# COMP3055
# Machine Learning

**Explain the Solution to Lab 4**

**Ying Weng**

2024 Autumn

# Bayesian Learning

$$\boxed{p(h \mid x) = \frac{P(x \mid h)P(h)}{P(x)}}$$

Understanding Bayes' rule

$x$ = data

h = hypothesis (model)

- rearranging

$p(h \mid x)P(x) = P(x \mid h)P(h)$

$P(x, h) = P(x, h)$

the same joint probability

on both sides

$P(h)$: prior belief (probability of hypothesis *h* before seeing any data)

$P(x \mid h)$: likelihood (probability of the data if the hypothesis *h* is true)

$P(x) = \sum_h P(x \mid h)P(h)$: data evidence (marginal probability of the data)

$P(h \mid x)$: posterior (probability of hypothesis *h* after having seen the data *d*)

# Choosing Hypotheses

- Generally, we want the most probable hypothesis(class label) given the observed data
  - Maximum a posteriori (**MAP**) hypothesis
  - Maximum likelihood (**ML**) hypothesis

# Maximum A Posteriori (MAP)

- Maximum a posteriori (**MAP**) hypothesis

$$\boxed{p(h \mid x) = \frac{P(x \mid h)P(h)}{P(x)}}$$

$$\boxed{h_{MAP} = \arg\max_{h \in H} p(h \mid x) = \arg\max_{h \in H} \frac{P(x \mid h)P(h)}{P(x)} = \arg\max_{h \in H} P(x \mid h)P(h)}$$

Note $P(x)$ is independent of $h$, hence can be ignored.

# Maximum Likelihood (ML)

$$h_{MAP} = \arg\max_{h \in H} P(x \mid h) P(h)$$

- Assuming that each hypothesis in $H$ is equally probable, i.e., $P(h_i) = P(h_j)$, for all i and j, then we can drop $P(h)$ in MAP. $P(x|h)$ is often called the likelihood of data $x$ given $h$. Any hypothesis that maximizes $P(x|h)$ is called the maximum likelihood hypothesis

$$h_{ML} = \arg\max_{h \in H} P(x \mid h)$$

# Estimating Probabilities

- When $Nc$ is small, however, such approach provides poor estimation. To avoid this difficulty, we can adopt the **m-estimate** of probability

$$\boxed{\frac{N_C + mP}{N + m}}$$

where $P$ is the prior estimate of the probability we wish to estimate, $m$ is a constant called the equivalent sample size, which determines how heavily to weight $P$ relative to the observed data.

A typical method for choosing $P$ in the absence of other information is to assume uniform priors: If an attribute has $k$ possible values we set $P=1/K$.

# Equivalent Sample Size (ESS)

- Equivalent Sample Size (ESS) is a statistical concept, referring to the number of observations required in a study to achieve a certain level of precision or reliability.
- ESS is particularly useful in survey sampling and experimental design, where researchers aim to estimate population parameters based on a limited number of observations.
- ESS provides a way to compare different sampling methods and their effectiveness in capturing the true characteristics of a population.

# Naïve Bayesian Classifier

- What about using Naïve Bayesian Classifier for our handwritten digit recognition problem?
  - Each pixel is an $x_i$. There will be 784 $x$'s.
  - Digit label is $d_k$. Note there will be 10 possible $d$'s.
  - $\boldsymbol{P(d_k)}$ can be calculated by counting number of training images for the digit, divided to total number of training images.
  - $\boldsymbol{P(x_i|d_k)}$ can be calculated by counting number of images for a given digit, given pixel position, and given an intensity value, divided by number of training images with that digit.

# Naïve Bayesian Classifier

- For a given input image $\boldsymbol{X}$ and given digit label $\boldsymbol{d_k}$, calculate $P(d_k)$ and all $P(x_i|d_k)$
- For each digit label $d_k$, calculate
  $$\boldsymbol{P(d_k|X) = P(d_k)P(x_1 = 0|d_k)P(x_2 = 255|d_k) \dots P(x_{784} = 0|d_k)}$$
- Choose the digit label k that give the **max value** according to above calculation.

Input image X

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 255 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 255 | 255 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 255 | 0 | 0 | 255 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 255 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 255 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 255 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 255 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 255 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 255 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 255 | 255 | 255 | 255 | 255 | 255 | 0 |

# All imports in this lab

```
from sklearn.datasets import fetch_openml
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter
import math
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import ComplementNB
from sklearn.naive_bayes import GaussianNB
```

# Dataset

```
# load the MNIST 784 dataset
X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
y = y.astype(int)

# train test split
x_train, y_train = X[:60000], y[:60000]
x_test, y_test = X[60000:], y[60000:]

# obtain a small set for the lab exercise
X_small = x_train[:1000].to_numpy().reshape((1000, 784))
Y_small = y_train[:1000]
X_test = x_test[:100].to_numpy().reshape((100, 784))
Y_test = y_test[:100]
```

# Tricks

- **TRICK 1:** treat **brightness as continuous value** to reduce calculation complexity

- **TRICK 2:** take log(x1)+log(x2)+... to instead of x1·x2·... to avoid decreasing to zero

- **TRICK 3:** only consider **two situations** for pixel, dark or bright, to be consistent with TRICK 1

# prioriP

```
# prior probability - P(d)
totalNum = train_x.shape[0]
classNum = Counter(train_y)
# P(d=i) = total number of class i / total number of all classes
prioriP = np.array([classNum[i] / totalNum for i in range(10)])
```

- Total number of samples: 1000
- Sample count per class: Counter({7: 117, 1: 116, 4: 105, 9: 100, 2: 99, 0: 97, 6: 94, 3: 93, 5: 92, 8: 87})
- Prior probabilities (P(d)): [0.097 0.116 0.099 0.093 0.105 0.092 0.094 0.117 0.087 0.1 ]

# posteriorP

```
 # posterior probability - P(X|d), also a set of P(xi|d)
   # create an empty array with shape of (10, 784) where 10 refers to the number of classes and 784 refers to the number of features
   posteriorNum = np.empty((10, train_x.shape[1]))
   posteriorP = np.empty((10, train_x.shape[1]))

   for i in range(10):
       # TRICK 1: treat brightness as continuous value to reduce calculation complexity
       posteriorNum[i] = train_x[np.where(train_y == i)].sum(axis=0)
       # m-estimation smoothing
       posteriorP[i] = (posteriorNum[i] / 255 + m/255) / (classNum[i] + m)
```

# Bayes_train

```
def Bayes_train(train_x, train_y, m):
    # prior probability - P(d)
    totalNum = train_x.shape[0]
    classNum = Counter(train_y)
    # P(d=i) = total number of class i / total number of all classes
    priorP = np.array([classNum[i] / totalNum for i in range(10)])

    # posterior probability - P(X|d), also a set of P(xi|d)
    # create an empty array with shape of (10, 784) where 10 refers to the number of classes and 784 refers to the number of features
    posteriorNum = np.empty((10, train_x.shape[1]))
    posteriorP = np.empty((10, train_x.shape[1]))

    for i in range(10):
        # TRICK 1: treat brightness as continuous value to reduce calculation complexity
        posteriorNum[i] = train_x[np.where(train_y == i)].sum(axis=0)
        # m-estimation smoothing
        posteriorP[i] = (posteriorNum[i] / 255 + m/255) / (classNum[i] + m)
    return priorP, posteriorP
```

# Bayes_pret

```
def Bayes_pret(test_x, test_y, prioriP, posteriorP):
  # create an empty array for recording the predictions
  pret = np.empty(test_x.shape[0])
  for i in range(test_x.shape[0]):
    # create an empty array for recording the probability of each class for the i th testing sample
    prob = np.empty(10)
    for j in range(10):
      # TRICK 2: take log(x1)+log(x2)+... to instead of x1·x2·... to avoid decreasing to zero
      # TRICK 3: only consider two situations for pixel, dark or bright, to be consistent with TRICK 1
      # if a pixel is bright (!=0), we take the computed posterior probability, otherwise 1 minus it
      temp = sum([math.log(1 - posteriorP[j][x]) if test_x[i][x] == 0 else math.log(posteriorP[j][x]) for x in
            range(test_x.shape[1])])
      prob[j] = np.array(math.log(prioriP[j]) + temp)

      # you can also try below code for the multiplication (without TRICK 2)
      # p = 1.
      # for x in range(test_x.shape[1]):
      #    temp = 1 - posteriorP[j][x] if test_x[i][x] == 0 else posteriorP[j][x]
      #    p = p * temp
      # prob[j] = prioriP[j] * p

    pret[i] = np.argmax(prob)   # get the digit with most probability
  return pret, (pret == test_y).sum() / test_y.shape[0]
```

# Any Questions?