gaussian_likelihood() where x represents the test data array and mean and stdev are the mean and standard deviation arrays for a given class. Here again the NumPy python library is used for performing array calculations to fasten the process.

Listing 5: Code for calculating log gaussian likelihood

```
# Determine the likelihood estimation of a given test data.
# Used log to eliminate arithmatic undeflow.
def log_gaussian_likelihood(x, mean, stdev):
  # Removing those attributes where the standard deviation is zero.
  remove_indices = np.where(stdev == 0)[0]

  stdev_cleaned = np.delete(stdev, remove_indices)
  mean_cleaned = np.delete(mean, remove_indices)
  x_cleaned = np.delete(x, remove_indices)
  sqrt_2_pi = math.sqrt(2 * math.pi)

  lgl = -(np.sum(np.log(sqrt_2_pi * stdev_cleaned)) + 0.5 * np.sum(((x_cleaned - mean_cleaned)
      / stdev_cleaned) ** 2)).reshape(-1, 1)

  return lgl
```

Finally, the posterior probability is calculated in the predict() function for all classes as shown in Listing 6. This function returns the class that produces the maximum posterior probability using NumPy's built in function argmax().

Listing 6: Code for predicting digits based on maximum posterior probability

```
# Predict the digit using Gaussian naive Bayes classification.
def predict(x):
  c_predict = np.argmax([np.log(pri_probs[c]) + log_gaussian_likelihood(x, means[c],
      stdevs[c]) for c in range(num_classes)])

  return c_predict
```

The predicted values are stored in an array called "predicted_digits". This array has a size equal to the number of rows in the test dataset and the predicted digit values are stored at the same index location as that of the test data. The code for creating the "predicted_digits" array is shown in Listing 7.

VI. Evaluation

Once the ”predicted_digits” array is available it should be possible to evaluate the prediction compared to the actual digit values. From the testing dataset, the array containing actual digit values can be obtained by taking column data_test[0]. The accuracy of prediction is calculated by comparing the two arrays and counting the places where the two arrays are equal, after that dividing it with the total test instance count, as shown in Listing 7. The accuracy obtained using the program is 0.6483 (64.83%).

Listing 7: Code for determining predicted digits and accuracy

```
# Get a row from the test dataset for a given index.
def get_test_data_row(index):
  test_row = np.zeros((num_attributes), dtype=np.float64)

  for i in range(num_attributes):
  test_row[i] = data_test[i+1][index]

  return np.array(test_row)

# Predict digit correspond to each row in the test dataset.
predicted_digits = np.zeros((test_data_count), dtype=np.int32)
for n in range(test_data_count):
  x = get_test_data_row(n)
  predicted_digits[n] = predict(x)

actual_digits = np.array(data_test[0], dtype=np.int32)

# Determine accuracy of prediction compared to the actual values.
accuracy = (actual_digits == predicted_digits).sum()/float(actual_digits.size)
```

From the predicted and actual digit arrays it is also possible to evaluate Precision and Recall values. The Precision represents the fraction of instances that systems assigned to class $c \in C$ that actually belongs to class c, as shown in Equation 9 [8].

$$P = \frac{truepositive}{truepositive + falsepositive} \tag{9}$$

Similarly, the Recall represents the fraction of instances that belong to class $c \in C$ that were actually classified as c, as shown in Equation 10 [8].

$$P = \frac{truepositive}{truepositive + falsenegative} \tag{10}$$

Using this information, the Precision and Recall values of each digit class is calculated and formed a Confusion Matrix using the function confusion_matrix(). Listing 8 shows the confusion_matrix() function.

Listing 8: Code for calculating Confusion Matrix

```
# Determine the confusion matrix.
def confusion_matrix(predicted_digits, actual_digits):
  cm = np.zeros((num_classes, num_classes), dtype=np.int32)

  for i in range(len(predicted_digits)):
  cm[actual_digits[i]][predicted_digits[i]] = cm[actual_digits[i]][predicted_digits[i]] + 1

  return cm
```

For the test dataset, the program produced a confusion matrix which has values as shown in Table IV. From the table it is clear that (5,8), (4,9) and (7,9) are some of the combinations where the classifier failed to predict correct values.

From the Confusion Matrix it is possible to calculate the Macroaverage Precision and Recall using the Equations 11 and 12. Listing 9 shows the code for finding Macroaverage Precision and Recall with functions p_macroaverage() and r_macroaverage() respectively.

$$P_{macroaverage} = \frac{1}{|C|} \sum_{c \in C} P_c \tag{11}$$

TABLE IV: Confusion Matrix

		Predicted class									
		0	1	2	3	4	5	6	7	8	9
Actual class	0	886	1	3	4	3	8	29	2	30	14
	1	0	1092	2	3	0	1	7	0	28	2
	2	43	33	456	82	7	3	185	7	192	15
	3	19	45	12	643	3	6	31	11	162	78
	4	12	8	8	2	349	10	48	9	101	435
	5	69	35	5	47	12	133	33	5	474	79
	6	13	17	7	0	1	9	885	0	25	1
	7	0	18	4	12	9	3	4	435	31	512
	8	14	93	4	15	7	17	11	7	666	140
	9	6	13	3	8	9	1	1	18	21	929

$$R_{macroaverage} = \frac{1}{|C|} \sum_{c \in C} R_c \tag{12}$$

Listing 9: Code for calculating Macroaverage Precision and Recall

```
# Determine precision macroaverage.
def p_macroaverage():
  p_macro = np.zeros((num_classes), dtype=np.float64)

  for i in range(num_classes):
    sum_t = 0
    frac = confusion_matrix[i][i]
    for j in range(num_classes):
      sum_t = sum_t + confusion_matrix[j][i]

    p_macro[i] = frac / sum_t

  return np.sum(p_macro) / num_classes

# Determine recall macroaverage.
def r_macroaverage():
  r_macro = np.zeros((num_classes), dtype=np.float64)

  for i in range(num_classes):
    sum_t = 0
    frac = confusion_matrix[i][i]
    for j in range(num_classes):
      sum_t = sum_t + confusion_matrix[i][j]

    r_macro[i] = frac / sum_t

  return np.sum(r_macro) / num_classes
```

For the MNIST test dataset the program calculated Macroaverage Precision as 0.730747003049 and Macroaverage Recall as 0.640935466183. Finally, the F1-score, which is the harmonic mean of the above two values are calculated using the Equation 13. The F1-score calculated by the program for MNIST test dataset is 0.682901008895. The evaluation result of the program execution is shown in Figure 1.

$$F_{1,macroaverage} = \frac{2\, P_{macroaverage} R_{macroaverage}}{P_{macroaverage} + R_{macroaverage}} \tag{13}$$

VII. Conclusion

In this project, a program for classification of handwritten digits based on the Gaussian naive Bayes approach is successfully realized. The program is written using python programming language. Even though the classifier could classify handwritten digits, the accuracy seems low (64.83%). Also, the classifier confused predicting the right digit values in some places such as (5,8), (4,9) and (7,9).

```
accuracy:  0.6483
[[ 886    1    3    4    3    8   29    2   30   14]
 [   0 1092    2    3    0    1    7    0   28    2]
 [  43   33  465   82    7    3  185    7  192   15]
 [  19   45   12  643    3    6   31   11  162   78]
 [  12    8    8    2  349   10   48    9  101  435]
 [  69   35    5   47   12  133   33    5  474   79]
 [  13   17    7    0    1    9  885    0   25    1]
 [   0   18    4   12    9    3    4  435   31  512]
 [  14   93    4   15    7   17   11    7  666  140]
 [   6   13    3    8    9    1    1   18   21  929]]
p_macroaverage:  0.730747003049
r_macroaverage:  0.640935466183
F1-score:  0.682901008895
```

Fig. 1: Evaluation result part of program output

REFERENCES

[1] *Machine Learning Tutorial*. javatpoint.com. [Online]. Available: https://www.javatpoint.com/machine-learning

[2] *MNIST handwritten image classification with Naive Bayes and Logistic regression*. Rajath Nagaraj. [Online]. Available: https://rnagara1.medium.com/mnist-handwritten-image-classification-with-naive-bayes-and-logistic-regression-9d0dd2b6edc0

[3] *Nave Bayes Implementation*. github.com. [Online]. Available: https://github.com/ocontreras309/ML_Notebooks/blob/master/Naive_Bayes_Implementation.ipynb

[4] *MNIST in CSV*. [Online]. Available: https://pjreddie.com/projects/mnist-in-csv/

[5] *How to Develop a CNN for MNIST Handwritten Digit Classification*. Machine Learning Mastery. [Online]. Available: https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/

[6] *MNIST Handwritten Digit Recognition in PyTorch*. Gregor Koehler. [Online]. Available: https://nextjournal.com/gkoehler/pytorch-mnist/#setting-up-the-environment

[7] *Pandas*. SQL Tutorial. [Online]. Available: https://mode.com/python-tutorial/libraries/pandas/

[8] *Naive Bayesian Classification*. Milan Gnjatović. [Online]. Available: http://gnjatovic.info/misc/naive.bayesian.classification.pdf

[9] *Prior probability*. wikipedia.org. [Online]. Available: https://en.wikipedia.org/wiki/Prior_probability