Introduction to Exploratory Machine [GKS4009-10]

Midterm Exam Project Report GKS20221115 Hyojeong Jeong

Problem and Theory

The problem is to classify mnist data. Mnist dataset contains 70,000 images of handwritten numbers with 28X28 pixels. To classify what number the specific image implies, the probability of pixels if it has 1 or 0 is important.

We choose Bernoulli Naïve Bayes Classifier because it is useful for classifying data that has two values. With the Bernoulli theory, we can get PMF (Probability Mass Function), (p^x)*((1-p)^(n-x)).

Codes:

```
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")

X, y = mnist.data.values, mnist.target.values

X_binary = (X > 128).astype(int)

X_train, X_test, y_train, y_test = train_test_split(X_binary, y,
test_size=0.2, random_state=42)

class BernoulliNBWithLog:
    def fit(self, X, y):
        self.classes = np.unique(y)
        self.class_priors = np.array([np.log(np.mean(y == c)) for c in
self.classes])

        self.feature_probs = []
        for c in self.classes:
            feature_prob = (X[y == c].sum(axis=0)) / (np.sum(y == c))
            self.feature_probs.append(np.log(np.clip(feature_prob, 1e-10, 1.0
- 1e-10)))
        self.feature_probs = np.array(self.feature_probs)

    def predict(self, X):
        log_likelihoods = np.dot(X, self.feature_probs.T)
        log_posteriors = log_likelihoods + self.class_priors

        predicted_class = self.classes[np.argmax(log_posteriors, axis=1)]
        return predicted_class

model = BernoulliNBWithLog()

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))
```

Explanation of the codes:

```
import numpy as np
```

#import numpy and named it as np

```
from sklearn.datasets import fetch_openml
```

#from scikit learn import fetch_openml to get mnist data

```
from sklearn.model_selection import train_test_split
```

#from scikit learn import model used to split train and test data

```
from sklearn.metrics import accuracy_score
```

#from scikit learn import accuracy_score matrix to get the accuracy of this implement

```
mnist = fetch_openml("mnist_784")
```

#give mnist_784 dataset to mnist variable

```
X, y = mnist.data.values, mnist.target.values
```

#give data of mnist and target of mnist to X and y as int type

```
X_binary = (X > 128).astype(int)
```

#make X have only 1 or 0 and give it to X_binary variable as int type. We can use Bernoulli naïve bayes now. We use 128 because it is the middle value of 0~255. 0~255 is the range of value that pixels originally had.

```
X_train, X_test, y_train, y_test = train_test_split(X_binary, y, test_size=0.2,
random_state=42)
```

#split data to train and test data with test size = 0.2, random state = 42. Datas will be devided to

80 train datas:20 test datas.

```
class BernoulliNBWithLog:
    def fit(self, X, y):
        self.classes = np.unique(y)
```

#np.unique(y) means category(=class) value. If a data image implies 9, its category is 9. There is 0~9 category(=class) values.

```
        self.class_priors = np.array([np.log(np.mean(y == c)) for c in
    self.classes])
```

#through this line, we can get the probability of each class. We get 1 if the specific data's class is c, else we get 0. We can get the probability of class if we get the mean of values we got (1 or 0). It is same with (the number of datas in each class / the number of whole datas).

```
        self.feature_probs = []
        for c in self.classes:
            feature_prob = (X[y == c].sum(axis=0)) / (np.sum(y == c))
```

#similar to the probability of each class, we can get the probability of pixels in each class through this line. Features implies pixels. X[y==c].sum(axis=0) means that it will sum all values in row. np.sum(y == c) means if there are N datas in the class, it will get N * the number of pixels in each data (it is perfectly same). Therefore, it means the probabiblity of pixels in each class and same with (the number of pixels that have True / the number of whole pixels in that class).

```
            self.feature_probs.append(np.log(np.clip(feature_prob,
        1e-10, 1.0 - 1e-10)))
```

#We didn't learn about clip, but we use it to restrict the probability not to close to 0 or 1 too much. If the probability is too close to 0, it might be considered as 0 by the computer, and if the probability is too close to 1, it might be considered as 1 by computer. Both situations can cause the problem during calculation.

Also, we use log to make calculations simpler and more precise.

```
            self.feature_probs = np.array(self.feature_probs)
```

```python
def predict(self, X):
        log_likelihoods = np.dot(X, self.feature_probs.T)
```

#np.dot means dot product of X and the probability of pixels in each class.

```python
        log_posteriors = log_likelihoods + self.class_priors
```

#log_posteriors mean the scores of each class. We can predict which class the data might imply. To sum up, log_likelihoods mean how similar the probability of pixels to the X data, and log_posteriors mean the probability of pixels * the probability of classes (in log, the probability of pixels + the probability of classes).

```python
        predicted_class = self.classes[np.argmax(log_posteriors, axis=1)]
```

#predicted_class the max value of log_posteriors, score of each class. The class with the highest score means it is the most probable class that the test data might implies.

```python
        return predicted_class
```

```python
model = BernoulliNBWithLog()
model.fit(X_train, y_train)
```

#train with train data sets

```python
y_pred = model.predict(X_test)
```

#predict and test with test data sets

```python
accuracy = accuracy_score(y_test, y_pred)
```

#compare the real number that the data implies and the predicted number and get accuracy.

```python
print("Accuracy: {:.2f}%".format(accuracy * 100))
```

#print it with the specific format.

Experimental Result

: We can get 70.36% accuracy. If we change the random state, the results are different.

Also, I have some idea about this implementation. I think the indexes or the positions of pixels that have True value are important in order to predict precisely, but this implementation doesn't consider it. To get higher accuracy, developed implementation is needed.