First, our team's plan was to binarize the training dataset among the MNIST datasets and train it to the model so that when the model is provided with test data, we can figure out what number the image is, i.e. which class it belongs to. To do this, we have written the following code.

To begin with, I called the necessary library to use in the following code. The MNIST dataset was loaded using the function 'fetch_openml'.

'mnist.data.values' contains the pixel values of the image, and 'mnist.target.values' contains the labels of the images.

Next, through unique(), non-overlapping values, are obtained and the minimum and maximum values are checked in this process.

And next code converts pixel values into binary data, which was set to give the number 1 if the pixel value was greater than 0, or 0 to the others. By doing this, we make the data suitable for Bernoulli naive bays because naïve bays use binary data. However, in the process, the accuracy was higher when the threshold was set to 128, which is the median value of the range of the pixel values, than 0 so we change this to 128.

This time, the given dataset is divided into a training dataset and a test dataset, and 80% of the data is used for training and 20% for testing.

Next code is little bit complex, so I'll explain it one by one.

- The class 'BernoulliNBWithLog' is defined for the Bernoulli Naive Beige classifier, which uses log calculations. The reason for converting values to logarithms is that log calculations are useful for converting multiplication operations into addition operations.
- The np.mean(y == c) code used to obtain prior probabilities computes the proportion of elements with values c in a given array y. To specify, y == c produces a Boolean array consisting of True or False depending on whether each element is equal to c. And the np.mean() function computes the percentage of True values in this Boolean array to calculate the percentage that class c occupies in the total data. This is the prior probability of class c.
- And we calculate the probability of each characteristic for each class. We sum the number of pixels binarized by '1' in the pixels belonging to each class. The np.clip used here (feature_prob, 1e-10, 1.0-1e-10) limits the elements in the feature_prob array to between 1e-10 (very small values) and 1.0-1e-10. This is generally used to prevent values close to 0 or 1 before performing log operations. This is called smoothing, which is one of the ways to stabilize the model. We did not learn this in class, so we removed it to make the code

work without using it, but we had no choice but to include it because the probability was too close to 0 or 1 and the code did not work because the code did not work.

- Then, based on the log dictionary probability and the characteristic probability for each class, the prediction is performed by defining the 'predict' method. 'self.T' after 'feature_probs' is an operation that replaces rows and columns of a NumPy array. Through this, information sorted by characteristic (column) is converted into information sorted by class (row).
- Finally, we make the codes that can find the class with the highest log posterior probability, that is, the class to the given image belongs to with the highest probability.

After that, the model is trained using training data by generating 'model' and calling 'model.fit(X_train, y_train)'. Finally, we perform predictions on test data using 'model. predict(X_test)', which causes us to compare the actual label with our predicted label values to calculate the predictive accuracy of the model.

Eventually, the accuracy of the model was printed as a percentage, and the accuracy was about 69.51%.