Richard Gibson
#0235677

CSC 421 Assignment #4

Apr. 4, 2006

Probability Theory

Suppose that we are given cipher texts which encode messages in such a way that each letter of the alphabet is mapped to a sequence of 3 letters so that the frequency counts of each letter in the cipher text are approximately equal. Here is how the text can be broken:

Since letter frequencies are approximately equal in the cipher text, directly counting the letter frequencies will do us no good. However, it will do us some good to count the number of times a letter occurs given the previous two letters since sequences of 3 letters are used for the encryption. Assuming that there are n letters in our alphabet, we can do this by setting up an n-by-n-by-n table and making a tally of when a letter occurs after the previous two letters. For instance, if we find an "a" in our cipher text in which the 2 letters immediately preceding it are "bc", then we would tally this in the (b,c,a)-entry of our table. In other words, we make the Markov assumption that a letter only depends on it's previous two letters. After completely examining all cipher texts, the entry with the highest tally is then the most common sequence. Assuming that we know the frequencies of the "real" letters in the language, we could make a good guess that the most frequent letter in the alphabet is mapped to the most frequent sequence in the table, the second most frequent letter to the second most frequent sequence, and so on. For instance, if the (r,s,t)-entry of the table has the highest tally, then we would guess that the most frequent letter in our language is mapped to the 3 letter sequence "rst". With a bit of experimenting, the code can be easily solved.

Suppose now that we are given cipher texts which encode messages as follows: For each letter of the alphabet that needs to be encoded, a 6-sided unbiased die is rolled 6 times and the sum of the numbers rolled is noted. Then a notebook is consulted which contains a table with substitution entries for each possible value of the sum. This process can be modeled as a Hidden Markov Model as follows:

- Let L_t be the tth letter of the real message. These are the state variables of the HMM which are hidden from us.
- Let E_t be the tth letter of the cipher text which encodes the real message. These are the evidence variables of the HMM which are directly observable.
- I will make the first-order Markov assumption that any given letter in this language only depends on the previous letter; i.e. $\mathbf{P}(L_t \mid L_{0:t-1}) = \mathbf{P}(L_t \mid L_{t-1})$.
- The transition model is obtained from our knowledge of the real language of how frequently certain letters follow other letters, and give us $\mathbf{P}(L_t \mid L_{t-1})$. Therefore the transition model will be an n-by-n table (where n is the number of letters in the alphabet) where, for instance, if we know that the letter "t" is followed by the letter "h" 5% of the time in our language, then the $\mathbf{P}(L_t = h \mid L_{t-1} = t)$ entry of our table will be 0.05.

- The sensor model is obtained from the entries in the notebook used to encode the messages (assuming that we have access to this notebook) and give us the values of $\mathbf{P}(E_t \mid L_t)$. This is done by summing the probabilities of all the rolls for L_t that use E_t to encode it. For instance, if we are encoding the letter "a" and if (6,b) and (36,b) are the only 2 entries in the table for "a" in which it is encoded by "b" (i.e. all other rolls dictate "a" to be encoded by a letter other than "b"), then $\mathbf{P}(E_t = b \mid L_t = a) = 1/6^6 + 1/6^6 = 2/6^6$ because the probability of rolling a sum of 6 (and similarly 36) is $1/6^6$.
- A reasonable prior probability distribution $\mathbf{P}(L_0)$ would be to simply use the frequencies of each real letter appearing in the language (which we assume we have sufficient knowledge of).

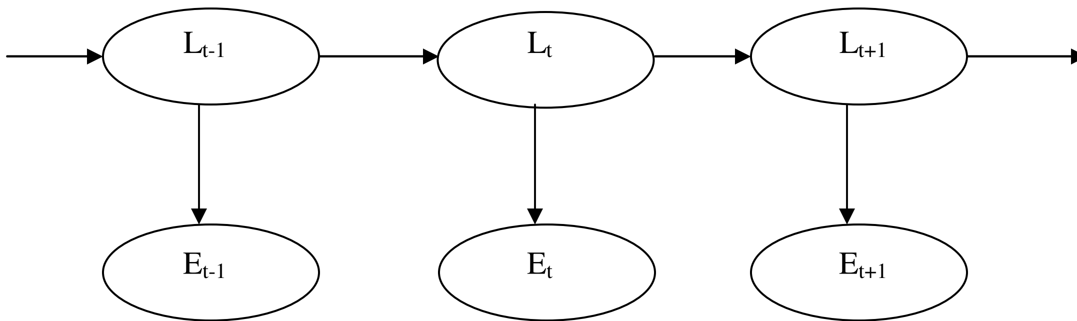


Figure 1: Baysian network topology of our Hidden Markov Model

Now to break the code, we want to perform the inference task of finding the most likely sequence of the letter variables $L_1, L_2, L_3, \ldots, L_k$ that produced the sequence of encoded letters $e_1, e_2, \ldots, e_k$. This can be done using the Viterbi algorithm as follows:

- View the possible state sequences for L_t as paths through a graph.
- For each t, calculate the message $m_{1:t}$ which gives the probability of the best sequence reaching each state at time t. Also keep track of each state's most likely predecessor.

The probability of the best sequence $m_{1:t}$ can be computed recursively as follows:

$$m_{1:t+1} = \max_{x1, \ldots, xt} \mathbf{P}(x_1, \ldots, x_t X_{t+1} \mid e_{1:t+1})$$
$$= \alpha\, \mathbf{P}(e_{t+1} \mid X_{t+1}) \max_{xt} \{ \mathbf{P}(X_{t+1} \mid x_t) \max_{x1, \ldots, xt-1} \mathbf{P}(x_1, \ldots, x_{t-1}, x_t \mid e_{1:t}) \}$$

In other words, just select the most likely letter that comes next given the evidence and the most likely sequence up to this point. To make this more concrete, here is an example of how this is done. Suppose that our language contains only 2 letters "a" and "b" and that we know that the letter "a" is followed by another "a" 70% of the time and that the letter "b" is followed by another "b" 45% of the time. Suppose also that there is a 40% chance that the letter "a" is encoded by the letter "a" and an 80% chance that the letter "b" is encoded by the letter "b" (governed by the likelihoods of the corresponding

rolls in the notebook). Assume that the frequencies of each letter in the language are approximately equal. Then we have the following transition and sensory models:

Table 1: Transition model

L_{t-1}	$\mathbf{P}(L_t = a \mid L_{t-1})$	$\mathbf{P}(L_t = b \mid L_{t-1})$
a	0.7	0.3
b	0.55	0.45

Table 2: Sensor model

L_t	$\mathbf{P}(E_t = a \mid L_t)$	$\mathbf{P}(E_t = b \mid L_t)$
a	0.4	0.6
b	0.2	0.8

Suppose we want to try to decode the message "aba". Now given that the frequencies of each letter in the language are approximately equal, we have $\mathbf{P}(L_0) = \langle \mathbf{P}(L_0 = a), \mathbf{P}(L_0 = b) \rangle = \langle 0.5, 0.5 \rangle$. Here are the computations:

$$
\begin{aligned}
m_{1:1} &= \mathbf{P}(L_1 \mid E_1 = a) \\
&= \alpha\, \mathbf{P}(E_1 = a \mid L_1)\, (\, \mathbf{P}(L_1 \mid L_0 = a)\mathbf{P}(L_0 = a) + \mathbf{P}(L_1 \mid L_0 = b)\mathbf{P}(L_0 = b)\,) \\
&= \alpha \langle 0.4,0.2 \rangle (\langle 0.7,0.3 \rangle 0.5 + \langle 0.55,0.45 \rangle 0.5) = \alpha \langle 0.25,0.075 \rangle \\
&= \langle 0.7692,0.2308 \rangle.
\end{aligned}
$$

$$
\begin{aligned}
m_{1:2} &= \max_{\gamma} \mathbf{P}(L_1 = \gamma, L_2 \mid E_1 = a, E_2 = b) \\
&= \alpha\, \mathbf{P}(E_2 = b \mid L_2) \max_{\gamma} \{ \mathbf{P}(L_2 \mid L_1 = \gamma)\mathbf{P}(L_1 = \gamma \mid E_1 = a) \} \\
&= \alpha \langle 0.6,0.8 \rangle \max \{ \langle 0.7,0.3 \rangle 0.7692, \langle 0.55,0.45 \rangle 0.2308 \} \\
&= \alpha \langle 0.6,0.8 \rangle \max \{ \langle 0.5384,0.2308 \rangle, \langle 0.1269,0.1039 \rangle \} \\
&= \alpha \langle 0.3230, 0.1846 \rangle \\
&= \langle 0.6363,0.3637 \rangle \text{ (most likely predecessors for both states is } L_1 = a\text{).}
\end{aligned}
$$

$$
\begin{aligned}
m_{1:3} &= \max_{\gamma_1,\gamma_2} \mathbf{P}(L_1 = \gamma_1, L_2 = \gamma_2, L_3 \mid E_1 = a, E_2 = b, E_3 = a) \\
&= \alpha\, \mathbf{P}(E_3 = a \mid L_3) \max_{\gamma_2} \{\mathbf{P}(L_3 \mid L_2 = \gamma_2) \max_{\gamma_1} \mathbf{P}(L_1 = \gamma_1, L_2 = \gamma_2 \mid E_1 = a, E_2 = b)\} \\
&= \alpha \langle 0.4,0.2 \rangle \max \{ \langle 0.7,0.3 \rangle 0.6363, \langle 0.55,0.45 \rangle 0.3637 \} \\
&= \alpha \langle 0.4,0.2 \rangle \max \{ \langle 0.4454,0.1909 \rangle, \langle 0.2000,0.1637 \rangle \} \\
&= \alpha \langle 0.1782,0.0382 \rangle \\
&= \langle 0.8235,0.1765 \rangle \text{ (most likely predecessors for both states is } L_2 = a\text{).}
\end{aligned}
$$

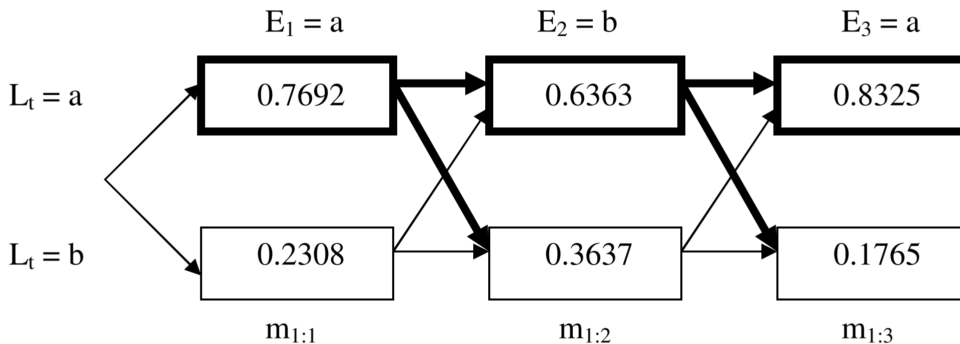


Figure 2: Most likely sequence of our two letter alphabet example

Therefore, following the bold arrows backwards from $L_3 = a$, we see that the most likely sequence is "aaa" and is therefore the best answer for decoding the message.

<u>Learning and Decision Trees</u>

I have chosen to base my decision tree on whether a board game will be fun or not given a set of attributes describing the game. Here is a set of training data:

Suggested Age Group	Average Time to Play (in hours)	Maximum Number of Players	Type of Game	Fun
Adults	1	6	Carcassonne	+
Kids	<1	4	Other	-
Adults	3	6	Other	-
Kids	2	2	Other	+
Adults	2	6	Settlers	+
Adults	3	6	Settlers	+
Kids	1	4	Other	-
Adults	2	6	Carcassonne	+
Adults	1	2	Carcassonne	+
Adults	>3	6	Settlers	-
Adults	>3	4	Other	-
Adults	2	2	Other	-
Kids	<1	6	Other	-
Adults	2	4	Other	-
Adults	2	6	Carcassonne	+
Adults	2	6	Other	+
Kids	2	4	Other	-
Adults	2	4	Settlers	+
Adults	>3	4	Settlers	-
Adults	2	6	Carcassonne	+

Here is the full trivial decision tree based on the training set above:

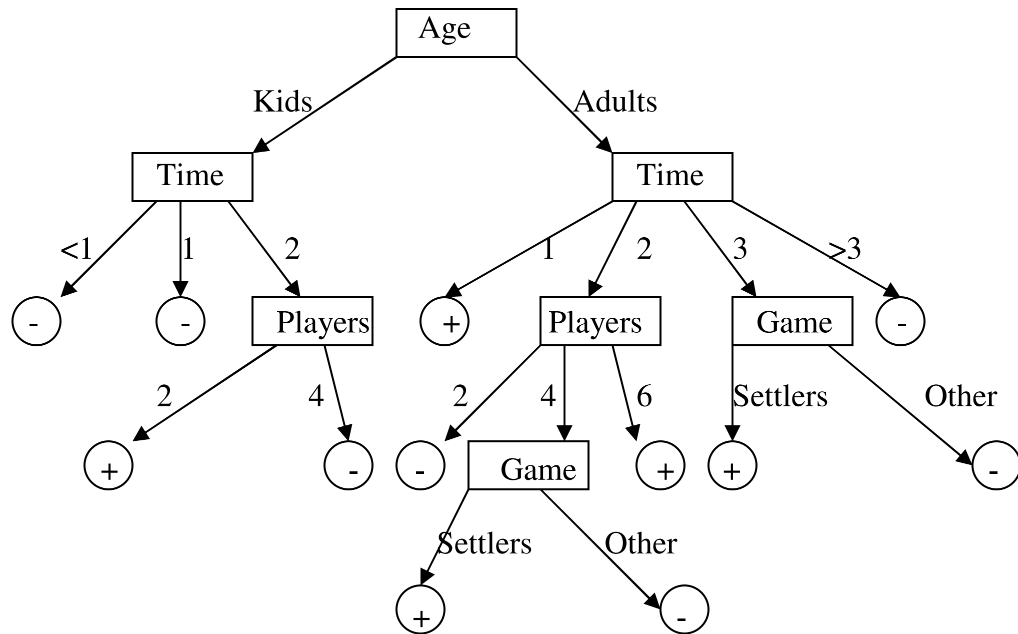


Figure 3: Default decision tree based on the training data

Now using the information-gain heuristic on the training data, we can learn a simpler decision tree as follows:

First, we need to find the attribute that minimizes the remainder of information needed to classify a new example. For each attribute A, we can separate the training data in sets E_i according to their values for A. Then we have

$$\text{Remainder}(A) = \sum_i (p_i + n_i)/(p + n)\ I(p_i/(p_i + n_i),\ n_i/(p_i + n_i))$$

where p_i and n_i are the number of positive and negative examples in E_i respectively, p and n are the total number of positive and negative examples in the entire training data, and

$$I(a, b) = -a \log_2 a - b \log_2 b$$

Here are the computations:

$$\begin{aligned}\text{Remainder (Age)} &= 5/20\ I(1/5, 4/5) + 15/20\ I(9/15, 6/15)\\ &= 0.25(-0.2 \log_2 0.2 - 0.8 \log_2 0.8) + 0.75(-9/15 \log_2 9/15 - 6/15 \log_2 6/15)\\ &\approx 0.9087\end{aligned}$$

$$\begin{aligned}\text{Remainder (Time)} &= 2/20\ I(0, 1) + 3/20\ I(2/3, 1/3) + 10/20\ I(7/10, 3/10)\\ &\quad + 2/20\ I(1/2, 1/2) + 3/20\ I(0, 1)\\ &= 0 + 0.15(-2/3 \log_2 2/3 - 1/3 \log_2 1/3)\\ &\quad + 0.5(-0.7 \log_2 0.7 - 0.3 \log_2 0.3) + 0.1(-\log_2 0.5) + 0\\ &\approx 0.6784\end{aligned}$$

$$\text{Remainder (Players)} = 3/20\ I(2/3, 1/3) + 7/20\ I(1/7, 6/7) + 10/20\ I(7/10, 3/10) \approx 0.7855$$

$$\text{Remainder (Type)} = 5/20\ I(1,0) + 5/20\ I(3/5, 2/5) + 10/20\ I(2/10, 8/10) \approx 0.6037$$

Since the attribute "Type" has the smallest remainder, it will become the root of our new decision tree. Then for each value v that type takes on (Carcassonne, Settlers, and Other), we repeat the process using only the examples of the training data where Type = v and find the best remaining attribute. We continue in this fashion until the whole tree is complete.

The training data was written as a Weka attribute file and then evaluated using the ID3 classifier in Weka explorer to find a learned decision tree:

In file "games.arff":

```
@relation Board_Games

@attribute age {adults, kids}
@attribute time {<1, 1, 2, 3, >3}
@attribute players {2, 4, 6}
@attribute type {carcassonne, settlers, other}
@attribute fun {yes, no}

@data
adults,1,6,carcassonne,yes
kids,<1,4,other,no
adults,3,6,other,no
kids,2,2,other,yes
adults,2,6,settlers,yes
adults,3,6,settlers,yes
kids,1,4,other,no
adults,2,6,carcassonne,yes
adults,1,2,carcassonne,yes
adults,>3,6,settlers,no
adults,>3,4,other,no
adults,2,2,other,no
kids,<1,6,other,no
adults,2,4,other,no
adults,2,6,carcassonne,yes
adults,2,6,other,yes
kids,2,4,other,no
adults,2,4,settlers,yes
adults,>3,4,settlers,no
adults,2,6,carcassonne,yes
```

Weka explorer output using ID3 classifier:

```
=== Run information ===

Scheme:       weka.classifiers.trees.Id3
Relation:     Board_Games
Instances:    20
Attributes:   5
              age
              time
              players
              type
              fun
Test mode:    evaluate on training data

=== Classifier model (full training set) ===

Id3


type = carcassonne: yes
type = settlers
|  time = <1: null
|  time = 1: null
|  time = 2: yes
|  time = 3: yes
|  time = >3: no
type = other
|  players = 2
|  |  age = adults: no
|  |  age = kids: yes
|  players = 4: no
|  players = 6
|  |  time = <1: no
|  |  time = 1: null
|  |  time = 2: yes
|  |  time = 3: no
|  |  time = >3: null

Time taken to build model: 0 seconds

=== Evaluation on training set ===
=== Summary ===

Correctly Classified Instances          20               100      %
Incorrectly Classified Instances         0                 0      %
Kappa statistic                          1
```

Mean absolute error 0
Root mean squared error 0
Relative absolute error 0 %
Root relative squared error 0 %
Total Number of Instances 20

=== Detailed Accuracy By Class ===

TP Rate	FP Rate	Precision	Recall	F-Measure	Class
1	0	1	1	1	yes
1	0	1	1	1	no

=== Confusion Matrix ===

```
  a  b   <-- classified as
 10  0 |  a = yes
  0 10 |  b = no
```

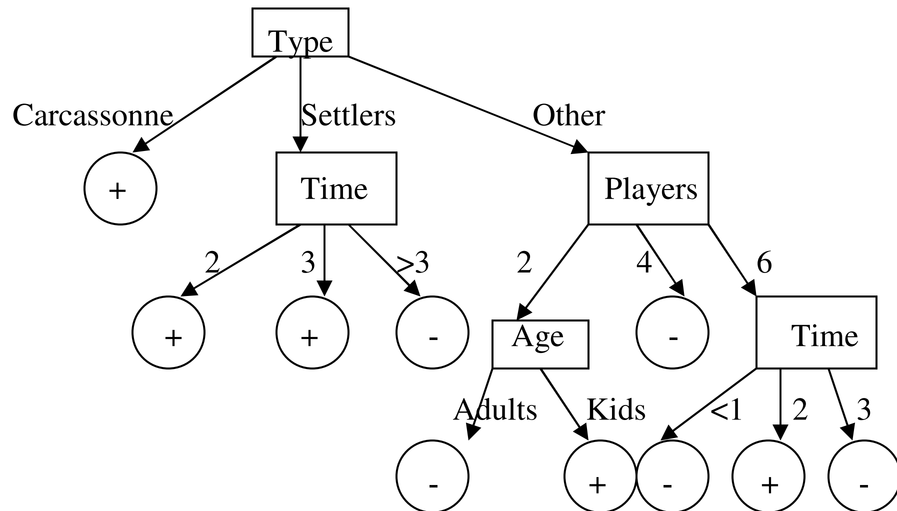


Figure 4: Simplified decision tree found by Weka

Statistical Learning

Here we are concerned with the exponential probability distribution, $p(x) = \Theta e^{-\Theta x}$. Here is a plot of this distribution on $0 < x < 100$ with $\Theta = 0.1$:

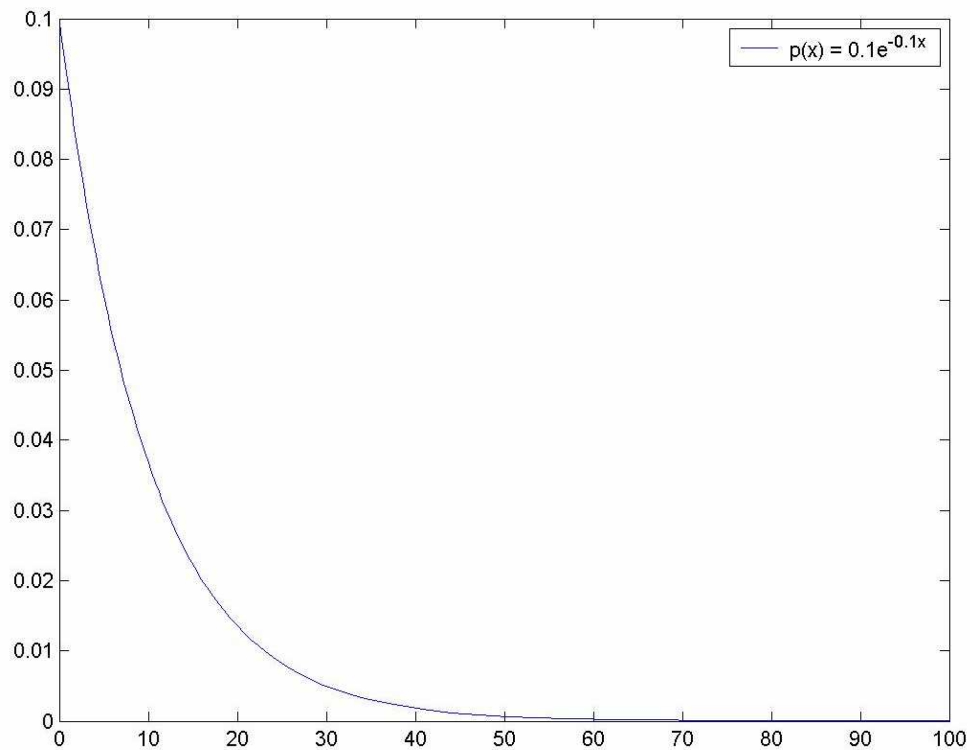


Figure 5: Plot of the Exponential Distribution with $\Theta = 0.1$

Here is some code that generates random samples of the exponential distribution for a given Θ (provided as an argument on the command line):

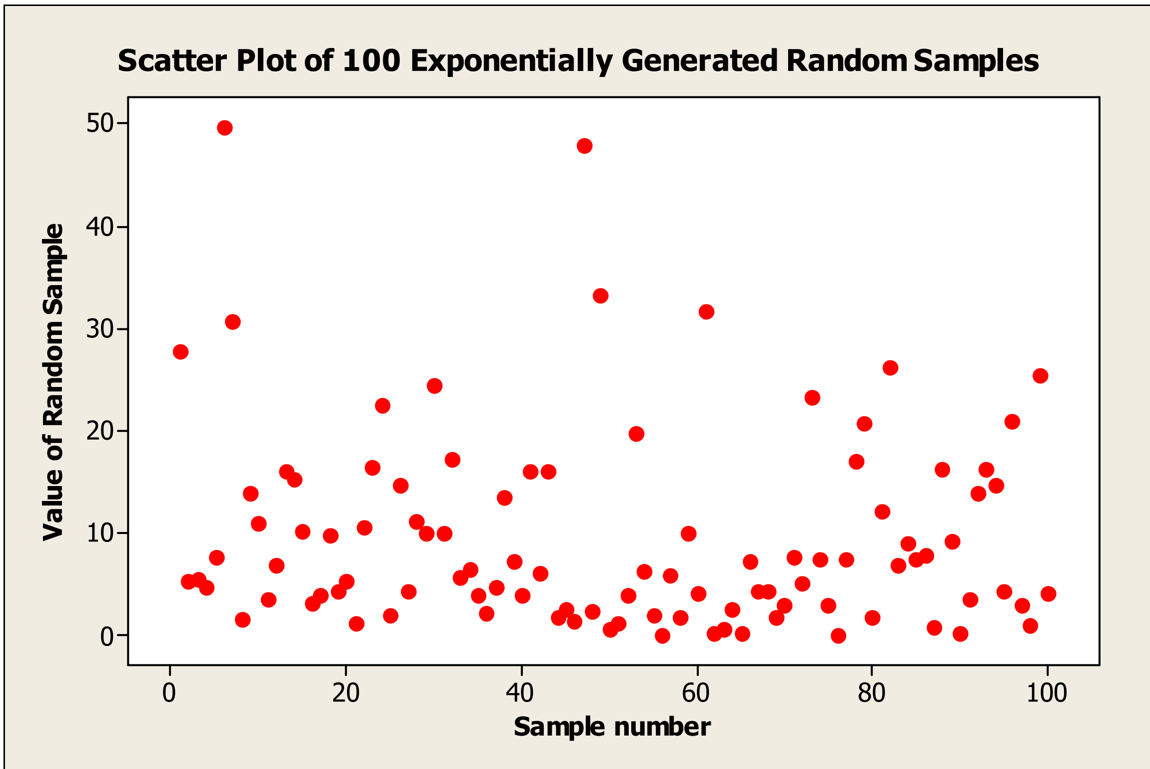
```
import java.util.*;
import java.io.*;

public class Exponential
{
	// randExp(theta, n) returns n random samples of the
	// corresponding exponential distribution (given theta)
	public static double[] randExp(double theta, int n)
	{
		double[] res = new double[n];
		for (int i = 0; i < n; i++)
		{
			double random = Math.random();
			// x = -ln(random)/theta
			res[i] = -Math.log(random)/theta;
		}
		return res;
	}

	// args[0] = theta, args[1] = n
	// Outputs all n random samples to file "exp.txt"
	// Outputs mean and standard deviation of samples to
	// console.
	public static void main (String[] args) throws Exception
	{
		FileOutputStream out = new FileOutputStream("exp.txt");
		PrintStream p = new PrintStream (out);
		double theta = Double.parseDouble(args[0]);
```

```
            int n = Integer.parseInt(args[1]);

            double[] res = randExp (theta, n);
            for (int i = 0; i < res.length; i++)
                   p.println (res[i]);

            // mean = (res[0] + ... + res[n-1])/n
            double mean = 0;
            for (int i = 0; i < res.length; i++)
                   mean += res[i];
            mean = mean/n;

            // variance = (res[0]^2 + ... + res[n-1]^2)/n - mean^2
            // sd = sqrt(variance)
            double sd = 0;
            for (int i = 0; i < res.length; i++)
                   sd += res[i]*res[i];
            sd = Math.sqrt ((sd/n) - (mean*mean));

            System.out.println ("n = " + n
                                              + "\nmean = " + mean
                                              + "\nstandard deviation = "
                                              + sd);
      }
}
```

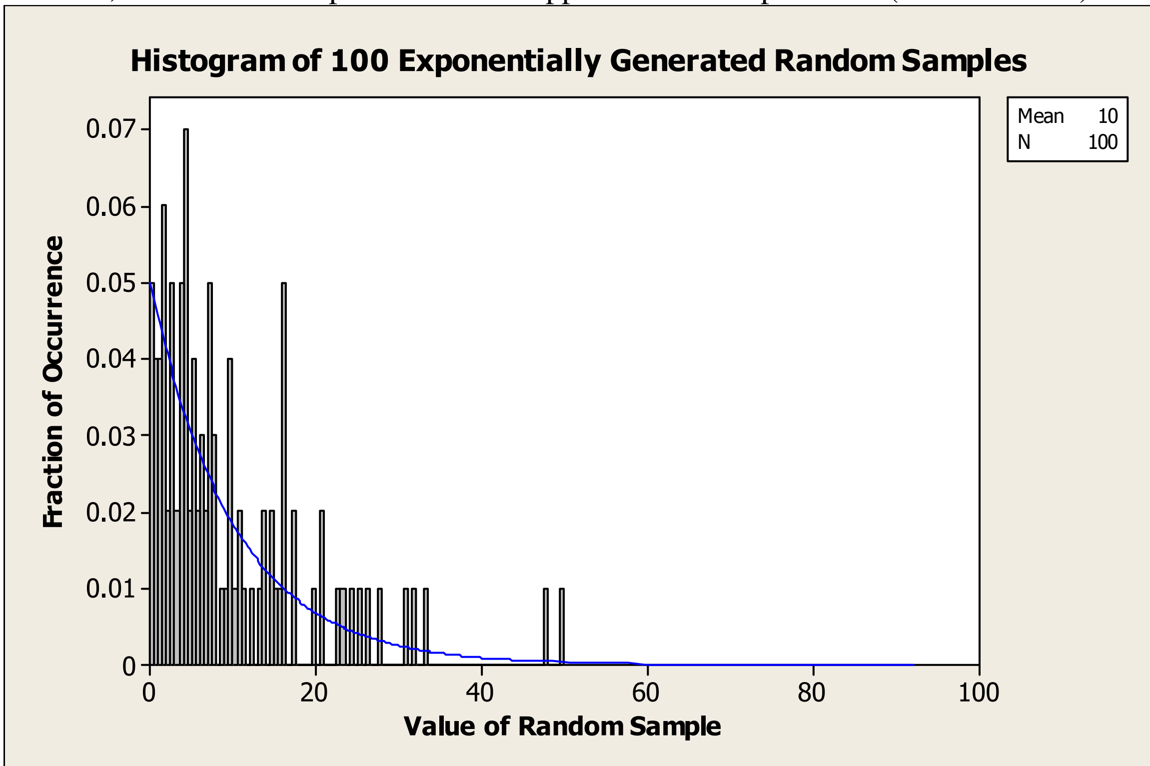
The program takes in two arguments, Θ and n, and produces n random samples of the given exponential distribution and prints them to a file “exp.txt”. The mean and standard deviation are then printed to the screen. Here is a test run:

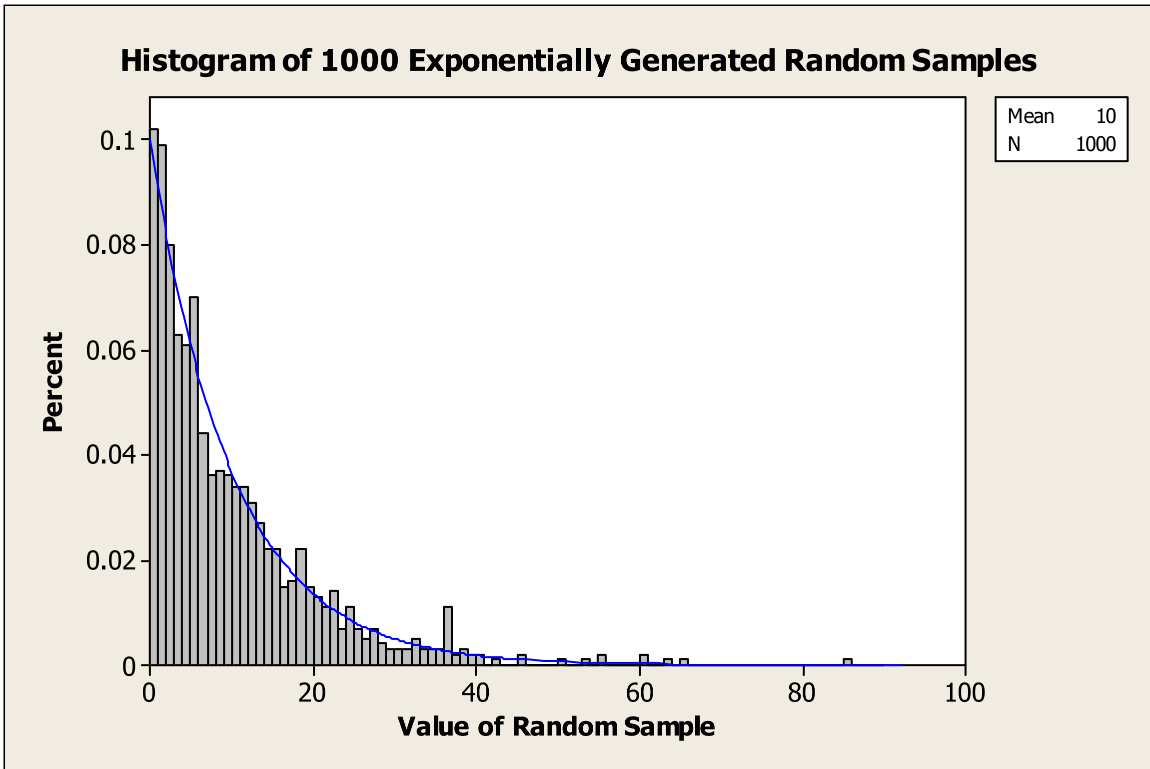
Command Line:
A:\csc421>java -cp . Exponential 0.1 100
n = 100
mean = 9.621445996206914
standard deviation = 9.624689652210852

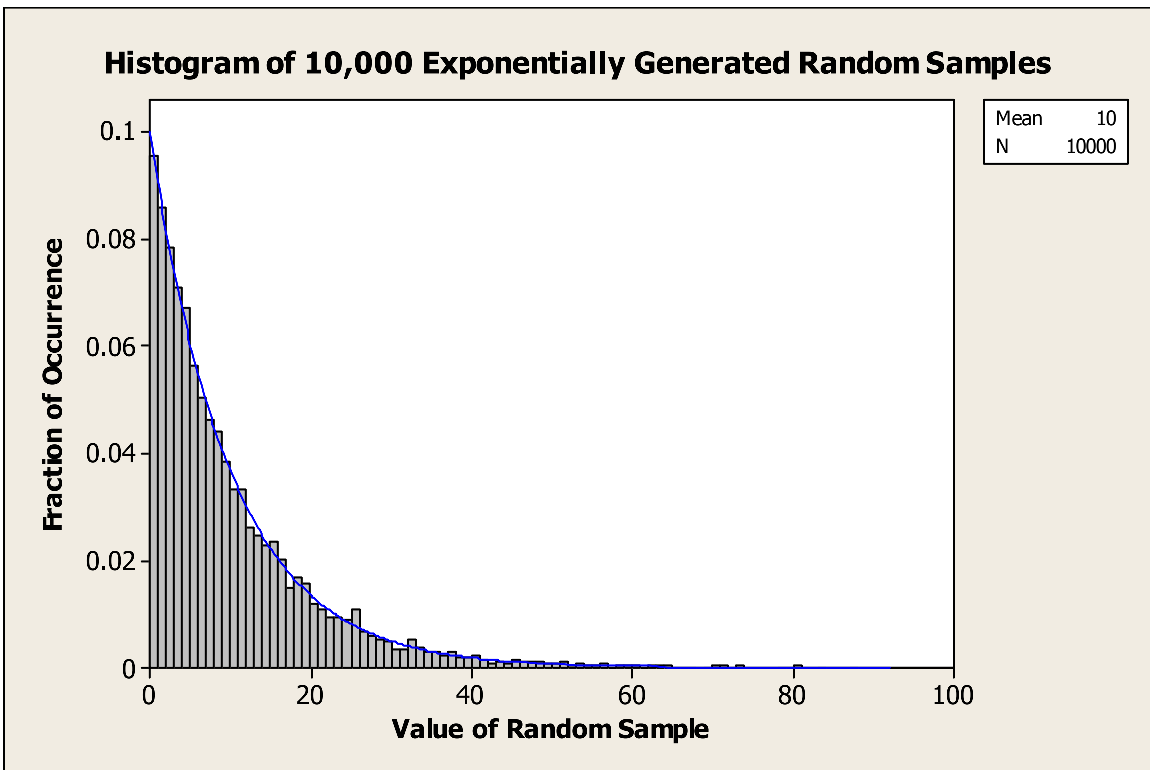
Note that the mean and the standard deviation are approximately equal and are close to 1/ Θ = 10. Here is a plot of the samples generated by this test run:



I then ran the program for n = 1000 and n = 10,000 and stored the output. Then using the output, I created the following histograms which show that as the number of samples increase, the random sample distribution approaches the exponential (the blue curve):





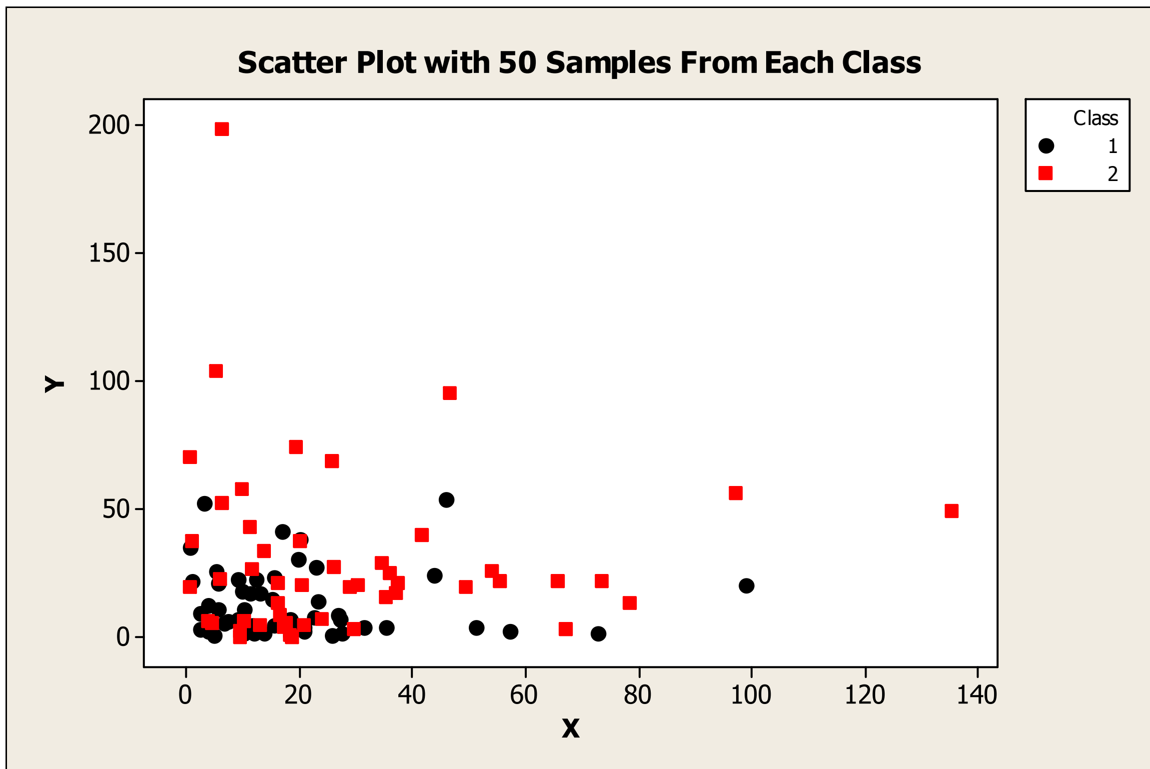


The following code uses the randExp function from the previous code to generate a matrix whose rows are [x y l], where x and y are random samples from specific exponential distributions and l is the class label (either 1 or 2). When l = 1, x comes from

an exponential distribution with mean 20 and y comes from an exponential distribution with mean 15. When l = 2, x and y both come from an exponential distribution with mean 30. In particular, this code makes a 3x100 matrix with 50 samples from each class and stores it in a file called "trainset.txt":

```
import java.util.*;
import java.io.*;

public class MachLearn
{
	/* makeTrainSet creates a 3x2n matrix where each row
	 * is [x y L] where x comes from an exponential
	 * distribution with mean 1/thetaLx, y comes from an
	 * exponential distribution with mean 1/thetaLy, and
	 * L is the class label.  n samples are made from each
	 * class.
	 */
	public static double[][] makeTrainSet(double theta1x,
	                                      double theta1y,
	                                      double theta2x,
	                                      double theta2y,
	                                      int n)
	{
		double[][] res = new double[2*n][3];

		// Make class 1 samples
		double[] x = Exponential.randExp (theta1x, n);
		double[] y = Exponential.randExp (theta1y, n);
		for (int i = 0; i < n; i++)
		{
			res[i][0] = x[i];
			res[i][1] = y[i];
			res[i][2] = 1;
		}

		// Make class 2 samples
		x = Exponential.randExp (theta2x, n);
		y = Exponential.randExp (theta2y, n);
		for (int i = n; i < 2*n; i++)
		{
			res[i][0] = x[i-n];
			res[i][1] = y[i-n];
			res[i][2] = 2;
		}
		return res;
	}

	// Output a specific training set to file "trainset.txt"
	public static void main (String[] args) throws Exception
	{
		FileOutputStream out = new FileOutputStream("trainset.txt");
		PrintStream p = new PrintStream (out);

		double[][] trainSet
		  = makeTrainSet(0.05, (double) 1.0/15,
		                 (double) 1.0/30, (double) 1.0/30,
		                 50);
		for (int i = 0; i < trainSet.length; i++)
			p.println (trainSet[i][0] + " " +
			           trainSet[i][1] + " " +
			           trainSet[i][2]);
	}
}
```

Here is a scatter plot using a training set produced by this code:



Finally, the next bit of code attempts to classify a set of (x,y) points in two ways: The first uses Baysian classification (making use of the true underlying exponential distributions) by estimating the likelihood that the point is from class 1 (or class 2) and then choosing the maximum of the two; the second uses Gaussian classification by assuming that the underlying distributions are both multivariate Gaussian (with independent coordinates) with means and standard deviations equal to the statistical means and standard deviations of the training set, and then computes likelihood similar to the first way. Here is the code:

```
import java.io.*;
import java.util.*;

public class Classifier
{
	public static void main (String[] args) throws Exception
	{
		int numErrors = 0; // tracks number of classifying errors
		double[][] trainset = new double[100][3];

		// Put samples in matrix
		Scanner s = new Scanner (new File ("trainset.txt"));
		for (int i = 0; i < 100; i++)
			for (int j = 0; j < 3; j++)
				trainset[i][j] = Double.parseDouble (s.next());

		// Classify samples according to known exponentials
		double thetax1 = 0.05;
		double thetay1 = (double) 1.0/15;
		double thetax2 = (double) 1.0/30;
		double thetay2 = thetax2;
```

```
for (int i = 0; i < 100; i++)
{
        // P(w|x) = aP(x|w)P(w)
        // a and P(w) are constant (since 50 samples from
        // each distribution), therefore just need to
        // compute P(x|w) to classify sample

        // Assuming independence
        // P((x,y) of class j) = P(x class j)P(y class j)
        //      = thetaxj*e^(-thetaxj*x)*thetayj*e^(-thetayj*y)
        double x = trainset[i][0];
        double y = trainset[i][1];
        double class1prob = thetax1*thetay1*Math.exp(-thetax1*x-
                                                        thetay1*y);
        double class2prob = thetax2*thetay2*Math.exp(-thetax2*x-
                                                        thetay2*y);

        // We classify according to higher probability.
        // If actual class different from computed class, then
        // get error
        if (class1prob > class2prob && trainset[i][2] != 1.0)
              numErrors++;
        if (class1prob < class2prob && trainset[i][2] != 2.0)
              numErrors++;
}

System.out.println ("Bayes Classification made " + numErrors + "
                    Errors out of 100");

// Classify samples according to multivariate Gaussian
// distributions
numErrors = 0;

// Need to get statistical means and standard deviations
// (assume independence of x and y coordinates)
double meanx1 = 0;
double meany1 = 0;
double meanx2 = 0;
double meany2 = 0;
double sdx1 = 0;
double sdy1 = 0;
double sdx2 = 0;
double sdy2 = 0;

for (int i = 0; i < 100; i++)
{
        if (trainset[i][2] == 1.0)
        {
              meanx1 += trainset[i][0];
              meany1 += trainset[i][1];
              sdx1 += trainset[i][0]*trainset[i][0];
              sdy1 += trainset[i][1]*trainset[i][1];
        }
        else
        {
              meanx2 += trainset[i][0];
              meany2 += trainset[i][1];
              sdx2 += trainset[i][0]*trainset[i][0];
              sdy2 += trainset[i][1]*trainset[i][1];
        }
}
meanx1 = meanx1/50;
```

```
            meanx2 = meanx2/50;
            meany1 = meany1/50;
            meany2 = meany2/50;
            // sd = sqrt(E(x^2) - E(x)^2) = sqrt(E(x^2) - mean^2)
            sdx1 = Math.sqrt(sdx1/50 - meanx1*meanx1);
            sdx2 = Math.sqrt(sdx2/50 - meanx2*meanx2);
            sdy1 = Math.sqrt(sdy1/50 - meany1*meany1);
            sdy2 = Math.sqrt(sdy2/50 - meany2*meany2);

            // Now classify samples according to these Gaussians
            for (int i = 0; i < 100; i++)
            {
                  double x = trainset[i][0];
                  double y = trainset[i][1];

                  // P((x,y) of class j) = P(x class j)P(y class j)
                  // P(x class j) = 1/(sdxj*sqrt(2pi))*e^(-(x-
                  //                                      meanxj)^2/2sdxj^2)
                  double class1prob = (1.0/(sdx1*Math.sqrt(2*Math.PI)))
                                      *(1.0/(sdy1*Math.sqrt(2*Math.PI)))
                                      *Math.exp(-(x-meanx1)*(x-
                                                  meanx1)/(2*sdx1*sdx1))
                                      *Math.exp(-(y-meany1)*(y-
                                                  meany1)/(2*sdy1*sdy1));

                  double class2prob = (1.0/(sdx2*Math.sqrt(2*Math.PI)))
                                      *(1.0/(sdy2*Math.sqrt(2*Math.PI)))
                                      *Math.exp(-(x-meanx2)*(x-
                                                  meanx2)/(2*sdx2*sdx2))
                                      *Math.exp(-(y-meany2)*(y-
                                                  meany2)/(2*sdy2*sdy2));

                  if (class1prob > class2prob && trainset[i][2] != 1.0)
                        numErrors++;
                  if (class1prob < class2prob && trainset[i][2] != 2.0)
                        numErrors++;
            }

            System.out.println ("Gaussian Classification made " + numErrors +
                              " errors out of 100");
      }
}
```

This code was run on the training set shown in the scatter plot and yielded the following results:

Command Line:

A:\csc421>java -cp . Classifier
Bayes Classification made 30 errors out of 100
Gaussian Classification made 37 errors out of 100

Note that the Gaussian classifier made more errors than the Bayes classifier. This makes sense since the Gaussian is using a wrong distribution (normal and not exponential) to classify the points by. A relatively high percentage of errors occurred because the two classes are not drastically different from one another.