Genome Sequence Analysis: Assignment 1

University of Cambridge

Henrik Åhl

January 27, 2017

Preface

This is an assignment report in connection to the *Genome Sequence Analysis* module in the Computational Biology course at the University of Cambridge, Michaelmas term 2016. All related code is as of January 27, 2017 available per request by contacting hpa22@cam.ac.uk, or by visiting `https://github.com/supersubscript/compbio/tree/master/src/assignments/gsa1`, where the corresponding lecturer should have been made a collaborator by the assignment deadline. This repository should also contain the auxilliary files used for the assignment, if any.

The approach for the solutions is done after the guidelines specified by Rabiner [1], and as summarized by Shen [2]. All code is implemented using Python, and in particular by usage of the `numpy` package.

Exercises

1

```python
# Get index for letter (symbol) occurring.
def get_ind(probs):
    rand = random.random()
    for ii in xrange(len(probs)):
        if rand < probs[ii]:
            return ii


# Read in transition matrix and initial distribution from files.
# If sumrows == TRUE, sum the rows
def read_and_output(mat_file, idist_file, seq_length, sumrows, transpose):
    # Read in data
    trans_mat = np.loadtxt(open(mat_file, "r"))
    init_dist = np.loadtxt(open(idist_file, "r"))

    # Act on the optional things
    if transpose:
        trans_mat = np.transpose(trans_mat)
    if sumrows:
        trans_mat = np.cumsum(trans_mat, 1)

    # Fill out the output probabilistically
    output = seq_length * [None]
    output[0] = np.random.choice(np.arange(0, len(trans_mat)), p=list(init_dist))
    for ii in xrange(1, seq_length):
        output[ii] = get_ind(trans_mat[output[ii - 1], :])

    "".join(str(s) for s in output)
    return output

```

```
mat_file = "/home/henrik/compbio/src/assignments/gsa1/report/mat_file.dat"
idist_file = "/home/henrik/compbio/src/assignments/gsa1/report/idist_file.dat"
output = read_and_output(mat_file, idist_file, 115, True, False)
```

Evoking the `print` command we get the following:

```
0011111111111111111111100100111100001110101111001111110111100111111111111111110110
1111110110011110100111111101111100
```

Our program is thus able to produce an output corresponding to that of a Markov chain.

2

For this exercise, as we are only to read a single Markov chain, the initial distribution will correspond to a 100 % probability of ending up in the first state in the sequence.

```
# Get transtion matrix and initial distribution from one (!) sequence
def infer_mat_and_init(sequence):
    # Get some data and create the matrices we need.
    elements = np.unique(list(sequence))
    trans_mat = np.zeros((len(elements), len(elements)))
    init_dist = np.zeros(len(elements)) # This is specific to this assignment
    init_dist[int(sequence[0])] = 1  # Count the first one

    # Count the occurrences of all events
    for ii in xrange(1, len(sequence)):
        trans_mat[int(sequence[ii - 1]), int(sequence[ii])] += 1

    # What are the probabilities?
    for ss in xrange(len(trans_mat)):
       trans_mat[ss,] /= np.sum(trans_mat[ss,])

    return trans_mat, init_dist


trans_mat, init_dist = infer_mat_and_init("
    0101010101011110100101011001010010101010101010010101")
```

Again, looking at the inferred transfer matrix and initial distribution tells us that our program appears to give an output corresponding to our expectations:

$$A = \begin{pmatrix} 0.15 & 0.85 \\ 0.85 & 0.15 \end{pmatrix}, \qquad \mu^0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

3

```
# Parameters
S = np.array([0, 1])
V = np.array([0, 1, 2, 3, 4])
A = np.matrix("0.8, 0.2;"
              "0.1, 0.9")
B = np.matrix("0.2, 0.5, 0.2, 0.1, 0.0;"
              "0.0, 0.1, 0.4, 0.4, 0.1")
mu0 = np.array([.5, .5])


# Emit a sequence given some transition and emission matrices, as well as
# an initial distribution.
def emit_sequence(seq_length, trans_matrix, emiss_matrix, init_dist):
    c_trans_mat = np.cumsum(trans_matrix, 1)
    c_emiss_mat = np.cumsum(emiss_matrix, 1)
```

```python

    emit = np.zeros(seq_length)
    hidden = np.zeros(seq_length)
    hidden[0] = np.random.choice(np.arange(0, len(trans_matrix)), p=list(init_dist))
    emit[0] = get_ind((c_emiss_mat[int(hidden[0]), :]).tolist()[0])

    # Parse sequence
    for ii in xrange(1, seq_length):
        hidden[ii] = get_ind((c_trans_mat[int(hidden[ii - 1]), :]).tolist()[0])
        emit[ii] = get_ind((c_emiss_mat[int(hidden[ii]), :]).tolist()[0])

    emit = "".join(str(int(x)) for x in emit)
    hidden = "".join(str(int(x)) for x in hidden)
    return emit, hidden


emit, hidden = emit_sequence(115, A, B, mu0)
```

The result of our emitted sequence, as well as the underlying state, can be seen in fig. 1, and appears to correspond well to the expected behaviour.

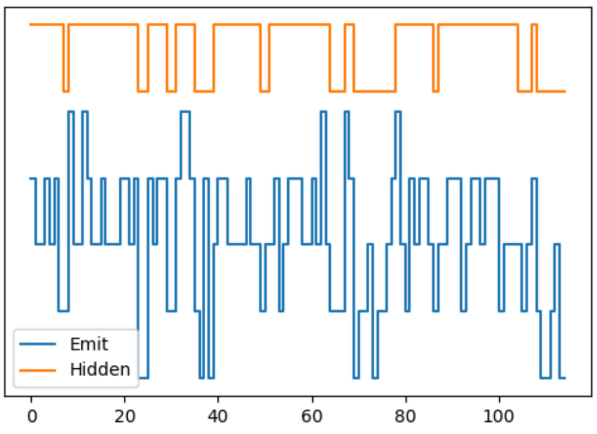

Figure 1: Output from the program emitting a sequence according to the designated parameters. 0-valued states tend to produce the lowest emission values on average, whereas the 1-valued states behave in the opposite manner.

4

We define a method for reading in a sequence from a file (which has been produced with our previously defined emission function).

```python
# Read in sequence from file.
def read_sequence(file, skipFirst):
    seq = ""
    with open(file, 'r') as in_file:
        if skipFirst:
            in_file.readline()
        seq = in_file.read().replace('\n', '')
    return seq


# Use the forward algorithm to determine (log) likelihood of sequence
def forward(sequence, trans_matrix, emiss_matrix, hidden_states, init_dist):
    # Function for getting normalisation constant
    def get_constant(a):
        sum = np.sum(a)
        return 1.0 / len(trans_matrix) if sum == 0 else 1.0 / sum

    # Pre-allocate
    fw = np.zeros((len(hidden_states), len(sequence)))
    constants = []

    # Initialise
    for ss in xrange(len(hidden_states)):
        fw[ss, 0] = init_dist[ss] * emiss_matrix[ss, int(sequence[0])]
    const = get_constant(fw[0, :])
    constants.append(const)

    # Scale accordingly
    for ss in xrange(len(hidden_states)):
        fw[ss, 0] *= const

    # Now go through the rest of the sequence
    probs = np.zeros(len(hidden_states))
    for ii in xrange(1, len(sequence)):
        for ss in xrange(len(hidden_states)):
            probs[ss] = np.sum(
                (fw[substate, ii - 1] * trans_matrix[substate, ss] * emiss_matrix[ss, int(
                    sequence[ii])]) for
                substate in xrange(len(hidden_states)))

        # Scale again
        const = get_constant(probs)
        constants.append(const)
        for ss in xrange(len(hidden_states)):
            fw[ss, ii] = const * probs[ss]

    # Sum up the probabilities; disregard if 0
    ln_prob = -np.sum([math.log(const) if const != 0 else 0 for const in constants])
    return ln_prob, constants, fw

out_sequence, __ = emit_sequence(115, A, B, mu0)
seq_file = '/home/henrik/compbio/src/assignments/gsa1/random_output_sequence.dat'
with open(seq_file, "w") as f_out:
    f_out.write(out_sequence)
    f_out.close()
that_same_sequence = read_sequence(seq_file, False)
prob, consts, fw = forward(that_same_sequence, A, B, S, mu0)
```

5

We use the *Saccharomyces cerevisiae* chromosome III genome from Ensembl [3]. The genome is binned by taking intervals of size 100, and subsequently identifying how many G or C bases are contained. In our implementation, bins are sequential and do not overlap.

Trying to approximate the correct distribution of categories, we label bins after percentage-wise GC content. Doing this rough fitting, the best fit appears to be found around $0 - 28.5$ %, $28.5 - 34.2$ %, $34.2 - 40.0$ %, $40.0 - 50.0$ % and a $50 - 100$ % split for label 1-5 respectively, with upper boundaries exclusive. As our genome is not evenly divisble by the bin size, the last bin will be misrepresentative. However, as our genome is large, and we thus will also have a large set of bins, the effects of this will be marginal. Therefore we choose to disregard this effect.

```
# We cheat a little and put the remnants in a bin of their own, i.e. if the last bin only has
    e.g. 20 elements, the GC
# content in this will be "the number of GC bases in those 20 bases"/bin_size, which is wrong
    , but we'll have to
# live with it.
def calc_gc(sequence, bin_size):
    gc_cont = []
    bin_size = float(bin_size)
    for ii in xrange(int(math.ceil(float(len(sequence)) / bin_size))):
        ceil = int((ii + 1) * bin_size)
        if ceil > len(sequence):
            ceil = len(sequence)
        bin = sequence[int(ii * bin_size): ceil]
        G = bin.count("G")
        C = bin.count("C")
        gc_cont.append((G + C) / bin_size)
    return gc_cont


# Read genome
sc_file = '/home/henrik/compbio/src/assignments/gsa1/sc_gen.fa'
sc_gen = read_sequence(sc_file, True)
gc = calc_gc(sc_gen, 100)


# Relabel according to predefined cuts
def relabel(seq, cuts):
    relab = []
    for ii in seq:
        for cut in xrange(len(cuts)):
            if ii <= cuts[cut]:
                relab.append(cut)
                break

    relab = "".join(str(x) for x in relab)
    return relab

# Compare the two strings
relab = relabel(gc, [.285, .342, .40, .5, 1.])  # Split by percentages
model3_seq, model3_hidden = emit_sequence(len(relab), A, B, mu0)
log_p, const, fw_seq = forward(relab, A, B, S, mu0)
```

The final classification can be seen in fig. 2, where the two distributions are shown.

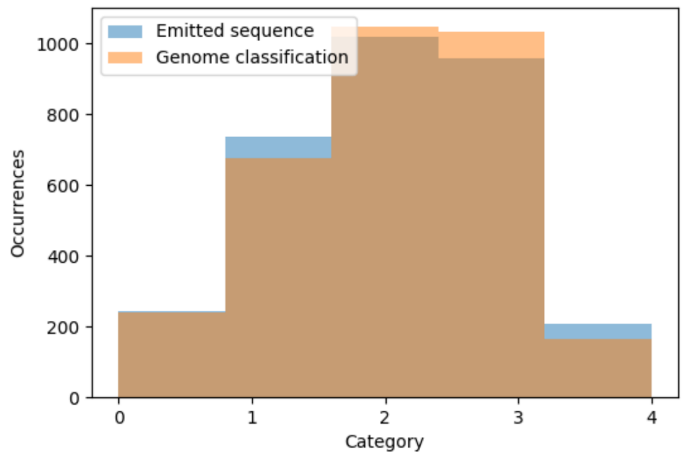

Figure 2: Distributions of emission values according to our given model and of the *Saccharomyces cerevisiae* genome after percentage-wise labelling

.

6

```python
# We only use this for Baum-Welch, so we use the same constants as in our forward case
def backward(sequence, trans_matrix, emiss_matrix, hidden_states, init_dist, constants):
    # Preallocate for our backwards sequence
    bw = np.zeros((len(hidden_states), len(sequence)))

    # Initialise
    for ss in xrange(len(hidden_states)):
        bw[ss, len(sequence) - 1] = 1.
        bw[ss, len(sequence) - 1] = constants[len(sequence) - 1]

    # Go through the rest of the sequence, continue from the back end
    for ii in xrange(len(sequence) - 2, -1, -1):
        b = np.zeros(len(hidden_states))
        for ss in xrange(len(hidden_states)):
            b[ss] = np.sum([(bw[substate, ii + 1] * trans_matrix[ss, substate] *
                             emiss_matrix[substate, int(sequence[ii + 1])]) for substate in
                                 xrange(len(hidden_states))])

        # Again, scale
        for ss in xrange(len(hidden_states)):
            bw[ss, ii] = constants[ii] * b[ss]

    ln_prob = -np.sum([math.log(c) if c != 0 else 0 for c in constants])
    return ln_prob, bw


# Now we can go onto estimating our matrices
def baum_welch(sequence, trans_matrix, emiss_matrix, hidden_states, init_dist):
    # Get those lists we need
    p1, constants, fw = forward(sequence, trans_matrix, emiss_matrix, hidden_states,
        init_dist)
    p1, bw = backward(sequence, trans_matrix, emiss_matrix, hidden_states, init_dist,
        constants)

    # Initial distribution
    for ss in xrange(len(hidden_states)):
        init_dist[ss] = fw[ss, 0] * constants[0] * bw[ss, 0]
    init_dist /= np.sum(init_dist)

    # New state matrix
    for ss in xrange(len(hidden_states)):
        for substate in xrange(len(hidden_states)):
            # Sum up the estimate, normalise and assign
            new_est = np.sum([(fw[ss, ii] * bw[substate, ii + 1] * trans_matrix[ss, substate]
                 * emiss_matrix[
                substate, int(sequence[ii + 1])]) for ii in xrange(len(sequence) - 1)])
            norm = np.sum([(fw[ss, ii] * bw[ss, ii] / constants[ii]) for ii in xrange(len(
                sequence) - 1)])
            trans_matrix[ss, substate] = new_est / norm if norm != 0 else 1. / len(
                hidden_states)

    # New emission matrix (structure same as above)
    for ss in xrange(len(hidden_states)):
        for jj in xrange(emiss_matrix.shape[1]):
            new_est = np.sum([(fw[ss, ii] * bw[ss, ii] / constants[ii] if int(sequence[ii])
                == jj else 0) for ii in
                              xrange(len(sequence))])
            norm = np.sum([(fw[ss, ii] * bw[ss, ii] / constants[ii]) for ii in xrange(len(
                sequence))])
            emiss_matrix[ss, jj] = new_est / norm if norm != 0 else 1. / emiss_matrix.shape
                [1]
    return p1

change = 99999999999999
prev   = change
while change > 0.000001:
    prob = baum_welch(relab, A, B, S, mu0)
    change = prev - prob
    prev = prob
```

We here achieve a steady increase in the log likelihood for the given sequence, showing that our model adapts to the data given. Ultimately, our matrices take the shapes of

$$A = \begin{pmatrix} 0.91 & 0.09 \\ 0.05 & 0.95 \end{pmatrix}, \quad B = \begin{pmatrix} 0.01 & 0.05 & 0.21 & 0.59 & 0.14 \\ 0.11 & 0.31 & 0.40 & 0.18 & 0.00 \end{pmatrix}.$$

Our final log likelihood for the same situation evaluates as:

```
-4251.17034431
```

7

```
# Retrieve the most likely input hidden states given and output sequence and
# the corresponding matrices.
def viterbi(sequence, trans_matrix, emiss_matrix, hidden_states, init_dist):
    # Define some things we're gonna use.
    constants = np.zeros(len(sequence))
    solution = np.zeros(len(sequence))
    dyn_mat = np.zeros((len(hidden_states), len(sequence)))
    store_path = np.zeros((len(hidden_states), len(sequence)))

    # Initialise rows
    dyn_mat[:, 0] = init_dist * emiss_matrix[:, int(sequence[0])]
    constants[0] = 1.0 / np.sum(dyn_mat[:, 0])
    dyn_mat[:, 0] *= constants[0]

    # Fill out the table
    for ii in xrange(1, len(sequence)):
        for ss in xrange(len(hidden_states)):
            probabilities = np.diag(dyn_mat[:, ii - 1]) * trans_matrix[:, ss]  # Probability
                of coming from either state
            store_path[ss, ii], dyn_mat[ss, ii] = max(enumerate(probabilities), key=
                itemgetter(1))
            dyn_mat[ss, ii] *= emiss_matrix[ss, int(sequence[ii])]

        # Now we scale
        constants[ii] = 1.0 / np.sum([dyn_mat[ss, ii] for ss in xrange(len(hidden_states))])
        dyn_mat[:, ii] *= constants[ii]

    # Backtrack
    solution[len(sequence) - 1] = dyn_mat[:, len(sequence) - 1].argmax()  # last state
    for ii in xrange(len(sequence) - 1, 0, -1):
        solution[ii - 1] = store_path[int(solution[ii]), ii]

    return solution

# Use the SC genome, and translate this to a format readable
# by our implementation.
model7_seq = sc_gen
gc = calc_gc(sc_gen, 100)
intab = "ATCG"
outtab = "0123"
trantab = maketrans(intab, outtab)
model7_seq = model7_seq.translate(trantab)

# Update parameters until convergence
prev = -99999999999999
change = -prev
while change > 0.0001:
    prob = baum_welch(relab, A, B, S, mu0)
    change = prob - prev
    prev = prob

# What is the most likely sequence?
solution = viterbi(relab , A, B, S, mu0)
```

The inferred sequence along with the corresponding GC content and category is visualised in fig. 3. We can again see how our predictions appear to produce sensible results; grouping lower values with the lower state, and the contrary for the higher one. According to NCBI, roughly 38.5 % of chromosome III in yeast has high GC content [**gc˙cont**], which is somewhat well-mirrored by our 32.5 % of bins corresponding to high-GC ones. However, as our labelling is inherently fairly arbitrary, we expect some misrepresentation. More accurate would have been to optimise the fit of the categorisation to a selection of sequences, which should to give us more accurate results. The comparison is also directly dependent on the binning approach, which thus also skews the outcome somewhat. Nevertheless, we are able to see the correct pattern in our results, which should reassure us of an overall solid approach.

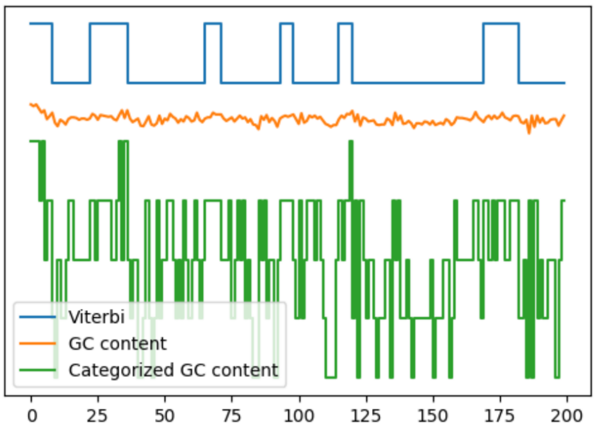

Figure 3: Inferred underlying hidden state sequence using the Viterbi algorithm, along with the corresponding categorized and non-categorized GC content under the previously determined binning scheme. Notably, the 1-valued states correspond well with the highest GC content, as expected from our matrices. Similarly, the lower state is more consistently affiliated with lower emission values.

8

Looking at fig. 4, the distribution of GC content amongst the bins is gaussian-like. It would therefore seem probable that a more accurate approach would be to have a normally distributed emission space, i.e. where we define a mean and variance according to

$$\mu_i = \frac{\sum_{m=0}^{N-1} v_m \times \gamma_m(i)}{\sum_{m=0}^{N-1} \gamma_m(i)}, \text{and}$$

$$\sigma_i^2 = \frac{\sum_{m=0}^{N-1} \gamma_m(i) \times (v_m - \mu_i)^2}{\sum_{m=0}^{N-1} \gamma_m(i)}$$

where v_m is the corresponding emission value, and $\gamma_m(i) = P(x_m = s_i|Y)$. Here x and s represent states, while Y is the observation.

Even though it is hard to determine from the looks of fig. 4, it might be viable to attempt to approximate the distribution using two states with different means and variances (i.e. corresponding to high and low GC content regions respectively). Looking at our initial emission matrix, which was able to approximate the yeast genome somewhat well, both states have distribution spaces reminiscent of gaussian-like distributions, so it seems probable that a normally distributed space would be better at capturing the finer details of the distribution of GC content in the genome.

These types of models could likely be used in helping to annotate genomes *de novo*, possibly by correlating the GC content to direct or indirect structural information about already known genes. In order to be more precise, limiting the analsysis to gene structures which are highly correlated with either high or low GC content might be a way to ensuring a higher certainty in the method. In addition, Besemer and Borodovsky [4] also discuss the possibility of matching GC content to amino acid residues, and utilizing this information in finding novel genes.

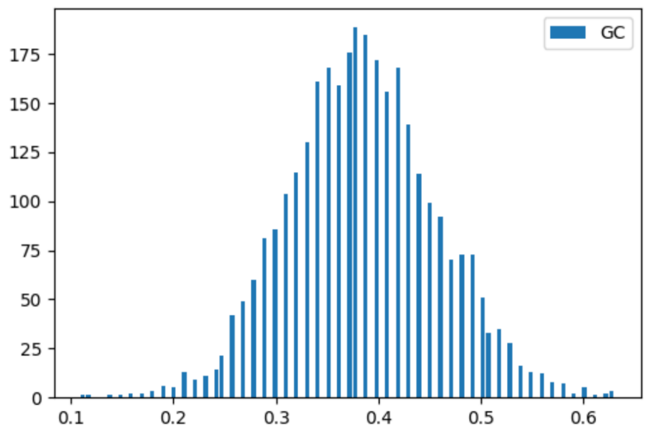

Figure 4: Histogram over GC content in our yeast. The distribution appears to take an overall gaussian shape.