Estimation for Speech Processing with Matlab or Octave

This paper is available at

`http://dpovey.googlepages.com/jhu_lab.pdf`

Daniel Povey

June, 2009

Overview

- This tutorial will introduce you to Octave and demonstrate some simple calculations used in speech processing (including calculating means, variances, etc.)

- Those who successfully complete the first part are asked to apply what they learnedf to translate some code for computing a mixture-of-Gaussians model from C++ into Octave.

- Those who complete the task above are asked to write semi-tied covariance estimation code in Octave, starting from some C++ example code. (Probably no-one will get this far).

Logging in to the CLSP machines

- Octave is a free replacement for Matlab– we are using it since CLSP does not have enough Matlab licences for everyone.
- You will need more than one window open, so from a terminal on a Mac type the following:

  ```
  xterm &
  ```

- From a terminal on a Mac, type (using your own userid, not dpovey):

  ```
   ssh -X dpovey@login.clsp.jhu.edu
   # type your password when prompted
  ```

- To be assigned a compute node to work on, type into the prompt:

  ```
   dpovey@login:~$ qlogin
  # and type your password
  ```

- Type `xterm&` from the prompt to get a new terminal on CLSP; if it doesn't work, do the same process described above from the other Mac terminal you opened.

Running octave on the CLSP machines

- Type: `octave 2>&1`
  on one of the CLSP xterm windows. The `2>&1` part would not normally be necessary, it is needed to fix a problem with the `qlogin` program. You should see a prompt like this:

  ```
  octave:1>
  ```

- If it says "command not found", log off (type `exit`) and type `qlogin` again: your machine may not have octave installed.

- Assign a simple variable:

  ```
  x = 1
  ```
  Octave will repeat back to you: `x = 1`. To turn this echoing off, use a semicolon:

  ```
  x = 1;
  ```

Executing a loop in Octave

- Make a simple loop:

  ```
  for x=1:3
    x*2
  end
  ```

  Octave will print out:

  ```
  ans = 2
  ans = 4
  ans = 6
  ```

  Each time `x*2` is evaluated, because there is no semicolon it prints out the answer as `ans`. `ans` is a special variable that gets assigned whenever an expression is evaluated and not explicitly assigned to another variable name. Type `ans` and see what its value is.

Vectors in Octave

- Type `y = [1 2 3]` into Octave. `y` is a row vector.
- Type `z = [1;2;3]` into Octave. `z` is a column vector.
- Type `z'` into Octave. The “single-quote” (') operator means transpose. The result is a row vector.
- Type `y * z` into Octave. Multiplying a row vector by a column vector gives a scalar. Check that the result is what you expect.
- Type `z * y` into Octave. This is the outer product between two vectors (the result is a matrix). If this does not make sense think of `z` and `y` as 3 by 1 and 1 by 3 matrices respectively. Remember that the result will always have the same number of rows as the LHS and the same number of columns as the RHS. Ask your teammates if you don't remember how matrix multiplication is defined.
- Try to use the transpose operator (single-quote) to generate the same output as your last command in a different way. Remember that `y` and `z` only differ by being transposed.
- What do you expect will happen if you type `y*y` ? Try it.

Matrices in Octave

- Type `M = [1 2; 3 4]` into Octave. `M` is a matrix.

- Create a two dimensional column vector called v (with any values) using the types of commands described in the last slide (but two not three dimensions).

- Type `M*v` into Octave. The result should be a column vector.

- Type `u = M*v` to assign the same thing to a variable `u`.

- Row vectors can multiply matrices on the left. Use the transpose operator (single-quote) to multiply `v` (transposed) on the left by `M` on the right. Is it the same?

- If you transpose `M` also in that multiplication, the result should be the same as `u` (but transposed). See if you can verify this.

- Type `size(M)`. This shows the dimension of `M`. Try to find the size of `y` and `z`. What does the answer tell you about how Octave represents vectors internally? Type `size(1)` ... how do you interpret the answer?

Rows, columns and sub-blocks of matrices in Octave

- Type `N = [1 2 3; 3 4 5; 6 7 8]` into Octave.
- You can set variable `v` to the first row of `N` by typing `v = N(1,:)`.
- The expression `N(:,2)` will access the second column. Try to work out what will be displayed before you type this expression in and see if you get it right.
- The expression `N(1:2,1:2)` will return the top left four elements.
- Type `N(1,:) = [ 0 0 0 ]` and see what happens.
- Type just `1:5` into the prompt. What kind of variable does this get expanded to?
- Type `1:5'` and `(1:5)'` Are they different? See if you can figure out why. Type `size(5)` to help make sense of what the first version does.
- Think back to before when you typed `N(1:2,1:2)`. What do you think the expressions `1:2` were evaluated to during Octave's processing of the expression? Can you type in an equivalent statement in a different way?

Special matrices in Octave

- Type `zeros(2,2)` into Octave.
- Remembering that vectors are represented as matrices with one row or column, work out how to create a column vector of size 5 of all zeros using the `zeros` function. Try to get it right the first time.
- The function `ones` is like `zeros` but with all ones. Try to create a row vector of all ones with size 10, again on the first try.
- You can multiply a vector or matrix by a scalar (e.g. `5 * [1 1]`). Using one typed function, try to create a matrix of size ten by ten, with all tens in it.
- The function `diag` has two meanings. Given a matrix it returns a column vector of the diagonal elements; given a vector it returns a matrix with those elements on the diagonal and otherwise zeros. Type `diag(v)` or `diag([1 2; 3 4])` to see examples.
- See if you can determine what `diag(M)` does if `M` is not square.
- Can you work out how to use two of the commands we just introduced to create a unit matrix of size 5, i.e. a 5 by 5 matrix that has ones on the diagonal and zeros everywhere else?

Matrix inversion

- Type `N = inv(M)` into Octave. This is the matrix inverse of `M`.
- Evaluate `M*N` and `N*M`. In both cases it should be the unit matrix $\mathbf{I} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. This is the definition of matrix inverse.
- Type `N*u` into Octave. It should be the same as `v`.
- To see why consider the equation `u = M*v` (which is where we got `u` from) and multiply both sides on the left by `N`.
- This would give us `N*u = N*M*v`, and `N*M` is the unit matrix which leaves `v` the same (this is the special property of the unit matrix). So we have `N*u = v`.
- This is matrix algebra. We don't just multiply both sides of the equation by `M`, we multiply "on the right" or "on the left". This is because matrix multiplication is sensitive to the order of the arguments (it is not commutative).

Function files

- Octave (like Matlab) allows you to define function files.
- In the other (non-octave) terminal window that you previously prepared, begin editing a file in your home directory called `test.m` If you do not know how to use any editor in UNIX, `emacs` is probably easiest. Type `emacs test.m &`.
- Type into the file:

  ```
  function y = test(x)
  y = 2*x;
  ```

  It may be helpful if you are using emacs to type `<Esc> x octave-mode<enter>` or `<alt-x>octave-mode<enter>` (the spaces are not to be typed; `<enter>` and `<esc>` and `<alt-x>` are not to be typed in literally!
- Remember to save it! It is important that the name of the file ("test") be the same as the function name inside the file.
- Type into your Octave window:

  ```
  test(5)
  ```

- Does it return what you expected?
- The return value is dictated not by a `return` statement as in most languages but by the expression before the = sign on the first line.

Function files with multiple arguments and return values, who, whos

- Try changing the file `test.m` to the following and re-running with two arguments:

  ```
  function [y,z] = test(w,x)
  y = 2*w;
  z = 2*x;
  ```

  You could call as `test(4,5)` or something like that. But this only gives you the first return value (`y`).

- To access both the return values you must type `[a,b] = test(4,5)`.

- Type `who`. This tells you what variables are defined.

- Type `whos`. This gives you their dimensions.

If-statements, comments, recursive functions

- Type into Octave:

```
a=1;
if a>2,
  b=a
else
  b=2
end
```

- Type into Octave: `b # blah`. Everything after the `#` does nothing.

- As with most languages, functions can call themselves (recursion). Create a file called `fib.m` that computes the n'th element of the Fibonacci seqence using the recursion

$$\text{fib}(n) = \left\{ \begin{array}{l} n < 2 \rightarrow n \\ n \geq 2 \rightarrow \text{fib}(n-1) + \text{fib}(n-2) \end{array} \right\}$$

- Try to verify that fib(8) = 21.

Loading data into Octave

- At your non-octave (shell) prompt, type:

  ```
  ln -s /home/dpovey/train.dat .
  ln -s /home/dpovey/test.dat .
  ```

  This creates soft links (UNIX speak for 'shortcuts') from these ascii data files to your home directory. Type `head train.dat` to see what the speech feature data in these files looks like, and `wc *.dat` to see the number of lines in these files. These are UNIX shell commands.

- At your octave prompt, type: `M = load('train.dat');`

- Type `size(M)` to see the dimension of `M`. Type `size(M,1)` and `size(M,2)` to see the number of rows and columns separately (useful when defining functions).

Computing statistics from data

- Create a file called `mean.m`. Type into it:

```
function m = mean(M);
nrows = size(M,1);
ncols = size(M,2);
m = zeros(1,ncols);
for r=1:nrows,
   m = m + M(r,:);
end
m = m * (1/nrows);
m = m';
```

- Type `mean(M)` The result should be a column vector.

Variance of a file.

- Create a file called `var.m`. Try to get it to return a vector containing the diagonal variance of the samples. The variance of d'th dimension of a sequence of samples $\mathbf{x}(1) \ldots \mathbf{x}(T)$ is:

$$\sigma_d^2 = \left(\frac{1}{T} \sum_{t=1}^{T} x_d(T)^2 \right) - \mu_d^2$$

  , where μ_d is the mean of the d'th dimension. Note that the variance itself is σ_d^2, we consider it a single variable even though it is written as a square.

- The variance should always be positive.

- Remember that a sample $\mathbf{x}(t)$ would correspond to the t'th *row* of `M`. The dimension d is the column index (1 to 39 for this data).

- You can access the matrix `M` element by element, or you may be able to use the `.*` operator in Octave, which multiplies the corresponding elements, e.g. `[1 2] .* [1 2]` returns `[1 4]`.

Log likelihood of data

- Here is some C code given in a previous lecture for computing the log likelihood of a single vector given a Gaussian (mean and variance). Translate it into Octave in a file `diag_loglike.m`. In octave π can be accessed as `pi`.

```
float diag_loglike(float *x, float *mu, float *sigmasq, int D){
   float ans=0.0;
   for(int i=0;i<D;i++)
     ans -= 0.5*(log(2*M_PI*sigmasq[i])+(mu[i]-x[i])*(mu[i]-x[i])/sigmasq[i]);
   return ans;
}
```

- The function should probably start with something like:

```
function ll = diag_loglike(x, mu, var)
dim = size(x,1) # assume column vector
ll=0
for i=1:dim,
  ...
```

- Test it using the first row of the matrix `M`.
- Create a wrapper function in `file_loglike.m` that calls this for each row of a matrix `M` and returns the average log likelihood over all the rows.

Mixture of Gaussians estimation (bonus assignment)

- The next slide (from one of the preceding lectures) contains C code to re-estimate a mixture of Gaussians distribution; modify and translate it into Octave.
- The code in the next slide is not very complete: it relies on a reasonable initialization of the mean and variance.
- A reasonable method would be to divide the training data into blocks and initially assign each mean and variance to the mean and variance of that block.
- The weights should all be initialized the same.
- The octave function (say `reest_gaussian_mixture.m`) should probably start something like:

```
function [ Mu, Var, weights ] = reest_gaussian_mixture(M, mixtures, iters)
dim = size(M,2)
T = size(M,1)
blksz = floor(T / mixtures)
Mu = zeros(mixtures, dim) # each row of Mu will be a mean
Var = zeros(mixtures, dim) # each row of Mu will be a diagonal variance
# now initialize the rows of Mu and var...
```

Mixture of Gaussians estimation (bonus assignment): testing

- In order to test your results from the previous assignment, you will have to write versions of `diag_loglike.m` and `file_loglike.m` (with different names) that give the likelihood of data given a mixture of Gaussians.

- It will be conceptually easiest to use the `exp` function to represent the output of `diag_loglike.m` as actual likelihoods, and sum them up with the weights, before taking a log and returning the answer.

- The log likelihood given the mixture of Gaussians should increase (or become less negative) as you increase the number of mixtures, when tested on the training data.

- If you load the file `test.dat` (say using the command `N = load('test.dat');`, you can measure the log likelihood on unseen, or "held-out" data, i.e. that was not trained on.

- As you increase the number of Gaussians, beyond a certain point the log likelihood on this new data ("test data") should start to decrease. See if you can find what this point is. (Do not get stuck here if it takes too long to run; just skip this part.)

Mixture of Gaussians estimation: code (diagonal case)

```
void reest_mixture(int D, int M, int T, int iters, const float **data,
    float **mu, float **var, float *weights){
  float *count_stats = new float[M], *loglikes = new float[M];
  float **mu_stats = alloc_matrix(M,D), **var_stats = alloc_matrix(M,D);
  for(int iter=0;iter<iters;iter++){
    // set count_stats, mu_stats and var_stats to zero here!
    for(int t=0;t<T;t++){
      float log_sum=-1.0e+10; // very negative->log(zero)
      for(int m=0;m<M;m++){
        loglikes[m] = diag_loglike(data[t],mu[m],var[m],D)+log(weights[m]);
        log_sum=log_add(log_sum,loglikes[m]);  }// log(exp(a)+exp(b));
      for(int m=0;m<M;m++){
        float gamma_m_t = exp(loglikes[m]-log_sum); // posterior of mix.
        count_stats[m] += gamma_m_t;
        for(int d=0;d<D;d++){
          mu_stats[m][d] += data[t][d]*gamma_m_t;
          var_stats[m][d] += data[t][d]*data[t][d]*gamma_m_t; }}}
    for(int m=0;m<M;m++){
      weights[m] = count_stats[m] / T;
      for(int d=0;d<D;d++){
        mu[m][d] = mu_stats[m][d]/count_stats[m];
        var[m][d] = var_stats[m][d]/count_stats[m] - mu[m][d]*mu[m][d]; }}}}
```

Semi-tied covariance estimation (double-bonus assignment)

- If you have got this far and you still have time left, attempt to implement semi-tied covariance estimation for your mixture of Gaussians model.
- The next two slides contains example C code from a previous lecture that you can translate into Octave.
- You can decide whether the make the accumulation and update separate functions
- The G statistics are a three dimensional array in the C code. Octave/Matlab allow three dimensional arrays but not in a very clean way. Here is an example of how to create them.
- Type `G = zeros(2,2,2)` . What is returned is a three dimensional array. The third dimension is special.
- If you type `G(:,:,1)` this will return a normal matrix. But `G(1,:,:)` will not.

Semi-tied covariance transform (STC): accumulation code

- Assumes any existing STC transform has already been applied to the means.

```
void stc_accu(int M, int D, int T, float **data, float **A, float **mu,
              float **var, float *weights, float ***G){
  float *Ax = new float[D], *loglikes = new float[M];
  for(int t=0;t<T;t++){
    float log_sum=-1.0e+10; float *x=data[t];
    do_stc_transform(Ax, A, x); // Ax = A*x^+
    m_v_prod(Ax, A, x, D,D); // Ax := A * x;
    for(int m=0;m<M;m++){
      loglikes[m] = diag_loglike(Ax,mu[m],var[m],D)+log(weights[m]);
      log_sum=log_add(log_sum,loglikes[m]); }
    for(int m=0;m<M;m++){
      float gamma_m_t = exp(loglikes[m]-log_sum);
      for(int d=0;d<D;d++)
        for(int e=0;e<D;e++)
          for(int f=0;f<D;f++) // in reality would do f<=e: symmetric.
            G[d][e][f]+=gamma_m_t*(Ax[e]-mu[m][e])*(Ax[f]-mu[m][f])
              /var[m][d];
    }
  }
}
```

Semi-tied covariance transform (STC): update code

```
void stc_upd(int D, float **A_in, float ***G, float beta, float **means, int M){
  float **A = alloc_matrix(D,D), **Ainv = alloc_matrix(D,D),
    **GInv = alloc_matrix(D,D);
  float *cd=new float[D],*tmp=new float[D],*anew=new float[D];
  for(int d=0;d<D;d++) A[d][d]=1.0; // Set A to unit matrix.
  float tot_objf_change=0;
  for(int iter=0;iter<10;iter++){
    for(int d=0;d<D;d++){
      invert_matrix(Ainv,W,D,D);  // Ainv := inverse(A);
      invert_matrix(Ginv,G[d],D,D); // Ginv:=inverse(G[d]);
      for(int e=0;e<D;e++) cd[e] = Ainv[e][d]; // c := d’th column of Ainv.
      float a=vmv_prod(cd,Ginv,cd,D,D), c=-beta, f=(-0 + sqrt(0*0 -4*a*c))/(2*a);
      for(int e=0;e<D+1;e++) tmp[e]=f*cd[e]; // tmp := c*f.
      m_v_prod(anew,Ginv,tmp,D,D); // anew:=Ginv*tmp.
      float objf_change = beta*log(beta/f) // log-det term
        -0.5*(vmv_prod(anew,G[d],anew)-vmv_prod(A[d],G[d],A[d]));
      Assert(objf_change >= 0); tot_objf_change += objf_change;
      for(int e=0;e<D;e++) A[d][e] = anew[e];
    }} // should print objf value on each iter.
  for(int m=0;m<M;m++) // Transform means by new part of transform.
    m_v_prod(means[m],A,means[m],D,D); //assume function works w/ repeat args.
  // Assume A_in is unit if first iteration.
  m_m_prod(A_in, A, A_in, D,D,D); // A_in:=A*A_in.  Assume works w/ repeat args.
}
```