Naive Bayes

1. Task

We are given a multivariate classification data set, which is composed of 400 face images of size 64×64 pixels which is equal to 4096 pixels. These images are from 40 subjects, where we have 10 images from each subject. The class labels indicate the gender, 1 stands for "Female" and 2 stands for "Male". We are tasked to construct a Naive-Bayes classifier for these images, where we take the first 200 images as the training dataset and the remaining 200 as the test dataset.

2. Application

We are going to use Naive-Bayes classifier, which is based on the Bayes rule.

2.1. **Theory.** The Bayes rule is stated as

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \tag{1}$$

Writing this equation in terms of our dataset $X = \{x_i, y_i\}_{i=1}^{N}$ where x_i is an array of length 4096, which are the pixels of the image, and $y_i \in \{1, 2\}$ where 1 stands for "Female" and 2 stands for "Male".

$$P(y = c|x_i) = \frac{P(x_i|y = c)P(y = c)}{P(x_i)} \tag{2}$$

This is literally the equation that gives us the probability of y being c given x_i. So there are two cases, given x_i, either the class is 1 or 2. We can calculate $P(y = 1|x_i)$ and $P(y = 2|x_i)$ and by choosing the maximum between two, we can classify x_i. Note that the denominator $P(x_i)$ is same for both, so we dont even have to calculate it. For calculating $P(x_i|y = c)$ we can make the **naive** assumption that the the random variables are independent and thus $P(x_i|y = c) = P(x_{i,1}|y = c) \times P(x_{i,2}|y = c) \times ... \times P(x_{i,4096}|y = c) = \prod_{p=1}^{4096} P(x_{i,p}|y = c)$. This is the origin of **naive** in the name of the algorithm. One complication arises here, this multiplication becomes huge, so much that our computer wont be able to handle it. To overcome this, we can use logarithm. Note that $a > b \iff log(a) > log(b)$ where $a, b \in \mathbb{R}$ and both a, b are positive. Thus we can compare the likelihoods using logarithm to ease the computations.

$$\log(P(y = c|x_i)) = \log(P(x_i|y = c)P(y = c)) \tag{3}$$

We omitted the denominator as we stated before. Now we will write $P(x_i|y=c) = \prod_{p=1}^{4096} P(x_{i,p}|y=c)$, which gives us

$$\log(P(y=c|x_i)) = \log(\prod_{p=1}^{4096} P(x_{i,p}|y=c) \times P(y=c))$$

Recalling the property of logarithm that $\log(a \times b) = \log(a) + \log(b)$, we get

$$\log(P(y=c|x_i)) = \sum_{p=1}^{4096} \log(P(x_{i,p}|y=c)) + \log(P(y=c))$$

Here, $P(y=c)$ is called the prior probability. The estimate is calculated as:

$$\hat{P}(y=c) = \frac{\text{number of data belonging to class } c}{\text{number of data}}$$

which is pretty straightforward. As for $P(x_{i,p}|y=c)$ we assumed that the data is independent, and we will also assume they are normally distributed. For that, we calculate the estimate mean for each pixel $\hat{\mu}_{i,p}$ and the estimate standard deviation for each pixel $\hat{\sigma}_{i,p}$.After that, we get the normal distribution $P(x_{i,p}|y=c) = N(x_i; \hat{\mu}_{i,p}, \hat{\sigma}^2_{i,p})$. The formula for Gaussian Distribution is

$$N(x_{i,p}; \hat{\mu}_{i,p}, \hat{\sigma}^2_{i,p}) = \frac{1}{\sqrt{2\pi\hat{\sigma}^2_{i,p}}} \exp\left(-\frac{(x_{i,p} - \hat{\mu}_{i,p})^2}{2\hat{\sigma}^2_{i,p}}\right) \tag{4}$$

After we calculate all these, and compare $\log(P(y=1|x_i))$ and $\log(P(y=2|x_i)$ we can predict the class of x_i.

2.2. **Implementation.** We have our training data set in `X_train` and training labels in `y_train`, similarly for test data we have `X_test` and `y_train`. Of course, we are using `numpy`. We calculate the means as below:

```
class1_means = (np.sum(X_train_c1, axis = 0)/len(y_train_c1))
class2_means = (np.sum(X_train_c2, axis = 0)/len(y_train_c2))
```

Pretty straightforward. As for the standart deviations, we have written a more vectorized version. First recall the standart deviation formula

$$\text{stddev} = \sqrt{\frac{\sum_{p=1}^{4096}(x_{i,p} - \mu_{c,p})^2}{N}}$$

where c is the class and i is the image, so x_i is basically a vector of pixels.

```
class1_stddev = np.sqrt((
  np.sum(X_train_c1 * X_train_c1, axis = 0)
    - 2 * np.sum(np.dot(X_train_c1,
    np.dot(class1_means[:,np.newaxis], np.ones((1, len(class1_means))))
       * np.identity(len(class1_means))), axis = 0)
    + (len(y_train_c1) * (class1_means * class1_means)))
    /len(y_train_c1))
class2_stddev = np.sqrt((
  np.sum(X_train_c2 * X_train_c2, axis = 0)
    - 2 * np.sum(np.dot(X_train_c2,
    np.dot(class2_means[:,np.newaxis], np.ones((1, len(class2_means))))
      * np.identity(len(class2_means))), axis = 0)
    + (len(y_train_c2) * (class2_means * class2_means)))
    /len(y_train_c2))
```

What we did here is we basically used the fact that $(x_{i,p} - \mu_{c,p})^2 = x_{i,p}^2 - 2x_{i,p}\mu_{c,p} + \mu_{c,p}^2$. We can speak of three terms here: $\sum_{p=1}^{4096} x_{i,p}^2$ which is equal to x_i^2 in a vectorized fashion. The third term is $\sum_{p=1}^{4096} \mu_{c,p}^2$ which is equal to $4096 \times \mu_{c,p}^2$. The middle term is trickier. To explain that we will consider a matrix A and the mean vector for A over it's columns, like the way we do in our task, which we note as μ.

$$A = \begin{bmatrix} a & b & c \\ x & y & z \end{bmatrix} \text{ and } \mu = \begin{bmatrix} \frac{a+x}{2} & \frac{b+y}{2} & \frac{c+z}{2} \end{bmatrix} = \begin{bmatrix} \mu_1 & \mu_2 & \mu_3 \end{bmatrix}$$

We want to get an array that looks like

$$\begin{bmatrix} -2a\mu_1 - 2b\mu_2 - 2c\mu_3 & -2x\mu_1 - 2y\mu_2 - 2z\mu_3 \end{bmatrix}$$

For that, we could use `np.sum` for the rows in the matrix below

$$\begin{bmatrix} -2a\mu_1 & -2b\mu_2 & -2c\mu_3 \\ -2x\mu_1 & -2y\mu_2 & -2z\mu_3 \end{bmatrix} = -2 \times \begin{bmatrix} a\mu_1 & b\mu_2 & c\mu_3 \\ x\mu_1 & y\mu_2 & z\mu_3 \end{bmatrix}$$

To obtain that, we can the dot product:

$$\begin{bmatrix} a & b & c \\ x & y & z \end{bmatrix} \cdot \begin{bmatrix} \mu_1 & 0 & 0 \\ 0 & \mu_2 & 0 \\ 0 & 0 & \mu_3 \end{bmatrix} = \begin{bmatrix} a\mu_1 & b\mu_2 & c\mu_3 \\ x\mu_1 & y\mu_2 & z\mu_3 \end{bmatrix}$$

And to obtain the 3×3 matrix there, we can do

$$\begin{bmatrix} \mu_1 \\ \mu_2 \\ \mu_3 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} \mu_1 & \mu_1 & \mu_1 \\ \mu_2 & \mu_2 & \mu_2 \\ \mu_3 & \mu_3 & \mu_3 \end{bmatrix}$$

And finally,

$$\begin{bmatrix} \mu_1 & \mu_1 & \mu_1 \\ \mu_2 & \mu_2 & \mu_2 \\ \mu_3 & \mu_3 & \mu_3 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mu_1 & 0 & 0 \\ 0 & \mu_2 & 0 \\ 0 & 0 & \mu_3 \end{bmatrix}$$

In short,

$$\begin{bmatrix} -2a\mu_1 & -2b\mu_2 & -2c\mu_3 \\ -2x\mu_1 & -2y\mu_2 & -2z\mu_3 \end{bmatrix} = -2 \times A \cdot ((\mu^T \cdot O) \times I)$$

where O is an array of ones of the same length with μ, and I is the identity matrix. The code above is basically this operation using `numpy`, and then finally taking the sums per row, which is again vectorized during the computations, hence providing the second term we require. After this, we just calculate the probabilities using the functions below,

```python
# Normal distribution calculation
def normalDist(x, mean, stddev):
   return (1/(stddev * np.sqrt(2 * np.pi)))
          * np.exp(((mean - x)*(x - mean))/(2 * stddev * stddev))

# Naive-Bayes implementation
def bayes(X, mean, stddev, prior):
   P = normalDist(X, mean, stddev)
   P = np.log(P)
   return np.sum(P) + np.log(prior)
```

and make our predictions based on the greater probability. As for the confusion matrix, we also used another method. It is important here that our labels were 1 and 2.

		Prediction	
		1	**2**
Truth	**1**	True Ones	False Ones
	2	False Twos	True Twos

Let us denote True Ones as TO, False Ones as FO, False Twos as FT and True Twos as TT. We can manipulate the label vectors and use logical operations to quickly calculate these. As an example, consider $y = \begin{bmatrix} 1 & 1 & 2 & 2 & 1 \end{bmatrix}$ and $\hat{y} = \begin{bmatrix} 1 & 2 & 1 & 2 & 1 \end{bmatrix}$. For instance, to calculate TO we could set twos to be zero, hence indicate *False*; keep ones as ones, hence indicate *True* and then do element-wise logical and operation between y and $\hat{y}$. Finally, sum the elements in the vector, which would be the number of *True* values, therefore giving you the total number of True Ones. Here is this in action:

$$y' = \begin{bmatrix}2 & 2 & 2 & 2 & 2\end{bmatrix} - \begin{bmatrix}1 & 1 & 2 & 2 & 1\end{bmatrix} = \begin{bmatrix}1 & 1 & 0 & 0 & 1\end{bmatrix}$$
$$\hat{y}' = \begin{bmatrix}2 & 2 & 2 & 2 & 2\end{bmatrix} - \begin{bmatrix}1 & 2 & 1 & 2 & 1\end{bmatrix} = \begin{bmatrix}1 & 0 & 1 & 0 & 1\end{bmatrix}$$
$$y' \wedge \hat{y}' = \begin{bmatrix}1 & 1 & 0 & 0 & 1\end{bmatrix} \wedge \begin{bmatrix}1 & 0 & 1 & 0 & 1\end{bmatrix} = \begin{bmatrix}1 & 0 & 0 & 0 & 0\end{bmatrix}$$

Finally we do `numpy.sum` over this array, effectively giving us $1+0+0+0+0=1$ therefore $TO=1$. Notice that we subtracted our arrays from an array of twos. Using `numpy`, we can just have an array $\begin{bmatrix}2\end{bmatrix}$ and it will do broadcasting by itself when we write $\begin{bmatrix}2\end{bmatrix} - y'$. So subtracting from twos is the way to keep ones alive and reset twos.
To count the twos, we have to set them to 1 so that they indicate *True*, and set 1's to be 0's so that they indicate *False*. As simple as before, we can just subtract them from an array of ones. Below is an example of calculating True Ones.

$$y' = \begin{bmatrix}1 & 1 & 2 & 2 & 1\end{bmatrix} - \begin{bmatrix}1 & 1 & 1 & 1 & 1\end{bmatrix} = \begin{bmatrix}0 & 0 & 1 & 1 & 0\end{bmatrix}$$
$$\hat{y}' = \begin{bmatrix}1 & 2 & 1 & 2 & 1\end{bmatrix} - \begin{bmatrix}1 & 1 & 1 & 1 & 1\end{bmatrix} = \begin{bmatrix}0 & 1 & 0 & 1 & 0\end{bmatrix}$$
$$y' \wedge \hat{y}' = \begin{bmatrix}0 & 0 & 1 & 1 & 0\end{bmatrix} \wedge \begin{bmatrix}0 & 1 & 0 & 1 & 0\end{bmatrix} = \begin{bmatrix}0 & 0 & 0 & 1 & 0\end{bmatrix}$$

Finally we do `numpy.sum` over this array, effectively giving us: $0+0+0+1+0=1$ therefore $TT=1$. Similarly, calculating False Ones FO would be to keep ones alive in truth, and keep twos alive in the prediction in the process above. Here is the example:

$$y' = \begin{bmatrix}2 & 2 & 2 & 2 & 2\end{bmatrix} - \begin{bmatrix}1 & 1 & 2 & 2 & 1\end{bmatrix} = \begin{bmatrix}1 & 1 & 0 & 0 & 1\end{bmatrix}$$
$$\hat{y}' = \begin{bmatrix}1 & 2 & 1 & 2 & 1\end{bmatrix} - \begin{bmatrix}1 & 1 & 1 & 1 & 1\end{bmatrix} = \begin{bmatrix}0 & 1 & 0 & 1 & 0\end{bmatrix}$$
$$y' \wedge \hat{y}' = \begin{bmatrix}1 & 1 & 0 & 0 & 1\end{bmatrix} \wedge \begin{bmatrix}0 & 1 & 0 & 1 & 0\end{bmatrix} = \begin{bmatrix}0 & 1 & 0 & 0 & 0\end{bmatrix}$$

Finally we do `numpy.sum` over this array, effectively giving us $0+1+0+0+0=1$ Thus $FO=1$. The calculation of FT is similar.

Speaking in code terms, here is how it works:

```
two = np.array([2]) # [2]
one = np.array([1]) # [1]

conf_train = np.zeros((2, 2), dtype=int)
conf_train[1,1] = np.sum(np.logical_and(y_train - one, y_train_pred - one))
conf_train[1,0] = np.sum(np.logical_and(y_train - one, two - y_train_pred))
conf_train[0,1] = np.sum(np.logical_and(two - y_train, y_train_pred - one))
conf_train[0,0] = np.sum(np.logical_and(two - y_train, two - y_train_pred))

conf_test = np.zeros((2, 2), dtype=int)
conf_test[1,1] = np.sum(np.logical_and(y_test - one, y_test_pred - one))
conf_test[1,0] = np.sum(np.logical_and(y_test - one, two - y_test_pred))
conf_test[0,1] = np.sum(np.logical_and(two - y_test, y_test_pred - one))
conf_test[0,0] = np.sum(np.logical_and(two - y_test, two - y_test_pred))
```

3. Conclusion

In conclusion, Naive-Bayes classifier is a reasonable and mathematically sensible way of classification, as well as being easy to understand and implement, but we can not ignore the inadvertent consequences of not utilizing any kind of features that we could from images. A basic example would be that, human male and female usually have different bone structures, which would perhaps be extracted as a feature from the image and used in the classification process. In my opinion, it should not be used for classifying images where there might be some spatial features available, instead, maybe some classification regarding pure numeric values could be reasonable. It should also be better to actually know the our data is composed of independent variables, so that our naive assumption is not detrimental. Notwithstanding, our confusion matrixes show us that we have an accuracy of 87% for training, and 88% for the test data.

$$\frac{18+156}{18+2+24+156} = 87\%$$

$$\frac{15+161}{15+5+19+161} = 88\%$$