Solving Sudoku: An Application of Handwritten Digit Classification

Paulito Mendoza, *ECE531 Project Report*
UIC Spring 2017 Prof. Mojtaba Soltanalian

***Abstract*—Character recognition is a common operation in modern administrative operations such as document scanning and conversion, and check deposits in banking transactions. This project will illustrate Bayes classifier methods trained on the well-known MNIST digit database to identify handwritten digits.**
To illustrate an entertaining example of character recognition: the Bayes classifier approach will be used to (attempt to) identify the digits of a scanned Sudoku puzzle, as part of an automated Sudoku puzzle solving application.

***Index Terms*—naïve Bayes classifier, Sudoku, MNIST, handwritten digit recognition.**

I. Introduction

As an example of estimation and detection concepts, handwritten digit classification has clear and obvious value to the modern workplace. Converting handwritten numbers to electronic format – and, in general, handwriting recognition of any text – has been used in such applications as postal mail sorting, handwritten document conversion, and bank check scanning.

A number of modern techniques (e.g. [1], [2]) exist that provide an error rate significantly small enough to be employed successfully in commercial applications as previously described. This project, however, intends to present character recognition from a statistical "first principles" approach, using Bayesian classification. While the error rate is not comparable to those more modern techniques, it is a useful baseline from which to consider those modern methods.

A secondary consideration for this project (and of more entertainment value) is to implement a digit classification approach that is simple enough to be modeled in a MATLAB environment, and migrated to a Windows application designed to solve Sudoku puzzles: images of unsolved, handwritten Sudoku puzzles will be scanned, parsed into digit bitmaps, and an electronic representation of the puzzle will be solved by a simple recursive algorithm.

II. MNIST Digit Database

The Modified National Institute of Standards and Technology (MNIST) database [3] was created specifically for the purpose of providing a handwritten digit database for use in machine learning and consists of several thousand 28x28 grayscale (8-bit) pixel images of handwritten digits, separated into a training set of images, and a test set of images. This database is actually derived from the original NIST database, which comprised a larger number of black-and white images and were segregated between a training set population (American census workers) and a test set population (American high school students). The MNIST database also mixed the source populations across the training and test sets.

Figure 1 shows an example of images from the MNIST database. Note that the images are "inverted", i.e. the digits represent high pixel intensity (~255) and the background is represented by low pixel intensity (~0).

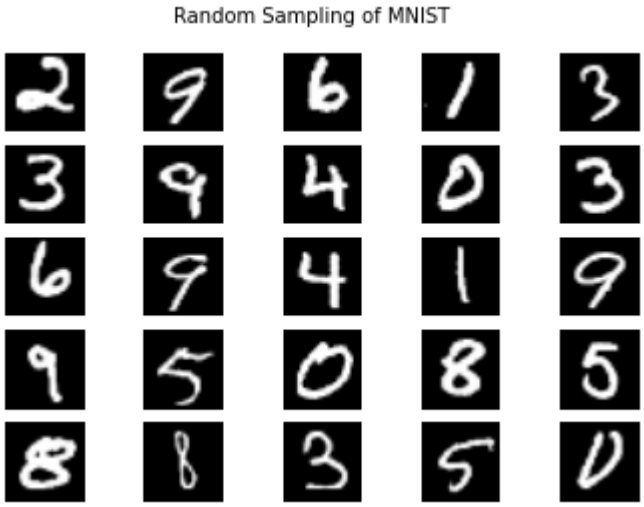
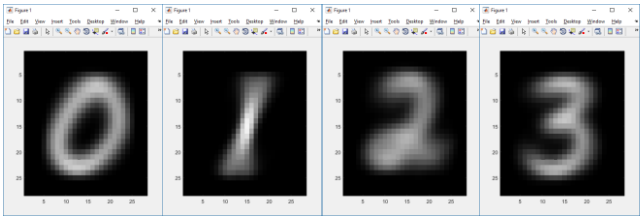


Figure 1. Example of MNIST images.

This paper used an abbreviated version of the MNIST database [4] which was also decomposed into an easier file format; this abbreviated database consists of 1,000 samples of each digit 0 through 9 (a total of 10,000 images).

To illustrate the consistency and clarity of the images, as a whole, for each digit: an "average" digit image was created in MATLAB by generating the arithmetic mean of each pixel, across the 1,000 samples of each image. These averaged images are shown below; they are very easy to identify.



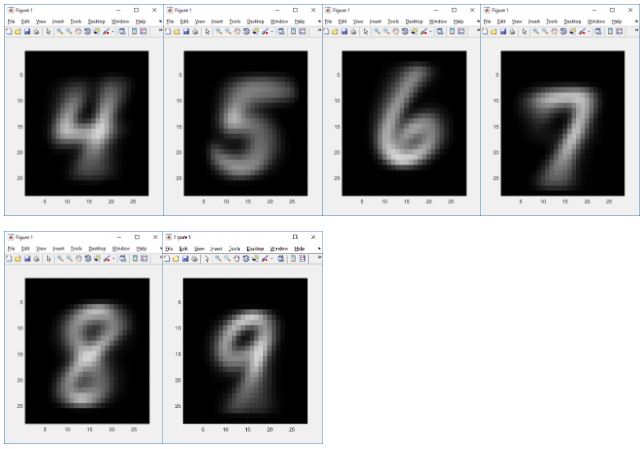


Figure 2. "Averaged" MNIST digit images.

III. ALGORITHMS

A. Bayesian Classifier Approach

The well-known Bayes Rule defines the probability of an event based on observations related to that event:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

Its use is prevalent in the earliest email spam filters, where the probability of an email message being classified as spam can be estimated from the probability that certain "spammy" words found in the message may occur in true spam messages, as well as the probability of spam messages, and the probability of those words in general:

$$P(IsSpam \mid SpamWords) = \frac{P(SpamWords \mid IsSpam)P(IsSpam)}{P(SpamWords)}$$

In a spam filtering application, a large training dataset is required, in order to estimate these *a priori* probabilities.

The Bayes approach in digit recognition is similar: the probability of a bitmap image *x* being classified as the digit *i* is dependent on the probability of that bitmap image *x* occurring, given that it is expected to represent the digit *i*, as well as the probability of the digit *i* occurring, and the probability of the bitmap *x*:

$$P(Digit\_i \mid Bitmap\_x) = \frac{P(Bitmap\_x \mid Digit\_i)P(Digit\_i)}{P(Bitmap\_x)}$$

B. MAP Estimator for Classifying Bitmaps

From the Bayes expression, we can classify a given bitmap *x* as a specific digit from the maximum *a posteriori* probability

$$Digit = \arg\max_i P(Digit\_i \mid Bitmap\_x)$$

In other words, the bitmap *x* is estimated to be the digit that maximizes the probability of its bitmap occurring as a representation of that digit.

Substituting the Bayes expression into the estimator above, we obtain

$$= \arg\max_i \frac{P(Bitmap\_x \mid Digit\_i)P(Digit\_i)}{P(Bitmap\_x)}$$

We can further simplify the expression above by (1) assuming that all digits have equal probability, i.e. P(*Digit_i*) is the same for all digits , and (2) observing that P(*Bitmap_x*) is not dependent on *i*. These terms can be eliminated to obtain

$$= \arg\max_i P(Bitmap\_x \mid Digit\_i)$$

The classification problem is thus simplified to determining the probability that a given bitmap *x* represents a digit *i* (for all digits 0 through 9) and then identifying the digit generating the maximum probability.

C. Binary Naïve Bayes

In the binary naïve Bayes algorithm, each pixel is represented by a Bernoulli random variable: $p_{i,j}$ is the probability that pixel *j* of digit *i* is ON, i.e.

$P_i(x_j) = p_{i,j}$	probability that pixel is ON
$P_i(x_j) = (1 - p_{i,j})$	probability that pixel is OFF

and is calculated by summing the values (ON or OFF) of pixel *j* over all bitmap images of digit *i* in the image training database.

Thus we can state that, for digit *i*, the probability of observing a specific bitmap X (i.e. a bitmap of 28x28 = 784 pixels representing a unique image) is the probability of each pixel taking on the value within that bitmap, based on the training dataset of images for digit *i*, defining the probabilities of the pixels:

$$P(Bitmap\_x \mid Digit\_i) = P_i(x_0, x_1, x_2 ..., x_{783})$$

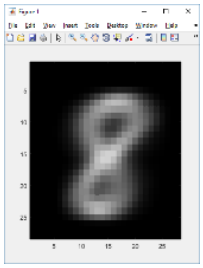
If we assume each pixel state is independent (a *naïve* assumption; thus, the name), we can simplify this expression as

$$= P_i(x_0)P_i(x_1)P_i(x_2)...P_i(x_{783})$$

$$= \prod_{j=0}^{783} P_i(x_j)$$

Note that, since the images are 8-bit grayscale, the pixel ON state would be defined as the pixel intensity value greater than some threshold, e.g. 128.

As an example: consider pixel values in the "averaged" digit image 8, created from the arithmetic mean of the pixels of all images representing the digit 8:



- the upper-left corner pixel at coordinates (x,y)=(0,0) is always OFF; if we consider this pixel 0, then
  - $p_{8,0} = 0$
- the center pixel at coordinates (x,y)=(15,15) is (almost) always ON; let's assume this is pixel index 434 out of 784. Its probability is
  - $p_{8,434} = 0.9558$
- the pixel at coordinates (x,y)=(10,8) (index 287) , which is roughly the upper left corner of the digit shape, is *sometimes* on, so its probability is
  - $p_{8,287} = 0.5391$

Given our bitmap X: if its pixel values are

- x_0 = OFF
- x_{434} = ON
- x_{287} = OFF

Then the probability of bitmap X's three pixels representing digit 8 is $(1 - p_{8,0}) * (p_{8,434}) * (1 - p_{8,287})$ or 1 * 0.9958 * 0.4609.

The calculation is carried across all 28 x 28 = 784 pixels, to compute the probability of a given bitmap.

Note that, due to the large number of very small probabilities, *underflow* may occur; and instead of multiplying pixel probabilities, we sum the log of the probabilities:

$$= \sum_{j=0}^{783} \ln P_i(x_j)$$

D. Gaussian Naïve Bayes

The Gaussian naïve Bayes approach is similar; however, instead of modeling each pixel as a Bernoulli random variable, we derive a Gaussian distribution of the pixel's 8-bit intensity value, based on the value of that pixel in all of the images for a given digit.

So from the training dataset of images, we derive, for each digit and each pixel for that digit

- mean $\mu_{j,i}$: average value of pixel *j* of digit *i* over all samples
- variance $\sigma_{j,i}$: average value of $(\text{pixel}_{j,i} - \mu_{j,i})^2$ over all samples

and we obtain an expression for the probability of a given bitmap representing a digit *i* as

$$P(Bitmap\_x \mid Digit\_i) = \prod \frac{1}{\sqrt{2\pi\sigma_{j,i}^{\,2}}} e^{-\frac{(x_{j,i}-\mu_{j,i})^2}{2\sigma_{j,i}^{\,2}}}$$

as before: we avoid calculation underflow by taking the log:

$$\ln P(Bitmap\_x \mid Digit\_i) = \sum \ln \frac{1}{\sqrt{2\pi\sigma_{j,i}^{\,2}}} e^{-\frac{(x_{j,i}-\mu_{j,i})^2}{2\sigma_{j,i}^{\,2}}}$$

$$= \sum \left(\ln \frac{1}{\sqrt{2\pi\sigma_{j,i}^{\,2}}} - \frac{(x_{j,i}-\mu_{j,i})^2}{2\sigma_{j,i}^{\,2}} \right)$$

E. Multivariate Gaussian

An alternative approach to these Bayesian models is a multivariate Gaussian distribution, where each 28x28 pixel image is treated as a vector of 784 random variables.

In theory, we expect this multivariate Gaussian distribution may do better than the naïve Bayes approaches, since the latter assume independence of each pixel (in the spam application, this independence has been referred to as the "bag of words" approach: the order of the spam words does not matter, and similarly, the order of the pixels in each digit does not matter, just the aggregate resulting probability value). A multivariate Gaussian distribution calculates the covariance of pixel values, and thus is retained a relationship of pixels to one another.

F. MATLAB Results

MATLAB implementations of the binary naïve Bayes, Gaussian naïve Bayes, and multivariate Gaussian approaches were implemented and then applied to the same abbreviated dataset of images used to train the models.

TABLE I. MATLAB RESULTS

ALGORITHM	SUCCESS RATE
Binary Bayes	83%
Gaussian Bayes	79%
Multivariate Gaussian	63%

Note that the multivariate Gaussian performed significantly worse than the Bayesian approaches; it is quite possible that this is due to errors in the MATLAB implementation. Since the multivariate Gaussian approach also involves inversion of a 784x784 covariance matrix, it was computationally intensive (approximately 2.5 hours to test the 10,000 dataset images!).

For these reasons, the results of this implementation were not migrated to the Windows platform as part of the Sudoku application of digit recognition.

IV. SUDOKU SOLVER APPLICATION

In this second part of the project, artifacts from the digit recognition algorithms are exported from MATLAB, in order to be used to recognize digits in an arbitrary (unsolved) hand-written Sudoku puzzle, so that a Sudoku solving algorithm could be executed to solve the puzzle.

An example of a handwritten Sudoku puzzle image that can be processed by the Sudoku solver application is as follows:

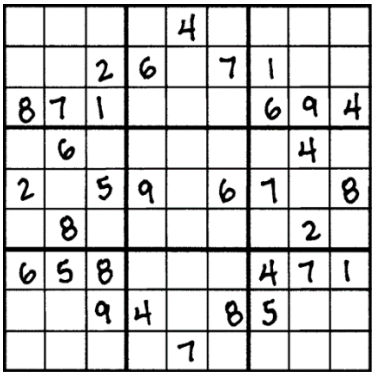


Figure 3. Sudoku puzzle to be solved.

A. Sudoku Solving Algorithm

While there exist [5][6][7] more sophisticated, formal mathematical approaches to solving Sudoku, this project uses an "ad hoc" algorithm that mimics a common, intuitive method of manual solving: a recursive backtracking algorithm.[1]

The algorithm assumes an abstraction of the 9x9 Sudoku puzzle into cells and gridsets:

- Each *cell* can either be blank, or is populated with a number
- Each cell belongs to a *row*, a *column*, and a *square*; these collection types might be considered *derived classes* of a higher-order *parent class*, a *gridset*. Thus, each cell belongs to three gridsets; in a solved Sudoku puzzle, the cell's number must be unique across its three gridsets, i.e. unique amongst the row, column, and square to which the cell belongs

This abstraction lends itself well to an object-oriented implementation such as C++.

The recursive backtracking algorithm is defined in two steps, as follows:

1. *Deterministic cell population.*
   a. Iterate over each unpopulated cell in the grid:
      i. Define that cell's possible values, based on the values already present in the other (populated) cells of that cell's row, column, and square (ie the cells comprising its three gridsets)
   b. Iterate over each unpopulated cell in the grid:
      i. If there is a cell with a single possible value: populate that cell with the value, and restart this algorithm
   c. If there are no more unpopulated cells
      i. Puzzle is solved; return SUCCESS and the resulting solution grid
   d. Else
      i. Go to step 2

2. *Recursive iteration.* At this point, each unpopulated cell has multiple possible values; each possible value must be attempted.
   Loop through each unpopulated cell in the grid
   a. If the cell has no possible values
      i. Puzzle does not have a solution; return FAILURE
   b. Else
      i. Populate the cell with the next possible value
      ii. Duplicate the whole grid
      iii. Recursively call this algorithm (starting at Step 1) with the duplicated grid
      iv. If the result is SUCCESS, the puzzle is solved; return SUCCESS. The returned grid is the solution.
      v. If the result is FAILURE, continue at step i,, above, to populate the next possible value

[1] An additional motivation was that I had already implemented this algorithm some years earlier, and have been waiting for an opportunity to add a GUI and some kind of digit recognition input mechanism.

1. If there are no more possible values, puzzle does not have a solution; return FAILURE

B. Application Architecture

The Sudoku application consists of the following major components:

- A GUI (implemented in Visual Studio 2015 VB.NET) to handle the presentation and user input
  - The application of both binary and Gaussian Bayes classifiers is also implemented in VB, using the training data generated from MATLAB
- The Sodoku solver algorithm (implemented in C++ as a DLL[2])

An image of the Sudoku GUI is as follows:

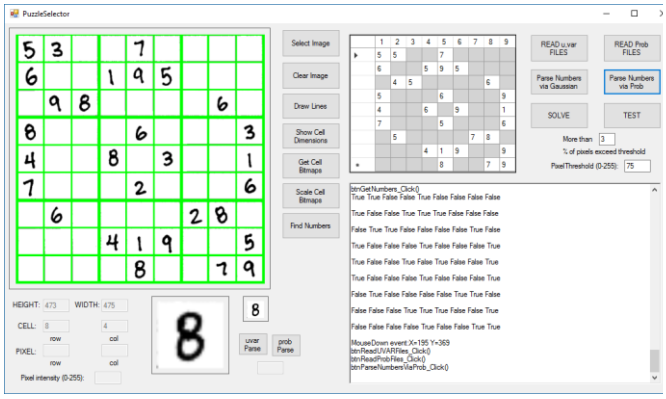


Figure 4. Sudoku GUI.

The Bayesian classifier data from MATLAB was imported to the application as follows:

- Binary Bayes: for each digit, a "pixel probability map" file was created in MATLAB in CSV format; each file consists of a 28x28 array of floating-point numbers representing the "ON" probability of each pixel
- Gaussian Bayes: for each digit, a "mean" CSV file, and a "variance" CSV file, was created in MATLAB; each file consists of a 28x28 array of floating-point numbers representing the mean or variance of each pixel

C. Application Image Parsing Algorithm

A significant task performed by the Sudoku application is to generate 28x28 pixel images from the populated puzzle cells of the input Sudoku image, in preparation for digit recognition. While not relevant to the classification task, this effort was significant enough to summarize:

- *Read in the image as a bitmap*. The image was assumed to be orthogonal, i.e. no deskewing was required. The image also required gridlines of sufficient contrast so that the cells could be discerned
- *Identify the coordinates of all the gridlines*. This was performed in order to determine the coordinates of each cell. A simple method of summing the pixel intensity values for each pixel row and column was performed; the rows and columns with maximum intensity sums were determined to be the gridlines.
  Note in the GUI image that the gridlines were visibly marked with green outlines.
- *Extract cell bitmaps.* From the gridline coordinates determined in the previous step, a bitmap was extracted for each cell.
- *Scale the bitmaps.* Each cell bitmap was scaled to a 28x28 size, to correspond to the classifier algorithm requirement.
- *Determine which cells contained numbers*. Trial-and-error determined that, if more than 3% of the pixels in a given 28x28 bitmap exceeded an intensity value of 75 (out of 255), then the cell was populated with a number to be classified.

D. Application Results

The application was only applied to three handwritten Sudoku puzzles, and the digit recognition results were only qualitatively considered:

- Binary Bayes algorithm appeared to perform better, identifying slightly more than half of the digits present in each puzzle
- Gaussian Bayes performed worse, identifying slightly less than half of the digits in each puzzle

As expected, the actual Sudoku solving algorithm (once all the incorrectly scanned and recognized digits were corrected manually) was successful in solving the puzzles.

V. CONCLUSIONS

A. Observations on Digit Classification

While successful on the MNIST database (~80% success rate), both Bayesian classifiers fell short when applied to the Sudoku puzzle scanning problem. This could be due to a number of reasons:

- *Small training set.* The classifiers were trained on a subset of MNIST. A larger training set, e.g. the full MNIST database, may improve results
- *Dissimilar test set.* The scanned Sudoku puzzles may not have been as similar to the MNIST training set. Considerations such as penstroke thickness, scan quality, image scaling, and noisy background intensity may significantly impact the results (as

[2] The C++ implementation was first ported to VB.NET, but the VB.NET version was found to be so slow as to be unusable for puzzles with greater than 15 empty cells. This is probably due to the recursive and memory-intensive nature of the algorithm not lending itself well to the "managed code" design (i.e. the Common Language Runtime approach of .NET) as well as the "garbage collection" scheme of memory management.

described earlier, the pixel equivalent of the "bag of words" approach of the Bayesian algorithm is sensitive to these scan / bitmap artifacts)

While there may be incremental improvements to the Bayesian approaches, a more sophisticated digit recognition scheme (e.g. a feature based approach using HOG, or a neural network scheme) may be more productive.

A future stretch goal might be to consider the ability to classify digits with electronic fonts (in addition to handwritten characters), so that this application could solve puzzles from images culled from the Internet at large.

B. Observations on Sudoku Application

The majority of the effort on this project was devoted to the Sudoku application, and especially the image processing aspects of importing the Sudoku puzzle images.

Quite a number of improvements could be considered for the future:

- A more modular design to better accommodate adding and selecting different image classification algorithms
- Automating populated-cell detection, perhaps with some type of adaptive filter approach
- Better pre-processing of scanned puzzles, e.g. automated contrast control, deskewing, denoising

REFERENCES

[1] D. Ciresan, U. Meier, J. Schmidhuber, "Multi-column Deep Neural Networks for Image Classification", Technical Report No. IDSIA-04-12, Dalle Molle Institute for Artificial Intelligence, February 2012

[2] K. Jarrett, K. Kavukcuoglu, M. Ranzato, Y. LeCun, "What is the Best Multi-Stage Architecture for Object Recognition?", Proc. Int'l Conf on Computer Vision, IEEE, 2009

[3] Y. LeCun, C. Cortes, C.J.C. Burges, "The MNIST Database of handwritten digits", http://yann.lecun.com/exdb/mnist/

[4] S. Gangaputra, Handwritten digit database derived from original MNIST database, http://cis.jhu.edu/~sachin/digit/digit.html

[5] M. Mahdian, E.S. Mahmoodian, "Sudoku Rectangle Completion", Electronic Notes in Discrete Mathematics 49 (2015), pp. 747-755

[6] T.K. Moon, J.H. Gunther, J.J. Kupin, "Sinkhorn Solves Sudoku", IEEE Transactions on Information Theory, Vol. 55, No. 4, April 2009

[7] J. Gunther, T. Moon, "Entropy Minimization for Solving Sudoku", IEEE Transactions on Signal Processing, Vol. 60, No. 1, January 2012