BAYES CLASSIFIER FOR A SUM OF DIGITS ON IMAGES

*Abstract*

When a problem of calculating the sum of digits on images arises, the obvious approach when using a Bayesian classifier would be to recognize all the digits one by one and calculate their sum. This way of solving the problem is not the most accurate one, and a better solution arises directly from Bayesian decision theory.

# **1. Bayesian decision theory**

### 1.1 Bayes’ rule

The Bayes Rule (or Bayes’ Theorem) is the basis for Bayesian decision theory, and enables one to describe the probability of an event based on certain prior knowledge. The theorem is stated as a following mathematical formula:

. (1)

In this case, is called the **posterior probability**, and  - **prior probability**. A common example of this rule’s application is detecting spam emails:

Example

Let us assume that in spam letters, the probability of occurrence of a certain word *w* is *p*, while in general, this word occurs in emails with probability . Let us also assume that the prior probability of an email being a spam one (in this case, the prior probability is the probability before accounting for email’s contents) is 0.2. The task is to determine the probability of an email with the word *w* in it being spam.

So, we need to determine the following probability:

With the help of formula (1):

This shows the power and simplicity of the Bayes’ rule.

### 1.2 Formulating the task

Let us imagine an object that has two parameters – *x* and *k*. *x* is a visible parameter, while *k* is hidden. Both parameters take values from finite sets *X* and *K*, respectively. It doesn’t matter now, what values exactly do those parameters take: in our example, *x* was the content of an email and *k* took only two values: *“Email is not spam”* and *“Email is spam”*.

Let us also assume that the joint distribution function is defined such that for all and , is the probability of the visible parameter taking value x and the hidden parameter taking value *k*.

Now, let us set a task: guess the value of *k* based on the value of *x*. The guess will be denoted as , where is the set of all possible guesses.

We need to define two functions:

* **Loss function**  such that is the value of the penalty taken when we make decision *d* and the hidden parameter value is *k*;
* **Strategy function** (referred to as **strategy** further on in this paper) such that it represents the decision .

The expectation of the loss function when using strategy is called the **risk** of the strategy and is denoted by :

The Bayesian decision task is finding a certain strategy *q* which minimizes the risk *R(q)* for a fixed set of visible and hidden parameter values *X* and *K*, fixed set of possible decisions (guesses) *D,* fixed joint probability distribution function and fixed loss function .

### 1.3 Types of loss functions

Choosing a loss function is an important step in solving our task. Some of the common ones are:

* Quadratic loss function: 
* Absolute loss function: 
* “0-1” loss function: 

Each one of these loss functions has its pros and cons:

* 0-1 function does not account for the distance between the decision and the true value (though a downside for our task, this property can be useful when dealing with problems, where distance isn’t defined);
* Quadratic loss function does account for distance and is easily differentiable, but is highly sensitive to outliers;
* Absolute loss function is less sensitive to outliers than the quadratic function, but it isn’t differentiable at point 0 and thus is harder to work with.

For our task, we will pick the absolute loss function.

# **2. Developing the algorithm**

The solution of the recognition task is made up of 2 parts: training the classifier and solving the task itself. The second part will be presented first, as it is more difficult, while the training procedure is the same as in any other Bayesian classifier.

### 2.1 Classification task

The input of our classifier will be a sequence of images. Let us denote the set of all possible sequences of images as, the images themselves -; set of all possible sequences of digits on the images as, the digits themselves – *K.*

Then, the following is true for the joint probability distribution:

,

as the images are independent of one another, as are the digits.

The risk of strategy is:

.

On the previous page, we chose the Absolute loss function, and we can adjust the risk formula accordingly:

,

where is the set of all sequences of *K* that sum up to *d.*

Let  . Then,

 (2.1)

This is a convex function, linear at intervals . Thus, its minimum is achieved in a point *q* such that:

, (2.2)

 (2.3)

Then we can rewrite (2.1) as:

 (2.4)

and

 (2.5)

If in (2.5) we substitute , we get:







.

Then, from (2.3):



 (2.6)

Also,

, (2.7)

as the left part is the sum of probabilities.

From (2.6) and (2.7):

 (2.8)

Similarly, for :







.

Substituting this into (2.2), we get:



 (2.9)

From (2.7):

. (2.10)

A number *q* such that it satisfies (2.8) and (2.10) is the solution of the risk minimization problem. A logical decision would be to calculate  first, then  and so on. Now, let us prove, that the first *q*, that satisfies (2.8), also satisfies (2.10):

As , from (2.7) we have:

 +  

As *q* satisfies (2.8),



.

Thus, a *q* that satisfies (2.8), also satisfies (2.10).

So, an algorithm for minimizing the loss function was developed. Later, we will empirically prove that for a fixed loss function this algorithm has a smaller error rate than the intuitive algorithm of recognizing each digit one-by-one and adding them.

**2.2 Training the model**

For calculating  we need the posterior probabilities . Using Bayes’ formula:

. (2.11)

The  probability distribution is easy to estimate from the training data. Let us denote the number of images in the training set as . Then:

.

Now, we need to find the parameters of distribution of *X*.

The MNIST dataset contains images sized pixels. We assume, that for each digit, the pixels form a Gaussian vector of size . Thus, we have to estimate the parameters of 10 multivariate gaussian distributions. We will also add an independent gaussian noise with known parameters to each image. The estimation can be done from the data:

,

where — the set of images, of the digit *k.*

,

The covariance of *i*th and *j*th pixels is estimated by an average of the product of their distance from the estimated mean. For each image, a covariance matrix is formed this way.

It is important to note, that due to approximation not being a perfect reflection of reality, the estimated covariance matrix might not be positive-definite. In this case, a regularization is performed [4]:

,

where  is an identity matrix of the same size as , and is a relatively small number. In this work, we will take.

**2.3 Recognition**

In this section, we will optimize the algorithm for application in digit-sum recognition and review the application itself.

The first step is to account for added gaussian noise. It is independent from the images, and thus for any:

,



Let us now use all the known values to find the multivariate normal distribution of *X* conditioned on *K* (of images conditioned on the digit pictured)*.*

. (2.12)

If  and are known, we can find utilizing the total probability formula:



Now we need to calculate.

As stated above, . Considering how expensive resource-vise the upfront calculation of this formula would be, and the fact that this value would have to be calculated for each value of the parameter *d* (which is every possible sum of *n* digits, where *n* is the number of images on input) it makes sense to optimize it.

Let us define *n* functions:

,



,



.

It is clear that . Now let us define through:







.

This formula makes it possible to calculate recursively, which is much faster.

**3. The code**

In this section, the algorithm will be implemented in Python using NumPy and SciPy libraries.

**3.1 The dataset**

We will need 2 datasets: a training one and a testing one. The MNIST dataset, used in this work, contains a training set of 60000 images and labels and a smaller testing set. Each image is sized, and each pixel takes values between 0 and 255.

MNIST library was used to read the MNIST files into program memory:

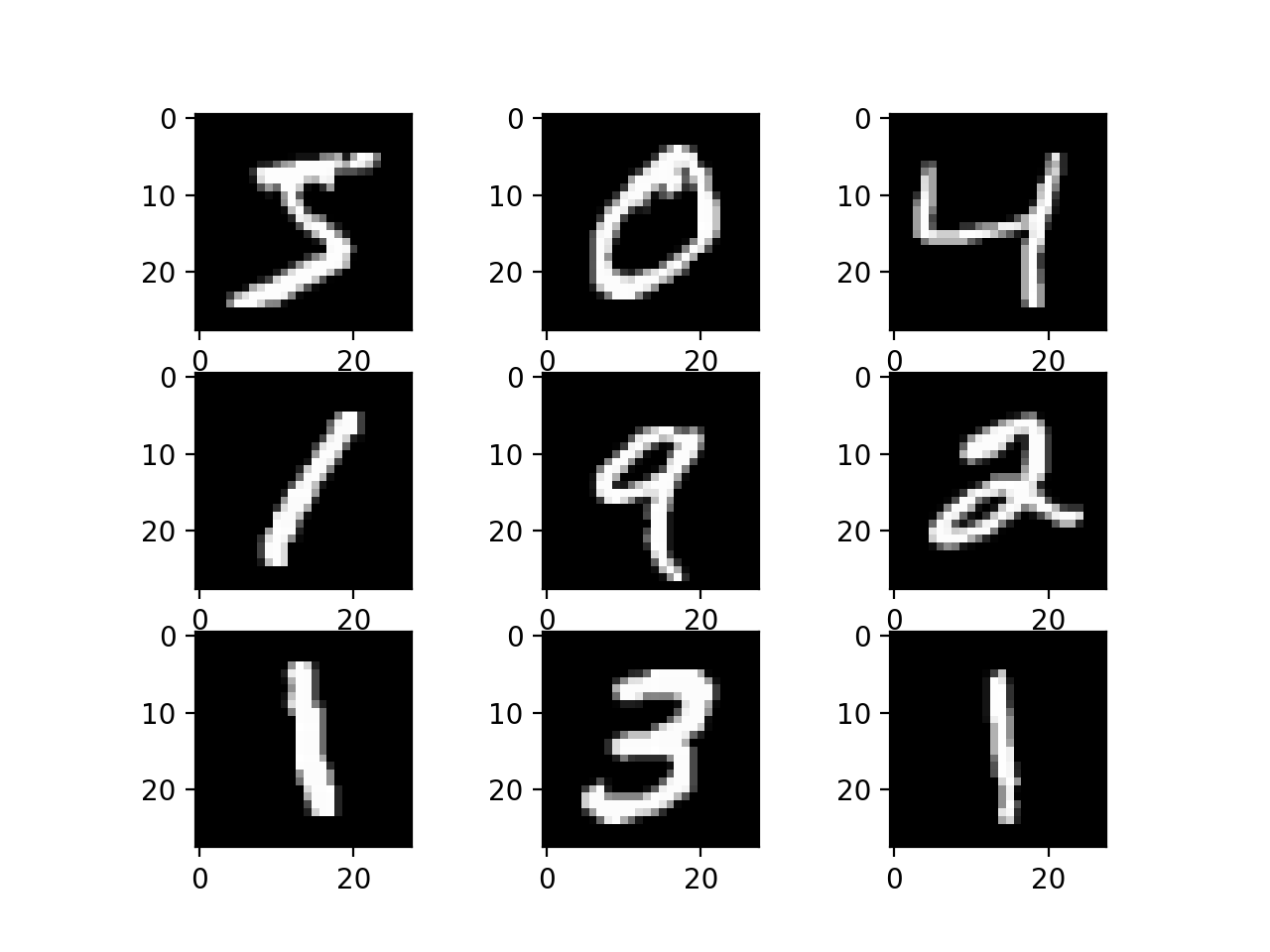
**from** mnist **import** MNIST  
**def** get\_training\_files(path):  
 mndata = MNIST(path)  
 mndata.gz = **True  
 return** mndata.load\_training()  
  
**def** get\_testing\_files(path):  
 mndata = MNIST(path)  
 mndata.gz = **True  
 return** mndata.load\_testing()

Рисунок 2 – приклад зображень з датасету MNIST

mn\_images, mn\_labels = get\_training\_files(**'./'**)  
mn\_images\_testing = np.array(get\_testing\_files(**'./'**)[0])  
mn\_labels\_testing = np.array(get\_testing\_files(**'./'**)[1])

We will need a function to group the images by the digit presented:

**def** group\_mnist\_images(mn\_images, mn\_labels):  
 mn\_images\_grouped = [[], [], [], [], [], [], [], [], [], []]  
 **for** i **in** range(len(mn\_images)):  
 mn\_images\_grouped[mn\_labels[i]].append(mn\_images[i])  
 **return** mn\_images\_grouped

Use the function:

mn\_images\_grouped = group\_mnist\_images(mn\_images, mn\_labels)

Take 100 images from the testing set:

n\_testing = 100  
mn\_images\_testing = np.take(indices=arange(0, n\_testing), a=mn\_images\_testing, axis=0)  
mn\_labels\_testing = np.take(indices=arange(0, n\_testing), a=mn\_labels\_testing, axis=0)

Map the pixel values to 0-1 interval:

mn\_images\_testing = mn\_images\_testing / (255)  
  
**for** i **in** range(len(mn\_images\_grouped)):  
 mn\_images\_grouped[i] = (np.array(mn\_images\_grouped[i])) / (255)

**3.2 Parameter approximation:**

**def** estimate\_properties(mn\_img\_grouped):  
 means = []  
 stdevs = []  
 covs = []  
 **for** num **in** range(len(mn\_img\_grouped)):  
 means.append(np.mean(a=mn\_img\_grouped[num], axis=0))  
 covs.append(np.cov(m=mn\_img\_grouped[num], rowvar=**False**, bias=**False**))  
 stdevs.append(find\_stdev(mn\_img\_grouped[num], means[num]))  
 **return** np.array(means), np.array(stdevs), np.array(covs)

Use it, and regularize the covariation matrix:

means, stdevs, covs = estimate\_properties(mn\_images\_grouped)  
*# making sure the covariance matrix is invertible*covs = covs + 0.1 \* np.identity(784)

Add the noise:

*# generate noise*noise\_means = np.average(means, axis=0) \* 0.5  
noise\_covs = np.average(covs, axis=0) \* 0.5  
*# apply noise*mn\_images\_testing += multivariate\_normal(mean=noise\_means, cov=noise\_covs).rvs(n\_testing)

**3.3 Recognition**

*# probabilities of each image conditioned on each training image set: p(X=x|K=k)*gau\_images\_probabilities\_cond = np.empty(shape=(10, np.size(axis=0, a=mn\_images\_testing)))  
**for** i **in** range(0, 10):  
 *# P(X=x|K=k)* gau\_images\_probabilities\_cond[i] = (  
 multivariate\_normal(mean=means[i] + noise\_means, cov=covs[i] + noise\_covs).pdf(x=mn\_images\_testing))  
 *# probability of each number: p(K=k)* gau\_digits\_probabilities[i] = np.size(axis=0, a=mn\_images\_grouped[i]) / images\_count  
*# joint probabilities: p(x, k)*gau\_joint\_probabilities = gau\_images\_probabilities\_cond.T \* gau\_digits\_probabilities.T  
*# total probs of x: p(x)*gau\_images\_probabilities = np.sum((gau\_images\_probabilities\_cond.T \* gau\_digits\_probabilities.T), axis=-1)  
*# conditional probabilities of each number conditioned on each image: list of p(k|x)*gau\_digits\_probabilities\_cond = gau\_joint\_probabilities \* (  
 np.array([1 / gau\_images\_probabilities] \* gau\_digits\_probabilities[0].size).transpose())

A recursive function for calculating:

**def** calculate\_f(pkx, j, r, ks, fj\_arr):  
 **if** r < 0:  
 **return** 0  
 **if** fj\_arr[r][j] **is not None**:  
 **return** fj\_arr[r][j]  
 **if** j == 1:  
 **if** r > ks.size - 1 **or** r < 0:  
 **return** 0  
 **else**:  
 **return** pkx[0][r]  
 **else**:  
 f\_sum = 0  
 **for** k **in** ks:  
 f\_sum += pkx[j - 1][k] \* calculate\_f(pkx, j - 1, r - k, ks, fj\_arr)  
 fj\_arr[r][j] = f\_sum  
 **return** f\_sum

Solve the Bayesian recognition task – find the *q*:

fj\_arr = np.array([[**None**] \* (n\_testing + 1)] \* (n\_testing \* 9 + 1))  
**for** q **in** range(0, n\_testing \* 9 + 1):  
 fj\_arr[q][1] = gau\_digits\_probabilities\_cond[0][q] **if** q < arange(0, 10).size **else** 0  
s = 0  
q = 0  
*# loss function = |d\*-d|***while** q < n\_testing \* 9 + 1:  
 s += calculate\_f(gau\_digits\_probabilities\_cond, n\_testing, q, arange(0, 10), fj\_arr)  
 **if** s >= 0.5:  
 **break** q += 1

print(**"Estimated sum:"**)  
print(q)  
print(**"True sum:"**)  
print(np.sum(mn\_labels\_testing))

**3.4 Results**

We can achieve different noise levels by using  та  multiplied by some constant *C,* as the parameters for noise distribution.

For comparison, let us also try using the “intuitive approach” of recognizing each digit separately and calculating the sum. As shown in Table 1, this approach yields lower accuracy.

***Table 1 – the results for different noise parameters***

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N. of images | True sum value | Sum value as recognized by the “correct” algorithm | Sum value as recognized by the intuitive algorithm | Noise level *С* |
| **10** | **42** | **42** | **42** | **0.1** |
| **10** | **42** | **42** | **42** | **1** |
| **10** | **42** | **91** | **100** | **5** |
| **100** | **448** | **455** | **456** | **0.1** |
| **100** | **434** | **432** | **428** | **1** |
| **100** | **434** | **901** | **1000** | **5** |