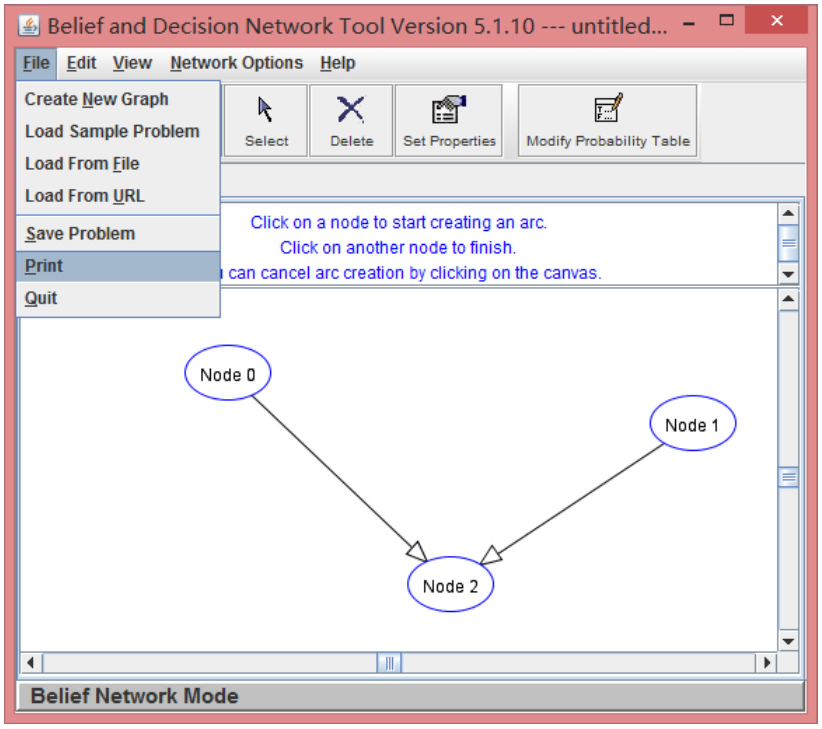
CS360 Project 3

Part 1: Bayesian Networks

From `http://aispace.org/bayes/` download the AIspace '`Belief and Decision Networks`' Java applet. You might need to add `http://www.aispace.org/` to the '`Exception Site List`' available in the '`Security`' tab of the '`Java Control Panel`' to get the applet to run. Read the documentation available at the download site and try it out. **Do this well before the deadline to ensure that it works on your computer.**



Click on a node to start creating an arc.
Click on another node to finish.
can cancel arc creation by clicking on the canvas.

Figure 1: AIspace "Belief and Decision Networks" Java Applet

Problem 1

Use the Java applet to create a Bayesian network that represents 5 time-slices of the umbrella DBN specified in Figure 15.13 (page 592) of the textbook using the DBN **unrolling** technique discussed on page 595. Note that this network should also include the initial distributions at time $t = 0$. Save your Bayesian network (using '`File`' $\rightarrow$ '`Save program`') and include a screenshot of your network in your answer. Use the Java applet to calculate the probability of $Rain_5$ (that is, the probability that it is raining at time $t = 5$), given the following sequence of umbrella observations: $Umbrella_1 = True$, $Umbrella_2 = True$, $Umbrella_3 = True$ $Umbrella_4 = False$, $Umbrella_5 = True$. Recall that observations begin at time $t = 1$, not time $t = 0$. Use the P(e) Query functionality of the Java applet to calculate the probability of this sequence of umbrella observations from $t = 1$ to $t = 5$. Include both of these solution probabilities in your answer to this problem. Calculating $P(Rain_5|Umbrella_1, ..., Umbrella_5)$ is known as **filtering**. Describe an example real-world problem that can be solved using filtering (with a suitable DBN).

Problem 2

The umbrella DBN you created (and unrolled) in Problem 1 is actually a **Hidden Markov Model** (or HMM, the simplest type of DBN) characterized by having a single observation variable ($Umbrella_t$) and a single state variable ($Rain_t$). However, DBNs can be much more complicated, having many observation and state variables at each time step. Assume that we wish to add another **state** variable $Humidity_t$ to this umbrella DBN that directly influences $Rain_{t+1}$ and $Humidity_{t+1}$ at each time step. Assume that $Humidity_t$ is conditionally independent of $Umbrella_t$ given $Rain_t$. At time step $t = 0$, assume that $Humidity_0$ and $Rain_0$ are independent. Modify your unrolled DBN in Problem 1 to accommodate this new state variable, $Humidity_t$, and the two specified (conditional) independencies. Explain in detail why your network satisfies the two specified (conditional) independencies. Save the resulting network and include a screenshot of it in your answer to this problem. See page 585 for an example DBN with two state variables (position and velocity, in this case).

Problem 3

Using the Java applet, develop a Bayesian network that can be used to predict whether Chase Bank should lend money to a customer. Random variables might include information about the income level of the customer, their existing debt, their credit score, etc, and anything else you might think would be relevant. This network should have 10 or more nodes and a non-trivial topology. Save your Bayesian network and include a screenshot of it in your answer. Create two interesting nontrivial scenarios where the values of some nodes are known. Show the predictions of your Bayesian network (whether to lend money to the customer) for these scenarios and explain why they make sense. Note that this question is asking for a regular Bayesian network, not a DBN.

Problem 4

Read through the remainder of this document carefully, and describe, in detail, what is being visualized by our naive Bayesian classifier in Figure 3. Explain what results in the "fuzziness" of the images of the digits and how this visualization helps to explain situations in which our classifier is likely to perform well or poorly. Provide two concrete means by which we might hope to increase the predictive accuracy of our classifier on this data set.

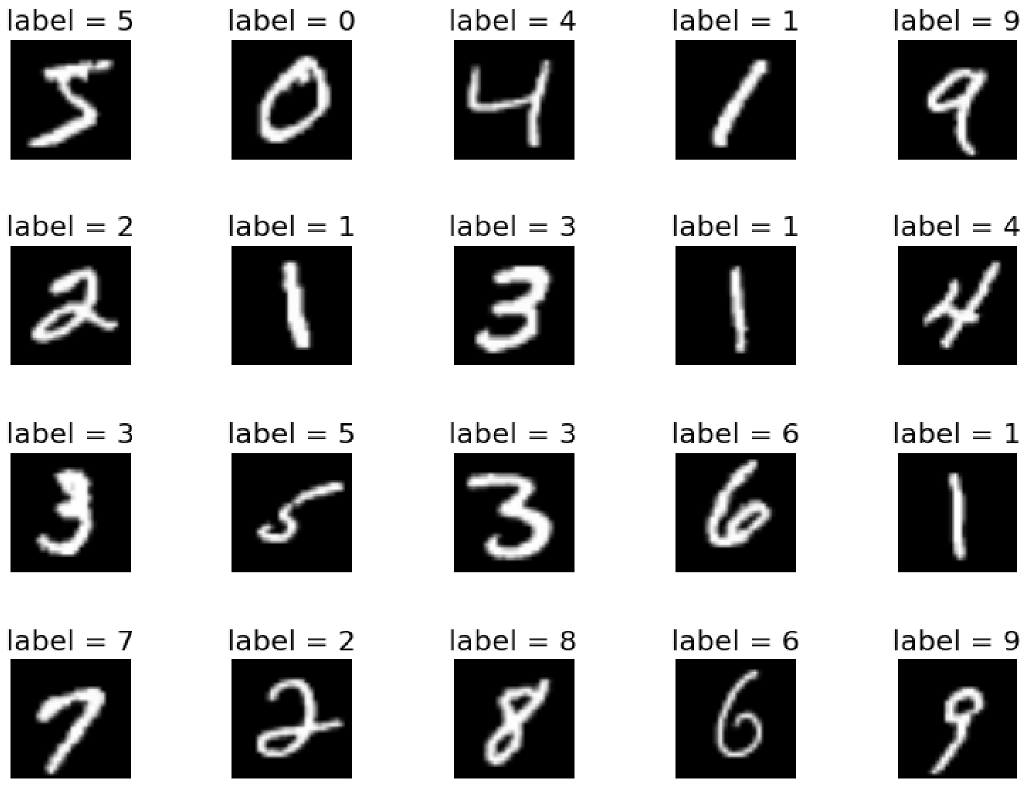


Figure 2: Example training instances from the MNIST database with labels.

Part 2: Naive Bayesian Learning

In this problem, you will implement a naive Bayesian classifier for classifying images of handwritten digits. Given an image representing a handwritten digit between 0 and 9, your goal is to output the digit represented by the image. You will train and test this algorithm using the **MNIST database**, one of the most famous machine learning databases[1]. It contains 60,000 training images and their associated **labels** (i.e., the correct digit represented by each image). It also contains 10,000 test images and their associated labels. You will train your naive Bayesian classifier on the 60,000 training images (the **training set**) and evaluate its performance on the 10,000 test images (the **test set**). In contrast to Project 2, there will be no validation set. Figure 2 shows a number of examples of MNIST training set images and their associated labels. Correctly identifying images of digits is vitally important in many applications, such as automatically reading the zip codes on pieces of US mail.

[1]http://yann.lecun.com/exdb/mnist/

Data

The MNIST database is provided for you in the folder mnist_data/, which consists of four files:

1. train-images-idx3-ubyte, which contains 60,000 28 × 28 grayscale training images, each representing a single handwritten digit.
2. train-labels-idx1-ubyte, which contains the associated 60,000 labels for the training images.
3. t10k-images-idx3-ubyte, which contains 10,000 28 × 28 grayscale test images, each representing a single handwritten digit.
4. t10k-labels-idx1-ubyte, which contains the associated 10,000 labels for the test images.

Files 1 and 2 are the training set. Files 3 and 4 are the test set. Each training and test instance in the MNIST database consists of a 28 × 28 grayscale image of a handwritten digit and an associated integer label indicating the digit that this image represents (0-9). Each of the $28 \times 28 = 784$ pixels of each of these images is represented by a single 8-bit color channel. Thus, the values each pixel can take on range from 0 (completely black) to 255 ($2^8 - 1$, completely white). To simplify the problem for this project, we are going to **binarize** this data set. Any pixel whose value is lower than 128 will be set to 0 (black) and any pixel whose value is 128 or greater will be set to 255 (white).

Example/skeleton code is provided in the src/ directory. The provided header files mnist_reader_common.hpp, mnist_reader_less.hpp, mnist_reader.hpp, and mnist_utils.hpp implement functions for easily parsing the MNIST database files. main.cpp provides example code for using these functions. You will only need to use two of these functions to solve this problem:

```
std::string MNIST_DATA_DIR = "../mnist_data";
//Read in the data set from the files
 mnist::MNIST_dataset<std::vector, std::vector<uint8_t>, uint8_t> dataset =
 mnist::read_dataset<std::vector, std::vector, uint8_t, uint8_t>(MNIST_DATA_DIR);
 //Binarize the data set (so that pixels have values of either 0 or 1)
 mnist::binarize_dataset(dataset);
```

The first two lines of the above code block (which you can use as-is) load the MNIST database, while the third line binarizes the training and test images, so that each pixel value in each image will be either a 0 (representing black) or 1 (representing white). **Note that white is represented by a 1 when we binarize, not 255!** You can extract the training images and labels as follows:

```
// get training images
std::vector<std::vector<unsigned char>> trainImages = dataset.training_images;
// get training labels
std::vector<unsigned char> trainLabels = dataset.training_labels;
```

The variable *trainImages* in the code block above is a vector of 60,000 vectors of unsigned chars, each representing a training image. For each $i \in \{0, ..., 59, 999\}$,

$trainImages[i]$ is a vector of 784 unsigned chars, each representing one of the 784 pixel values of training image i (**linearized in row major order**). The variable $trainLabels$ in the code block above is a vector of 60,0000 unsigned chars. For each $i \in \{0, ..., 59, 999\}$, $trainLabels[i]$ is an unsigned char (a single byte) indicating the true digit represented by $trainingImages[i]$. To use these values, they should first be converted to integers (or uint8_t) as follows:

```
//get pixel value j of training image i (0 for black, 1 for white)
int pixelValue = static_cast<int>(trainingImages[i][j])
//get the label (correct digit) of training image i
int label = static_cast<int>(trainingLabels[i]);
```

Note that the variable $label$ in the code block above will be an integer in the range [0 - 9], while the variable $pixelValue$ will be either 0 (if the pixel is black) or 1 (if the pixel is white), because we have binarized the data set. The images and the labels in the test set can be extracted analogously and also should be converted to ints or uint8_ts before use:

```
//get the test images
std::vector<std::vector<unsigned char>> testImages = dataset.test_images;
//get the test labels
std::vector<unsigned char> testLabels = dataset.test_labels;
```

Algorithm Framework

We will build a naive Bayesian model for this problem by **considering each pixel individually as a binary feature** that takes on the value 0 or false when the pixel is black and 1 or true when the pixel is white. In other words, feature F_j is a random variable representing the value of pixel j, for all $j \in \{0, ..., 783\}$. The class variable, C, will have 10 possible values, one for each possible digit 0-9. From our training set, we will learn the **prior probability** of each class $c \in C$, $P(c)$. We will also learn the **conditional probabilities** of each feature, given its class. That is, for all $c \in C$, we will learn $P(F_j|c)$, where $P(F_j|c)$ is a distribution over the possible values that F_j can take on (0 or 1, in this case) given that the class (digit) is c. Once we have **learned these probabilities from the training set**, we will **evaluate our model on the test images**. Let T_i represent test image i and let $t_{i,j}$ denote the value (0 or 1) of pixel j in test image i. For each test image i, we will use our model to compute $P(C|F_0 = t_{i,0}, ..., F_{783} = t_{i,783})$, the **posterior probability distribution** over the possible values of the class variable (in this case, the possible digits that the image might represent) given the pixel values of the test image. We will predict that test image T_i represents the digit with the highest posterior probability, that is, $\hat{c}_i = argmax_c P(c|F_0 = t_{i,0}, ..., F_{783} = t_{i,783})$. Let c_i^* represent the correct class (digit) of test image T_i. If $c_i^* = \hat{c}_i$, our model has successfully classified image T_i. The **accuracy** of our model on the test set is:

$$\frac{\sum_{i=0}^{9,999} \mathbb{1}(c_i^* = \hat{c}_i)}{10,000}$$

Note that $\mathbb{1}$ is the indicator function, which returns 1 when $c_i^* = \hat{c}_i$ and 0 otherwise. So the accuracy of our model is simply the ratio of successful classifications to total test images.

Since the intermediate probabilities can be quite small, we use the summation of log probabilities instead of the multiplication of probabilities in order to avoid underflow errors. In other words, instead of calculating $\prod_i P_i$, you will calculate $\sum_i \log P_i$ since $\prod_i P_i = \exp\{\sum_i \log P_i\}$. This implies that you will store $\log P_i$ instead of P_i for all of the following probabilities.

To learn the naive Bayesian classifier, you should do all the following steps **only for the training set**.

- Determine the prior probabilities for each class $c \in \{0,1,2,3,4,5,6,7,8,9\}$:

$$P(C = c) = \frac{\#\{\text{images of digit } c\}}{\#\{\text{images}\}}$$

  Make sure that: $\sum_{c=0}^{9} P(c) = 1$.

- For each pixel F_j for $j \in \{0, ..., 783\}$ and for each class $c \in \{0,1,2,3,4,5,6,7,8,9\}$, determine $P(F_j = 1|C = c)$, the probability that pixel F_j is white given that it is an image of digit c:

$$P(F_j = 1|C = c) = \frac{\#\{\text{images of digit } c \text{ where pixel } F_j \text{ is white}\}}{\#\{\text{images of digit } c\}}$$

  Note that, if $P(F_j = 1|C = c) = 0$ (if pixel F_j has never been white in any image of digit c), then, for any image in the test set of digit c where pixel F_j is white, our classifier will predict that the probability that the image is of digit c is 0. To address this issue, we can pretend that we have observed every outcome once more than we actually did (called Laplace smoothing):

$$P_L(F_j = 1|C = c) = \frac{\#\{\text{images of digit } c \text{ where pixel } F_j \text{ is white}\} + 1}{\#\{\text{images of digit } c\} + 2}$$

  Note that $P(F_j = 0|C = c) = 1 - P(F_j = 1|C = c)$ since each feature is binary.

Once you have trained your naive Bayesian classifier using the training set, you need to evaluate its performance on the **test set**. For a test image T_i and pixel F_j, let $t_{i,j}$ denote the value of pixel j in test image T_i. Note that $t_{i,j}$ will be either 0 or 1 because we have binarized the test set as well as the training set.

For each test image T_i, compute the probability that it belongs to each class $c \in \{0,1,2,3,4,5,6,7,8,9\}$:

$$P(C = c|F_0 = t_{i,0}, ..., F_{783} = t_{i,783}) = \frac{P(F_0 = t_{i,0}, ..., F_{783} = t_{i,783}|C = c) \cdot P(C = c)}{P(F_0 = t_{i,0}, ..., F_{783} = t_{i,783})}$$

$$= \frac{(\prod_{j\in\{0,...,783\}} P_L(F_j = t_{i,j}|C = c)) \cdot P(C = c)}{P(F_0 = t_{i,0}, ..., F_{783} = t_{i,783})}$$

The second step is due to the naive Bayesian assumption.

The test image T_i is classified as belonging to the class c with the highest posterior probability $\hat{c}_i = argmax_c P(C = c|F_0 = t_{i,0}, ..., F_{783} = t_{i,783})$. The term $P(F_0 = t_{i,0}, ..., F_{783} = t_{i,783})$ can be dropped from the calculations without changing the resulting classification. Since we are using Laplace smoothing, so we are using $P_L(F_j = t_{i,j}|C = c)$ rather than $P(F_j = t_{i,j}|C = c)$. Furthermore, remember that, in order to avoid underflows, instead of multiplying probabilities, we are adding their logarithms. Therefore, we classify image T_i as belonging to the class $\hat{c}_i$ that maximizes the following expression:

$$argmax_c(\sum_{j\in\{0,...,783\}} \log P_L(F_j = t_{i,j}|C = c)) + \log P(C = c)$$

The reference implementation performed the maximization using the std::max_element() function defined in the $< algorithm >$ header file, if you wish to be fully consistent with the reference implementation in case there are ties (which should be exceedingly rare).

Evaluation

Working with images affords us unique opportunities to visualize the parameters we have learned for our naive Bayesian classifier. For the first part of this evaluation, you will use the provided Bitmap class (also in the src/ directory and defined in bitmap.cpp and bitmap.hpp) to output images representing the conditional probabilities that each feature takes on value 1 given class $C = c$ (i.e., $P(F_j = 1|C = c)$). Use the following code to visualize the naive Bayesian model you have learned:

```
int numLabels = 10;
int numFeatures = 784;
for (int c=0; c<numLabels; c++) {
     std::vector<unsigned char> classFs(numFeatures);
     for (int f=0; f<numFeatures; f++) {
      //TODO: get probability of pixel f being white given class c
      p = probability that pixel f is white given digit c
         uint8_t v = 255*p;
         classFs[f] = (unsigned char)v;
     }
     std::stringstream ss;
     ss << "../output/digit" <<c<<".bmp";
     Bitmap::writeBitmap(classFs, 28, 28, ss.str(), false);
}
```

To finish the code block above you must assign to the variable p the probability that the pixel at location f is white given that the class is c after learning those probabilities from the training set. The remainder of the code can be executed as is. This will write 10 bitmap files to the output directory digit$i$.bmp for $i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. The results should look like Figure 3.

Your program should output two additional files:

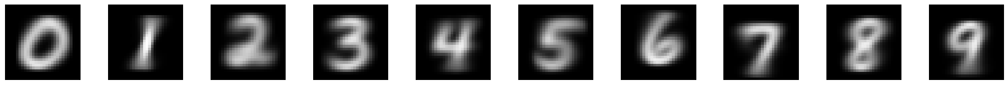


Figure 3: Visualization of the class conditional densities of each pixel for each class $c \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

- **network.txt** This file should list the values of all conditional probabilities parameterizing the network. The first 784 lines should be $P(F_j = 1|C = 0)$. The next 784 lines should be $P(F_j = 1|C = 1)$. The final 10 lines should be the prior probabilities of each class, in order from 0 to 9.
- **classification-summary.txt** This file should contain a 10×10 matrix of integers. The integer in row r and column c should be the number of images in the test set of digit r which our model predicted was an image of digit c. The final line of this file should be the final accuracy of your naive Bayesian model (number correctly classified/10,000) on the test set of 10,000 images. For comparison, the reference implementation achieves a test accuracy of **84.43%**.

Finally, we note that using a naive Bayesian model to solve this problem does not represent the current state of the art. For instance, a relatively simple convolutional neural network can achieve an accuracy of about 99.2%. A committee of 5 convolutional neural networks can achieve an accuracy of approximately 99.8%. The Python TensorFlow library has a very informative tutorial on building such a model for the same digit recognition task at https://www.tensorflow.org/get_started/mnist/pros. Though it is not part of this project, we encourage you to think about why such models might perform better than the naive Bayesian model we have built in this project.

Submission

You need to submit your solution of Part 1 as a PDF file named 'Part1.pdf'. It should contain the screenshots of the three different Bayesian networks you have developed for Part 1. You also need to submit three xml files named 'Part1a.xml', 'Part1b.xml' and 'Part1c.xml', one for each Bayesian network (using `'File'` $\rightarrow$ `'Save program'` generates these xml files).

You need to submit your solution of Part 2 as a zip file that contains your source code and a makefile (our sample code includes a makefile, but you might need to modify it as you add new cpp files) that produces an executable called proj3. When we run your executable, it should process the training and test sets and output the bitmap files and the two text files that we mention in the Evaluation section of Part 2.

You should submit all your files through the blackboard system by Monday, Nov. 20th, 11:59pm.