Introduction to Exploratory Machine [GKS4009-01]

Midterm Project

GKS20221115 Hyojeong Jeong

I. Problem Description

The problem is to classify mnist dataset. Mnist dataset contains 70,000 data that have 28*28 pixels image. Each pixels have values in range 0 ~ 255, which is used to express the gray-scale image of a handwritten number. The numbers are integer in range 0 ~ 9. Each number has a lot of 28*28 pixels images and they have slightly different shapes even though they all express the same number.

II. Theory

Classifying model can use 60,000 train data and 10,000 test data. Also, to classify handwritten numbers, which pixels are used to express their shapes should be considered. In order to express a number on a 28*28 pixels image, each pixel on an image has different probability to be used because handwritten numbers have a lot of different shapes. Also, the pixel probabilities do not affect each other and pixels on a 28*28 pixels image have only two values, True (1) or False (0). To sum up, pixels on each 28*28 pixels image have only two values, and if there is a 28*28 pixels image to express a number, each pixel on it has different probability to be used.

Bernoulli Naïve Bayes is used to classify mnist dataset normally. Bernoulli Naïve Bayes is recommended because the pixel probabilities do not affect each other and they have only two values. Bernoulli Naïve Bayes uses the PMF (Probability Mass Function): f(x; p) = p^x * (1-p)^(1-x). Also, in Bernoulli Naïve Bayes, log, the mathematic concept, is used because it can change calculation with large number to be easier.

III. Method

To make a model similar to Bernoulli Naïve Bayes for classifying mnist dataset, train_test_split model, accuracy_score metrics from scikit-learn are used. Mnist dataset is divided to train and test dataset in order to train model with train dataset and test the accuracy of model with test dataset.

In the process of training model, train dataset is used to get the probability that how many pixels images each number has. Also, train dataset is used to get the probability of each pixel in 28*28 pixels image.

Next, in the process of testing model, model predicts the numbers that test dataset might imply. The model's accuracy is evaluated by comparing predicted numbers and original numbers that test dataset implies.

IV. Implementation

```
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
```

```
from sklearn.metrics import accuracy_score

mnist = fetch_openml("mnist_784", parser = 'auto')

X = mnist.data.values
X = (X > 128).astype(int)
y = mnist.target.values.astype(int)

X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.2)

class BN:
    def __init__(self):
        self.prob_c = []
        self.prob_p = []

    def fit(self, X, y):
        for c in np.unique(y):
            num_c = np.sum(y == c)
            prob_c = (num_c + 1) / len(X)
            self.prob_c.append(np.log(prob_c))

            num1 = X[y == c].sum(axis=0) + 1
            prob_p = num1 / num_c
            self.prob_p.append(np.log(prob_p))

    def predict(self, X):
        y_pred = []

        for x in X:
            class_scores = []

            for c, c_prob in zip(self.prob_c, self.prob_p):
                p_prob = 0
                for i, pixel_value in enumerate(x):
                    if pixel_value == 1:
                        p_prob += c_prob[i]
                    else:
                        p_prob += np.log(1 - np.exp(c_prob[i]))
                class_scores.append(c + p_prob)

            predicted_class = np.argmax(class_scores)
            y_pred.append(predicted_class)

        return y_pred
```

```
model = BN()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: {:.2f}%".format(accuracy * 100))
```

code explanation:

```
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

Import `numpy`, dataset, scikit-learn `train_test_split` model and `accuracy_score` metrics.

```
        mnist = fetch_openml("mnist_784", parser = 'auto')
```

`mnist` gets mnist dataset and `parser = 'auto'` makes importing process faster eliminating useless process.

```
X = mnist.data.values
X = (X > 128).astype(int)
y = mnist.target.values.astype(int)

X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.2)
```

`X` gets data values of `mnist` and `y` gets target values of `mnist`. Then, if each `X` is bigger than 128, it becomes 1 and else, it becomes 0. 128 is the median of range 0 ~ 255. Also, both `X` and `y` gets data as `int` type. Both `X` and `y` are divided to train and test data with the proportion (80:20) and in order to repeat experiment with randomly divided data, the code doesn't fix `random_state`.

```
class BN:
    def __init__(self):
        self.prob_c = []
        self.prob_p = []
```

`class BN` has `__init__(self)`, `fit(self, X, y)`, `predict(self, X)`. First, `__init__(self)` is needed to initialize probability arrays every experiment. In this code, each pixels image data are categorized into each class by the number they imply. `self.prob_c` array has probability of each class and `self.prob_p` array has probability of each pixel for each class.

```
    def fit(self, X, y):
        for c in np.unique(y):
            num_c = np.sum(y == c)
            prob_c = (num_c + 1) / len(X)
            self.prob_c.append(np.log(prob_c))

            num1 = X[y == c].sum(axis=0) + 1
            prob_p = num1 / num_c
```

```
            self.prob_p.append(np.log(prob_p))
```

`for c in np.unique(y)` means that `c` gets category number of data from `y one by one`. `np.unique(y)` is an array of category numbers from 0 to 9. It looks like `[0, 1, 2, … , 9]`. Therefore, `c` becomes 0 at first, and becomes 9 finally.

`num_c` is the number of pixels image data that implies `c` (`c` is the category number). Actually, class is a similar concept to category in this model. `prob_c` gets the proportion that the number of data in each class to the number of whole data. In order to calculate probability in log, probability should not be 0. Therefore, 1 is added to `num_c` and as a result of it, 1 divided by the number of whole data is added to `prob_c`. Each `prob_c` is saved in a `self.prob_c` array.

num1 is the number of 1 in each pixel of whole data in the class. For example, if there is 4 data with 28 * 28 pixels in the class, there is an array with 784 spaces and each space has the number in range 0 to 4. Actually, the number of 1 in each pixel is equal to the sum on value that each pixel with same index has. This is the why the expression of it is `X[y == c].sum(axis = 0) + 1`. `axis = 0` means the vertical. The picture below is attached to help understanding the explanation.

	pixel1	pixel2	pixel3		pixel784
data1	0	0	1		0
data2	0	0	1		1
data3	0	0	1	…	1
data4	0	1	1		0
num1	0	1	4		2

`prob_p` is the proportion, `num1` to `num_c. 1` is added to `num1` because `prob_p` will be used for calculation in log. To sum up, `prob_p` implies the probability of each pixel's having True value. Then, each `prob_p` array of class is saved in `self.prob_p` array.

```
def predict(self, X):
        y_pred = []

        for x in X:
            class_scores = []

            for c, c_prob in zip(self.prob_c, self.prob_p):
                p_prob = 0
                for i, pixel_value in enumerate(x):
                    if pixel_value == 1:
                        p_prob += c_prob[i]
                    else:
                        p_prob += np.log(1 - np.exp(c_prob[i]))
                class_scores.append(c + p_prob)

            predicted_class = np.argmax(class_scores)
            y_pred.append(predicted_class)

        return y_pred
```

`x` gets data from `X_test` one by one. `c` gets data from `self.prob_c` and `c_prob` gets data from `self.prob_p` one by one. `c` is probability of each class and `c_prob` is the array of pixels probability. `i` is index of `x` and `pixel_value` is the value that each pixel has. If `pixel_value` is 1, c_prob[i] is added and if `pixel_value` is 0, (1 - c_prob[i]) is added to p_prob. This code is expression of Bernoulli Naïve Bayes. log(p^x * (1-p)^(1-x)) = x*log(p) + (1-x)*log(1-p). np.exp() is used for numerical stability.

Then, `c` is added to `p_prob` in order to consider class probability and the result is saved to `class_scores` array. Finally, the largest value in `class_scores` array is chosen as predicted class. Each predicted class is saved to `y_pred` array. As a result, y_pred array has numbers that model predicts as numbers that pixels image imply.

```
model = BN()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: {:.2f}%".format(accuracy * 100))
```

Through this procedure, model is trained with train dataset and is tested with test dataset.

V. Experimental result

Experiment was repeated 12 times. The table below show each result with spent times.

N	accuracy	times
1	83.21%	5m 28.3s
2	83.67%	5m 54.2s
3	83.09%	5m 34.3s
4	83.64%	5m 41.7s
5	83.91%	5m 18.3s
6	83.50%	5m 41.2s
7	83.86%	5m 27.5s
8	83.60%	5m 33.6s
9	83.41%	5m 30.8s
10	83.89%	5m 45.1s
11	83.45%	5m 46.1s
12	84.34%	5m 16.1s
13	83.55%	5m 19.4s
14	83.65%	5m 55.8s

Accuracy is about 83% and it takes about 5 minutes. Accuracy is not bad considering the model made before had about 70% accuracy. However, it takes longer time that the previous model. One possible reason is that there are quite many for loop in this code.

Also, if np.exp() is used whenever pixel_value is 1 or 0, accuracy is about 51%. This report failed to discover why np.exp() is used only when pixel_value is 0. More studying in order to make developed model with higher accuracy and shorter time is needed.