Hidden markov model for CG density

Contents

1 Introduction

Most of the code was written in Python2.7 with dependencies documented in "requirements.txt"

```
numpy
#functools
matplotlib
```

1.1 Code organisation

Filename	Functionality
util.py	Defining the model as Var_hmm() class. Various I/O utilities
util_genome.py	Reading fasta and calculating CG content
forward.py	Forward-backward algorithm were added to Var_hmm() class, along with Viterbi algorithm
bulk.py	Defining chains() class that allow bulk forwarding chains

2 Question 1: File format

2.1 Model: *.hmm file

The model file *.hmm is defined with 3 sections:

1. TRANSITION: K*K transition matrix (float)
2. INTITIAL : 1*K initial distribution for internal state (float)
3. EMISSION : K*L emission probability of L states given an internal state

Filename:“Q3.hmm”

```
INITIAL
0.5  0.5
TRANSITION
0.8  0.2
0.1  0.9
EMISSION
0.2  0.5  0.2  0.1  0
0.0  0.1  0.4  0.4  0.1
```

2.2 State chains: *.echain file

A file in which each tab-delimited line is read as an emission chain, indexed from 0.

Filename:“test.echain”

```
1 1 0 1 1 0 1 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 1 1 0 1 1 0 0 0 0 1 1 1 1 0 1 1 1 0 0 1 1 1 0 0 1
```

2.3 I/O utility

1. To create a hmm instance from model file, simply do `h1 = util.Var_hmm.read_model(fname)`
2. To read a chain file, do `echain = util.read_chain(fname, dtype = 'int')`

3 Question 2: MLE estimator for transition matrix from an hidden state chain (See 9.2.1)

Run-log:

Filename:“test___mle_chain2transition.py"

```
## [0.2  0.18 0.17 0.25 0.2 ]
## Reading chains from file 'test.echain' :
## [[1 1 0 1 1 0 1 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 1 1 0 1 1 0 0 0 0 1 1 1 1 0
##   1 1 1 0 0 1 1 1 0 0 1]]
## Truncating chain to 20 elements: [1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0]
## MLE for transition matrix is:
## [[0.42857143 0.57142857]
##  [0.41666667 0.58333333]]
## TRUE transition matrix is:
## [[0.8 0.2]
##  [0.1 0.9]]
```

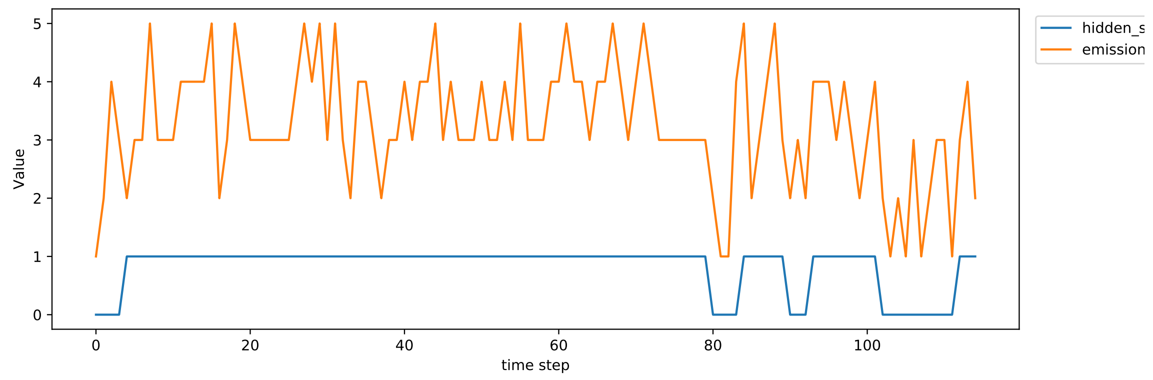

Figure 1: Trajectory over 115 time-steps sampled from “Q3.hmm”.

```
## MAE of the estimator is:
## [[0.37142857 0.37142857]
##  [0.31666667 0.31666667]]
```

4 Question 3: Sampling emissions from a hidden markov model (See 9.2.2)

This functionality is provided by “`util.Var_hmm.run_for`”

Run-log:

Filename:“test___sampling.py"

[0.19 0.2 0.16 0.2 0.25], Saving figure to: test___sampling.pdf

In order to verify that the fuction is behaving as expected, I also verified the distribution of the state is approaching stationary.

(TBA)

5 Question 4: Calculate $P(Y_0^N|model)$ with forward algorithm (See 9.2.3)

More detailed notes for deduction and implementation can be found 9.1.1

Filename:“test___forward.py"

```
## [0.19 0.16 0.23 0.2  0.22]
## Reading chains from file 'test.echain' :
## [[1 1 0 1 1 0 1 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 1 1 0 1 1 0 0 0 0 1 1 1 1 0
##   1 1 1 0 0 1 1 1 0 0 1]]
## [[1 1 0 1 1 0 1 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 1 1 0 1 1 0 0 0 0 1 1 1 1 0
##   1 1 1 0 0 1 1 1 0 0 1]]
## (47,)
## Log-likelihood: [-61.60410022]
```

6 Question 5: Analysing GC content of some real sequence (See 9.2.4)

Chromosome III of *Saccharomyces cerevisiae* was downloaded from ensemble (link). The file is preprocessed with "`util_genome`".

Run log:

Filename:"Q5_clean.py"

```
## [0.24 0.2  0.21 0.18 0.17]
## Creating 3166 chunks of length 100 from 316620 elements
## 316620
## Scheme 1:
## The border of bins are: [0.355 0.415 0.475 0.575]
## Scheme 2:
## The border of bins are: [0.26  0.355 0.415 0.475]
## Saving figure to: Q5_fig1.pdf
## Saving figure to: Q5_fig2.pdf
## --------
## Internal state space size:2
## Emission state space size:5
##
## Initial distribution of internal states is:
## [[0.5 0.5]]
##
## Transition matrix is:
## [[0.8 0.2]
##  [0.1 0.9]]
##
## Emission matrix is:
## [[0.2 0.5 0.2 0.1 0. ]
##  [0.  0.1 0.4 0.4 0.1]]
## --------
##
## Scheme 1
## [array([4, 4, 4, ..., 3, 3, 4])]
## (3166,)
## Log-likelihood: [-4618.32308088]
##
## Scheme 2
## [array([4, 4, 4, ..., 4, 4, 4])]
## (3166,)
## Log-likelihood: [-4389.99715996]
```

I calculated fraction of GC-bases within 3166 non-overlapping windows over this 316620bp sequence (figure 2). After inspecting the distribution of GC-density, I chose to bin those densities with lowly-occupied values as border of bins. Two binning schemes were proposed, and log-likelihood under these binning schemes were evaluated to be -4618 and -4389.

7 Question 6: Baum-Welch Learning Algorithm

More detailed notes for deduction and implementation can be found 9.1.2

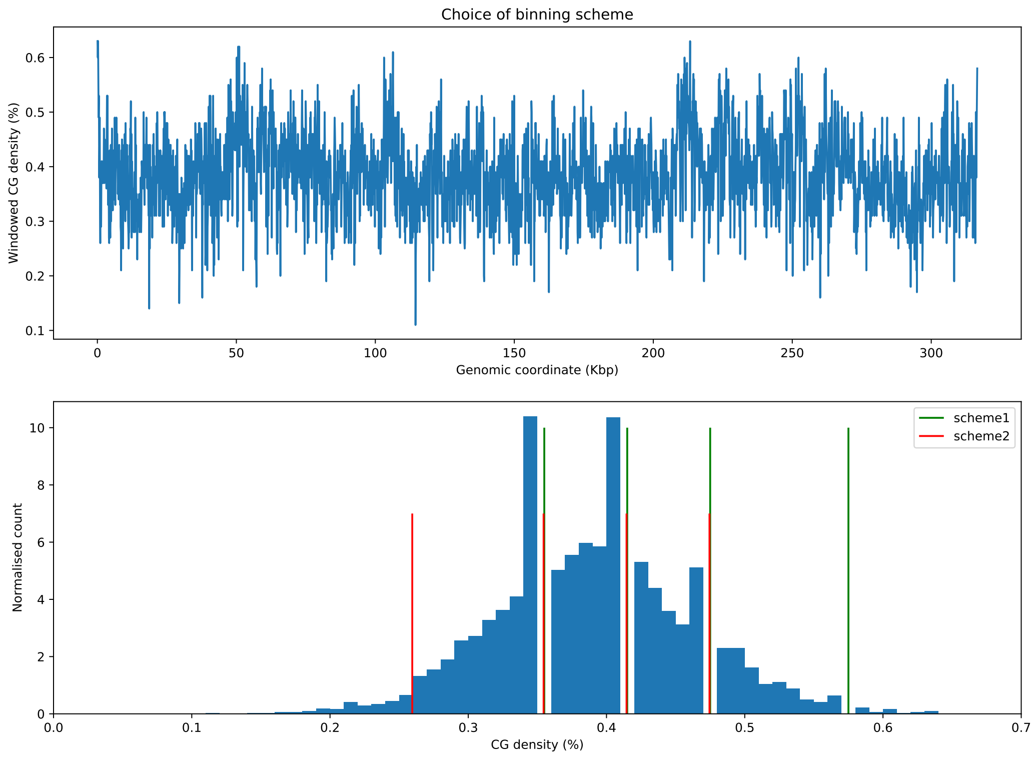

Figure 2: Overview of CG-density in Chromosome III of *S. cerevisiae*

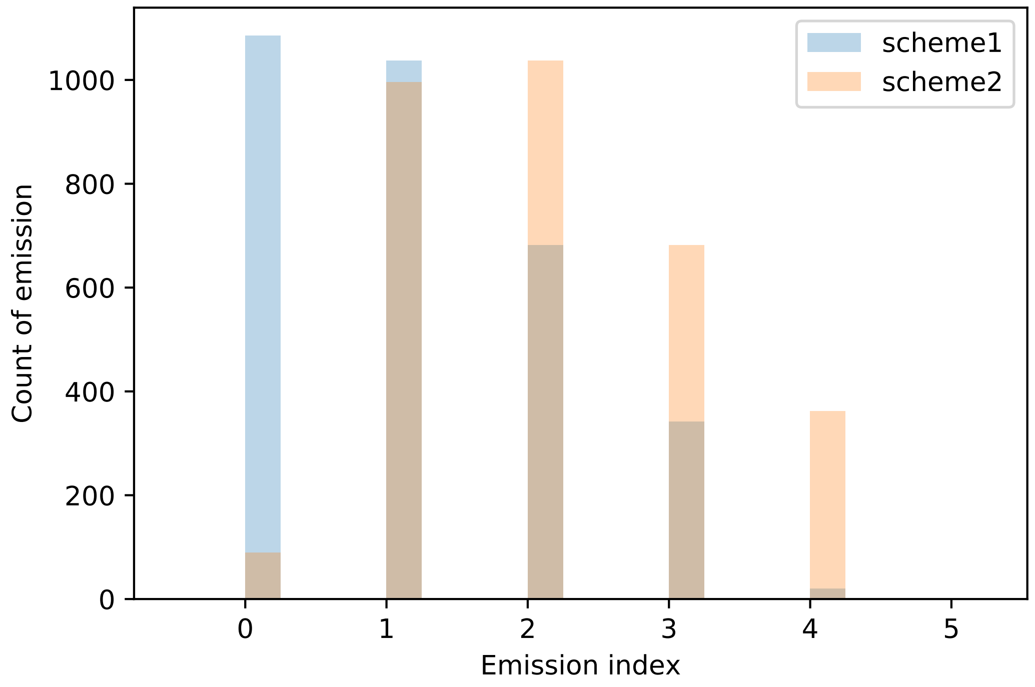

Figure 3: Distribution of states in encoded GC-density sequence

Baum-Welch algorithm is an expectectation-maximization routine that iterates between: 1. Expectation: Finding the form of log-likelihood of the observation using current estimate of model parameter 2. Maximization: Update the model parameter with the parameter set that maximises this log-likelihood function.

Here we will set the algorithm to terminate once the loglikelihood does not change significantly anymore.

$$L = \log P(Y_0^N \mid model)\Delta L < threshold$$

Although BW is an EM algorithm, its E-M steps are not completely separated from each other. In practice, BW provides a MLE for the model parameter, which is directly dervied from the current estimate of model parameter, which can be iteratively substituted until convergence. Recall the model is defined by $(\mu(s_i), A_{ij}, b_{s_i}(v_n))$, the BW-MLE takes the form:

$$\begin{aligned} \hat{\mu}(s_i) &= \gamma_0(s_i) \\ \hat{A}_{ij} &= \frac{E(n_{ij} \mid Y)}{E(n_i \mid Y)} \\ \hat{b}_{s_i}(Y = v_k) &= \frac{\sum_{\{n:\ Y_n = v_k\}} \gamma_n(s_i)}{\sum_n \gamma_n(s_i)} \end{aligned}$$

where n_{ij} is the random variable denoting the number of $i \to j$ transition in a hidden state chain, with their conitional expectation being:

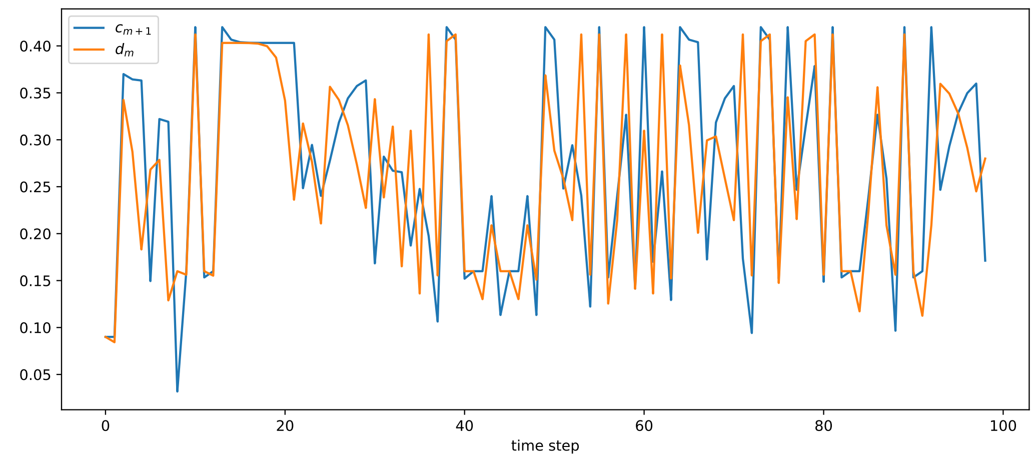

Figure 4: Quality-controlling forward-backward algorithim:d_m is well approximated by c_{m+1}

$$E(n_{ij} \mid Y) = \sum_{m=0}^{N-1} P(X_m = s_i, X_{m+1} = s_j \mid Y_0^N)$$
$$= \sum_{m=0}^{N-1} \xi_m(s_i, s_j)$$

And $\gamma_m(s_i)$ is essentially $\xi_m(s_i, s_j)$ marginalised over the s_j, which reads:

$$\gamma_m(s_i) = \sum_j \xi_m(s_i, s_j)$$

7.1 Result (See 9.2.5)

Run log

Filename:"Q6.py"

I first verified that the forward and backward variables scaled appropriately (figure 4). After that, BW-procedure are repeated until the difference in log-likelihood between updates is smaller than 0.005. Supericially, binning scheme 1 seems to admits a model that better explains the data. The log-likelihood under the new model are -3926 and -4149 respectively (figure 5).

8 Question 7: Viterbi algorithm for finding maximally likely sequence $\hat{X}_0^N$

More detailed notes for deduction and implementation can be found 9.1.3

Although Baum-Welch algorithm can be applied to refine the model iteratively, it is impossible to infer the maximally likely hidden-state sequence using the temporary variables invovled, because the interwined dependency precluded simple combination of individual maximally likely hidden states. Luckily Viterbi algorithm offers an alternative MLE to infer hidden sequence given emission sequence, namely finding

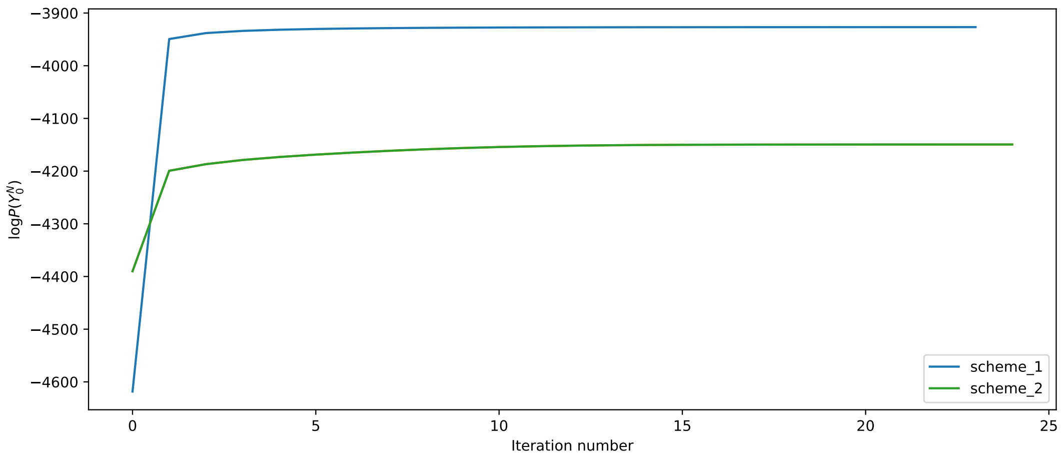

Figure 5: Quality-control for BW algorithm: Saturation of log-likelihood

$$\hat{X_0^N} = \underset{X_0^N}{\text{argmax}}[P(X_0^N \mid Y_0^N, \{\mu(s_i), A_{ij}, b_{s_i}(v_n)\})]$$

But notice $P(Y_0^N \mid \{\mu(s_i), A_{ij}, b_{s_i}(v_n)\})$ is a constant, which means we only need to focus on

$$\hat{X_0^N} = \underset{X_0^N}{\text{argmax}}[P(X_0^N, Y_0^N)]$$

8.1 Result (See 9.2.6)

Run log Filename:“Q7.py”

It can be seen that neither of the two models captures the local fluctuation of the CG density. Furthermore, the scheme 2 tends to predict longer contiguous segments than scheme 1, which will be a good feature if we are only interesting in that granularity. Visually speaking, the model is trying to segment the genome into CG-rich and CG-depleted regions. To evaluate its relevance to CpG island, we compared this annotation to that of UCSC Genome browser (figure 7). The called hidden states seem to correspond to gene regions, which means the model can be modified to predict the gene position.

The model can be improved in various ways, via chaning the emission model, the state space, and the learning algorithm:

1. The emission model. It is conceivable that CG density is a continous rather than discrete variable. Although a discrete binning scheme is straight-forward to implement and to test, a continous emission model based on gaussian distribution should improve the model’s likelihood function to be more robust to noise.

2. The state space: The size of the state space may be increased to use more latent variables to improve the flexibility of the model. However such practice must be carefully quality-controlled to avoid overfitting.

3. The learning algorihtm: Currently the model is refined using Baum-Welch algorithm, which may be replaced by Viterbi learning. Furthermore, with the assumption that state space corresponds to existence/absence of genes, this information may be further integrated to guide the fitting of parameters, using general purpose algorithm like gradient-descent. However, introducing a model on the hidden state will increase the complexity of the likelihood function and hence the algorithm, which means maybe leave such information on the emission layer could be a simpler alternative.

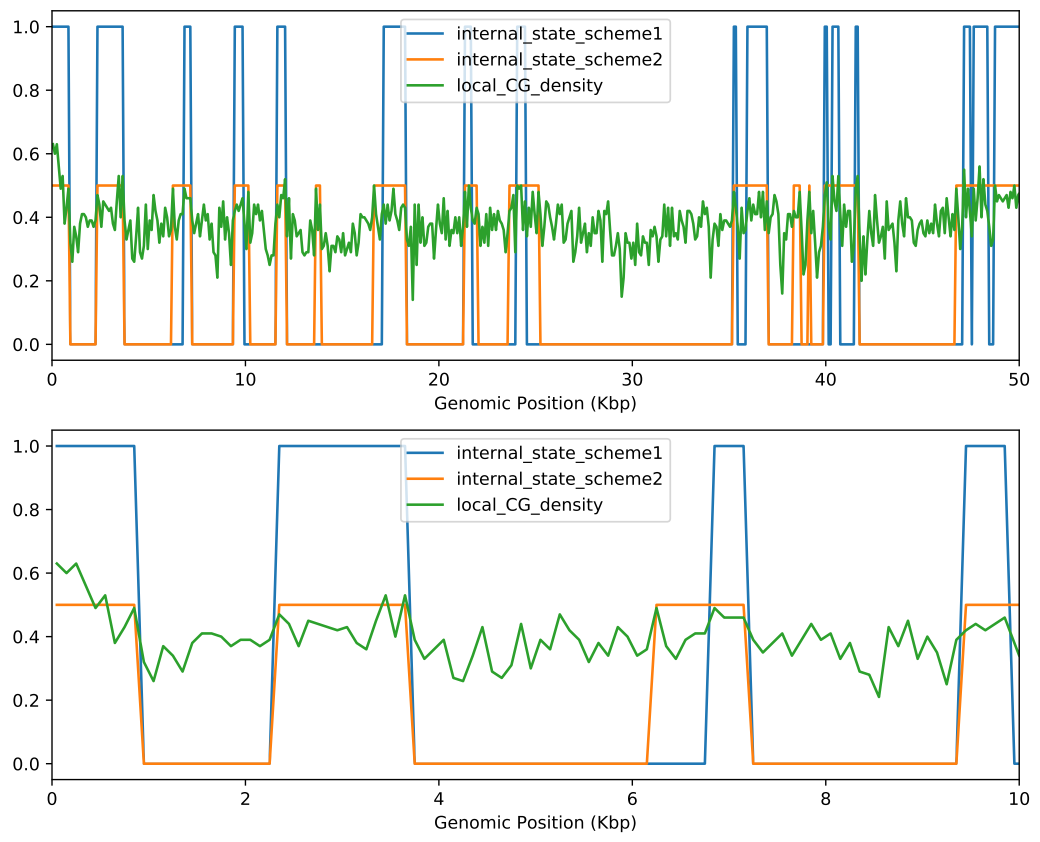

Figure 6: Maximaly likely sequences as inferred from the models refined with BW

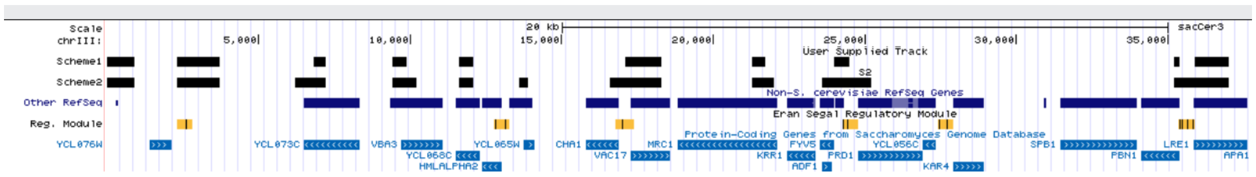

Figure 7: 0-50Kbp of ChrII viewed on UCSG genome browser

9 Appendix

9.1 Deductions

```
jupyter nbconvert --to markdown forward_algo.ipynb --output Q5
jupyter nbconvert --to markdown Q6-Backward_algo.ipynb --output Q6
jupyter nbconvert --to markdown Q7-Viterbi_algorithm.ipynb --output Q7
```

9.1.1 Question 5:Forward algorithm

The idea of forward algorithm is to exploit the hierarchy within the collection of all possible hidden chains using dynamic programming, so as to reduce the time complexity from $O(J^N)$ to $O(NJ^2)$, with J the cardinality of hidden states set, and N the length of the chain.

9.1.1.1 Naive forward algorithm

The forward algorithm aims to calculate the likelihood of an observation chain Y_0^N given the model $(\mu(s_i), A_{ij}, b_{s_i}(Y))$ as:

$$P(Y_0^N) = \sum_i P(Y_0^N, x_N = s_i)$$

Let

$$\alpha_N(s_i) = P(Y_0^N, x_N = s_i)$$
$$P(Y_0^N) = \sum_i \alpha_N(s_i)$$

All that's left is to construct $\alpha_N(s_i)$ recursively from $\alpha_0(s_i)$, since

$$\begin{aligned}\alpha_0(s_i) &= P(Y_0^0, x_0 = s_i)\\ &= P(Y_0|x_0 = s_i)P(x_0 = s_i)\\ &= b_i(Y_0)\mu(s_i)\end{aligned}$$

For convenienve we also define

$$\nu(Y_0) = \sum_i b_i(Y_0)\mu(s_i) = P(Y_0)$$

The recursion formula can be obtained using the Markov assumption (conditional independency) to be:

$$\alpha_{n+1}(s_j) = b_j(Y_{n+1}) \sum_i [A_{ij}\alpha_n(s_i)]$$

9.1.1.2 Rescaled forward algorithm

The naive implementation, however, suffers from numeric underflow, because $A_{ij} < 1, b_{s_i}(Y) < 1$. To avoid that, the $\alpha_N(s_i)$ is normalised so that $\sum_i \hat{\alpha}_N(s_i) = 1$, which implies:

$$\begin{aligned}\hat{\alpha}_N(s_i) &= \frac{\alpha_N(s_i)}{\sum_i \alpha_N(s_i)} \\ &= \frac{\alpha_N(s_i)}{P(Y_0^N)}\end{aligned}$$

So that the recursion becomes:

$$\begin{aligned}\hat{\alpha}_{n+1}(s_j) &= \frac{\alpha_{n+1}(s_j)}{P(Y_0^{n+1})} \\ &= \frac{1}{P(Y_0^{n+1})} b_j(Y_{n+1}) \sum_i [A_{ij}\alpha_n(s_i)] \\ &= \frac{1}{P(Y_0^{n+1})} b_j(Y_{n+1}) \sum_i [A_{ij}\frac{\alpha_n(s_i)}{P(Y_0^n)}P(Y_0^n)] \\ &= \frac{P(Y_0^n)}{P(Y_0^{n+1})} b_j(Y_{n+1}) \sum_i [A_{ij}\hat{\alpha}_n(s_i)]\end{aligned}$$

To avoid storage of $P(Y_0^n)$ directly, define:

$$\begin{aligned}c_n &= \frac{P(Y_0^n)}{P(Y_0^{n-1})} \\ \hat{\alpha}_{n+1}(s_j) &= \frac{1}{c_{n+1}} b_j(Y_{n+1}) \sum_i [A_{ij}\hat{\alpha}_n(s_i)]\end{aligned}$$

In practice, we calculate the $c_{n+1}\hat{\alpha}_{n+1}(s_i)$ firstly as (See "`def _iter_cNalpha()`"):

$$c_{n+1}\hat{\alpha}_{n+1}(s_j) = b_j(Y_{n+1}) \sum_i [A_{ij}\hat{\alpha}_n(s_i)]$$

and then separate the $\hat{\alpha}_{n+1}(s_i)$ with:

$$\begin{aligned}c_{n+1} &= \sum_j c_{n+1}\hat{\alpha}_{n+1}(s_j) \\ \hat{\alpha}_{n+1}(s_j) &= \frac{c_{n+1}\hat{\alpha}_{n+1}(s_j)}{c_{n+1}}\end{aligned}$$

9.1.1.3 Initial condition

$$\begin{aligned}\hat{\alpha}_0(s_i) &= \frac{P(Y_0^0, x_0 = s_i)}{P(Y_0^0)} \\ &= \frac{b_i(Y_0)\mu(s_i)}{\nu(Y_0)} \\ c_0 &= P(Y_0^0) = \nu(Y_0) \\ c_0\hat{\alpha}_0(s_i) &= b_i(Y_0)\mu(s_i)\end{aligned}$$

In order to recover $P(Y_0^n)$, do:

$$
\begin{aligned}
P(Y_0^N) &= P(Y_0^0)\prod_{n=1}^{N}\frac{P(Y_0^n)}{P(Y_0^{n-1})} \\
&= c_0\prod_{n=1}^{N} c_n \\
&= \prod_{n=0}^{N} c_n \\
\ln P(Y_0^N) &= \sum_{n=0}^{N}\ln c_n
\end{aligned}
$$

9.1.2 Question 6: Backward algorithm and Baum-Welch MLE

Baum-Welch algorithm is an expectectation-maximization routine that iterates between: 1. Expectation: Finding the form of log-likelihood of the observation using current estimate of model parameter 2. Maximization: Update the model parameter with the parameter set that maximises this log-likelihood function.

Here we will set the algorithm to terminate once the loglikelihood does not change significantly anymore.

$$
L = \log P(Y_0^N \mid model)\Delta L < threshold
$$

Although BW is an EM algorithm, its E-M steps are not completely separated from each other. In practice, BW provides a MLE for the model parameter, which is directly dervied from the current estimate of model parameter, which can be iteratively substituted until convergence. Recall the model is defined by $(\mu(s_i), A_{ij}, b_{s_i}(v_n))$, the BW-MLE takes the form:

$$
\begin{aligned}
\hat{\mu}(s_i) &= \gamma_0(s_i) \\
\hat{A}_{ij} &= \frac{E(n_{ij} \mid Y)}{E(n_i \mid Y)} \\
\hat{b}_{s_i}(Y = v_k) &= \frac{\sum_{\{n:\ Y_n = v_k\}}\gamma_n(s_i)}{\sum_n \gamma_n(s_i)}
\end{aligned}
$$

where n_{ij} is the random variable denoting the number of $i \to j$ transition in a hidden state chain, with their conitional expectation being:

$$
\begin{aligned}
E(n_{ij} \mid Y) &= \sum_{m=0}^{N-1} P(X_m = s_i, X_{m+1} = s_j \mid Y_0^N) \\
&= \sum_{m=0}^{N-1}\xi_m(s_i, s_j)
\end{aligned}
$$

And $\gamma_m(s_i)$ is essentially $\xi_m(s_i, s_j)$ marginalised over the s_j, which reads:

$$
\gamma_m(s_i) = \sum_j \xi_m(s_i, s_j)
$$

In practice, $\xi_m(s_i, s_j)$ is obtained by combining forward and backward variables, $\alpha_m(s_i)$ and $\beta_m(s_j)$ respectively (deduction ignored):

$$\begin{aligned}\xi_m(s_i, s_j) &= \frac{\alpha_m(s_i) \cdot \beta_{m+1}(s_j) \cdot b_j(Y_{m+1}) \cdot A_{ij}}{\sum_i \alpha_m(s_i) \cdot \beta(s_i)} \\ &= \frac{\alpha_m(s_i) \cdot \beta_{m+1}(s_j) \cdot b_j(Y_{m+1}) \cdot A_{ij}}{P(Y_0^N)}\end{aligned}$$

With the foward and backward variables being:

$$\begin{aligned}\alpha_m(s_i) &= P(Y_0^m, x_m = s_i) \\ \beta_m(s_i) &= P(Y_{m+1}^N, x_m = s_i)\end{aligned}$$

Because recursion for $\alpha_m(s_i)$ is already introduced, here I focus on the recursion for $\beta_m(s_i)$

$$\beta_m(s_i) = \sum_j \beta_{m+1}(s_j)\ b_j(Y_{m+1})\ A_{ij}$$

initialised at $m = N$ with:

$$\beta_N(s_i) = 1$$

To avoid underflow, $\beta_m(s_i) = 1$ is normalised to $\sum_i \hat{\beta}_m(s_i) = 1$, which means

$$\begin{aligned}\hat{\beta}_m(s_i) &= \frac{\beta_m(s_i)}{\sum_i \beta_m(s_i)} \\ &= \frac{\beta_m(s_i)}{P(Y_{m+1}^N)}\end{aligned}$$

The recursion then becomes

$$P(Y_{m+1}^N)\hat{\beta}_m(s_i) = \sum_j P(Y_{m+2}^N)\hat{\beta}_{m+1}(s_j)b_j(Y_{m+1})A_{ij}\frac{P(Y_{m+1}^N)}{P(Y_{m+2}^N)}\hat{\beta}_m(s_i) = \sum_j \hat{\beta}_{m+1}(s_j)b_j(Y_{m+1})A_{ij}d_m\hat{\beta}_m(s_i) = \sum_j \hat{\beta}_{m+1}(s_j)b$$

So that we can take advantage of the stored c_{m+1} with:

$$c_{m+1}\hat{\beta}_m(s_i) = \sum_j \hat{\beta}_{m+1}(s_j)b_j(Y_{m+1})A_{ij}$$

Intialised with:

$$\hat{\beta}_N(s_i) = \frac{1}{\sum_i 1}$$

9.1.2.1 Implementing BW-MLE

Recall $\xi_m(s_i, s_j)$ is obtained by combining forward and backward variables, $\alpha_m(s_i)$ and $\beta_m(s_j)$ respectively. In practice, this is done in log-space:

$$\begin{aligned}\xi_m(s_i, s_j) &= \frac{\alpha_m(s_i) \cdot \beta_{m+1}(s_j) \cdot b_j(Y_{m+1}) \cdot A_{ij}}{\sum_i \alpha_m(s_i) \cdot \beta_m(s_i)} \\ \log \xi_m(s_i, s_j) &= \log \alpha_m(s_i) + \log \beta_{m+1}(s_j) + \log\left[b_j(Y_{m+1}) \cdot A_{ij}\right] - \log P(Y_0^N)\end{aligned}$$

$$\log \alpha_m(s_i) + \log \beta_{m+1}(s_j) + \log\left[b_j(Y_{m+1}) \cdot A_{ij}\right] - \log P(Y_0^N)$$

$$\hat{\mu}(s_i) = \gamma_0(s_i)$$
$$\hat{A}_{ij} = \frac{E(n_{ij} \mid Y)}{E(n_i \mid Y)}$$
$$\hat{b}_{s_i}(Y = v_k) = \frac{\sum_{\{n:\ Y_n = v_k\}} \gamma_n(s_i)}{\sum_n \gamma_n(s_i)}$$

where n_{ij} is the random variable denoting the number of $i \to j$ transition in a hidden state chain, with their conitional expectation being:

$$\begin{aligned} E(n_{ij} \mid Y) &= \sum_{m=0}^{N-1} P(X_m = s_i, X_{m+1} = s_j \mid Y_0^N) \\ &= \sum_{m=0}^{N-1} \xi_m(s_i, s_j) \end{aligned}$$

Which is futher maginalised to give

$$E(n_i \mid Y) = \sum_{j=1}^{J}[\sum_{m=0}^{N-1} \xi_m(s_i, s_j)]$$

And $\gamma_m(s_i)$ is essentially $\xi_m(s_i, s_j)$ marginalised over the s_j, which reads:

$$\gamma_m(s_i) = \sum_j \xi_m(s_i, s_j)$$

which means some computation can be saved by exploiting

$$\begin{aligned} E(n_i \mid Y) &= \sum_{j=1}^{J}[\sum_{m=0}^{N-1} \xi_m(s_i, s_j)] \\ &= \sum_{m=0}^{N-1}[\sum_{j=1}^{J} \xi_m(s_i, s_j)] \\ &= \sum_{m=0}^{N-1} \gamma_m(s_i) \end{aligned}$$

9.1.3 Question 7: Viterbi algorithm

Although Baum-Welch algorithm can be applied to refine the model iteratively, it is impossible to infer the maximally likely hidden-state sequence using the temporary variables invovled, because the interwined dependency precluded simple combination of individual maximally likely hidden states. Luckily Viterbi algorithm offers an alternative MLE to infer hidden sequence given emission sequence, namely finding:

$$\hat{X_0^N} = \underset{X_0^N}{\text{argmax}}[P(X_0^N \mid Y_0^N, \{\mu(s_i), A_{ij}, b_{s_i}(v_n)\})]$$

But notice $P(Y_0^N \mid \{\mu(s_i), A_{ij}, b_{s_i}(v_n)\})$ is a constant, which means we only need to focus on

$$\hat{X_0^N} = \underset{X_0^N}{\text{argmax}}[P(X_0^N, Y_0^N)]$$

Although it is hard to write down the direct formulation for a such sequence, helper variables δ and ψ (traceback varaiable) may be defined to simplify the case and to construct the recursion:

$$\begin{aligned}\delta_n(s_i) &= \max_{X_0^{n-1}} P(Y_0^n, X_0^n) \\ &= \max_{X_0^{n-1}} P(Y_0^n, X_0^{n-1}, X_n = s_i)\end{aligned}$$

Note for $n = N$ this is we can simply take another max to obtain the log-likelihood

$$\begin{aligned}\delta_N(s_i) &= \max_{X_0^{N-1}} P(Y_0^N, X_0^N) \\ \max_{X_N} \delta_N(s_i) &= \max_{X_N} \max_{X_0^{N-1}} P(Y_0^N, X_0^N) \\ \max_{X_N} \delta_N(s_i) &= \max_{X_0^N} P(Y_0^N, X_0^N)\end{aligned}$$

9.1.3.1 Recursion for δ_n (deduction omitted) :

Note

$$\begin{aligned}\delta_{n+1}(s_j) &= \max_{X_0^{n-1}, X_n} b_j(Y_{n+1}) A_{ij} \delta_n(s_i) \\ &= b_j(Y_{n+1}) \max_{X_n = s_i} A_{ij} \delta_n(s_i)\end{aligned}$$

9.1.3.2 Numerical underflow

In practice, we store $\phi_n(s_i) = \log \delta_n(s_i)$ to avoid numerical underflow. The recursion then becomes

$$\begin{aligned}\exp(\phi_{n+1}(s_j)) &= \max_{X_0^{n-1}, X_n} b_j(Y_{n+1}) A_{ij} \exp(\phi_n(s_i)) \\ \phi_{n+1}(s_j) &= \log[b_j(Y_{n+1})] + \max_{X_n = s_i} [\log[A_{ij}] + \phi_n(s_i)]\end{aligned}$$

9.1.3.3 Initialisation

At $n = 0$, we have:

$$\begin{aligned}\phi_0(s_i) &= \log[max_{\emptyset} P(Y_0^0, X_0^0)] \\ &= \log[P(Y_0, X_0 = s_i)] \\ &= \log[\mu(s_i) b_i(Y_0)]\end{aligned}$$

9.1.3.4 Traceback

While doing $\phi_n(s_i) \to \phi_{n+1}(s_j)$ will indicate the likelihood, we still need to find the maximally likely path using this information. This is done by storing the "argmax" result for all "max" operator consumed. For example, in:

$$\phi_{n+1}(s_j) = \log[b_j(Y_{n+1})] + \max_{X_n = s_i} [\log[A_{ij}] + \phi_n(s_i)]$$

The argmax will be stored as a traceback pointer $\psi_{n+1}(s_j) \to (X_n = s_i)$:

$$\psi_{n+1}(s_j) = \underset{X_n = s_i}{\operatorname{argmax}} [\log[A_{ij}] + \phi_n(s_i)]$$

We can then recover the MLE $\hat{X_0^N} = \{X_0^*, X_1^*, ..., X_N^*\}$, with the traceback recursion

$$X_{n-1}^* = \psi_n(X_n^*)$$

Initialised with

$$X_N^* = \underset{X_N=s_i}{\text{argmax}} \phi_N(s_i)$$

and can be rationalised by seeing its relation with maximum likelihood

$$\begin{aligned}\phi_N(X_N^*) &= \max_{X_N=s_i} \phi_N(s_i) \\ &= \max_{X_0^N} \log P(Y_0^N, X_0^N)\end{aligned}$$

9.2 Code

```
tar -cvzf GSA_fg368.tar.gz src/
```

9.2.1 Filename:“test___mle_chain2transition.py"

```
#### Question 2
####### MLE for a single internal state chain
# from util import *
import util
import matplotlib.pyplot as plt
import collections
import numpy as np
def mle_tmat_internalChain(chain, Hn = None,epsilon = 0):
    '''
    epsilon: an adjustable pesudo count
    '''
    Hn = max(chain) + 1
    trans = collections.Counter([tuple(chain[i:i+2]) for i in range(len(chain)-1)])

    mat = np.zeros((Hn,Hn))
    for k,v in trans.iteritems():
        mat[k]=float(v) + epsilon
    mat /= mat.sum(axis = 1,keepdims = 1)
    return mat

# import inspect
# print inspect.getsource(mle_tmat_internalChain)
h1 = util.Var_hmm.read_model(fname = 'Q3.hmm')
chain = util.read_chain(fname='test.echain',dtype=int)[0]
chain = list(chain[:20])
mle_tmat = mle_tmat_internalChain(chain)

print "Truncating chain to 20 elements:",chain
print 'MLE for transition matrix is:\n',mle_tmat
print 'TRUE transition matrix is:\n', h1.transition
print 'MAE of the estimator is:\n', abs(mle_tmat - h1.transition)
```

```
## [0.2  0.11 0.27 0.26 0.16]
## Reading chains from file 'test.echain' :
## [[1 1 0 1 1 0 1 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 1 1 0 1 1 0 0 0 0 1 1 1 1 0
##   1 1 1 0 0 1 1 1 0 0 1]]
## Truncating chain to 20 elements: [1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0]
## MLE for transition matrix is:
## [[0.42857143 0.57142857]
##  [0.41666667 0.58333333]]
## TRUE transition matrix is:
## [[0.8 0.2]
##  [0.1 0.9]]
## MAE of the estimator is:
## [[0.37142857 0.37142857]
##  [0.31666667 0.31666667]]
```

9.2.2 Filename:"test___sampling.py"

```
#### Question 3
import util
import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt

import numpy as np
h1 = util.Var_hmm(
    internal_list = [0,1],
    emission_list = [ (x) for x in [1,2,3,4,5]],
    transition = np.array([[0.8, 0.2],[0.1,0.9]]),
    emission_mat = np.array([[0.2, 0.5, 0.2, 0.1, 0],[ 0, 0.1, 0.4 ,0.4 , 0.1]])
)
n = 115
chain = h1.run_for(n,as_idx = 1)
emission_chain = h1.bulk_emit(chain, as_idx = False)

fig = plt.figure(figsize= [12,4])
plt.subplot(111)
plt.plot(chain,label = 'hidden_state')
plt.plot(emission_chain, label ='emission_state')
plt.legend(bbox_to_anchor=(1.19, 1.00))
plt.xlabel('time step')
plt.ylabel('Value')

#plt.show()

figname = __file__.rsplit('.',1)[0]+'.pdf'
print 'Saving figure to: %s' % figname
plt.savefig(figname)
#h1.bulk_emit([1,1,0,0,0],as_idx = 1)
```

9.2.3 Filename:"test___forward.py"

```
import util
from forward import *
import warnings

if __name__=='__main__':
    h1 = util.Var_hmm.read_model("Q3.hmm")
    echain = util.read_chain('test.echain',dtype = 'int')
    #print "Calculating likelihood for chain:",echain
    _ = h1.emission_likelihood(echain,debug=1)
```

9.2.4 Filename:“Q5_clean.py”

```
# coding: utf-8

# In[1]:

import util
from forward import *
from util_genome import *
import collections, functools
import numpy as np

# In[2]:

# In[48]:

# In[3]:

if __name__ == '__main__':
    seqs = read_fasta('Saccharomyces_cerevisiae.R64-1-1.dna.chromosome.III.fa')
    seq = seqs[0][1]
    gr,chunks = to_chunks(seq,100,100,debug = 1)
    CG_densi = [CG_density(x) for x in chunks]
    print len(seq)

# In[10]:

import matplotlib.pyplot as plt

plt.figure(figsize = [14,10])
plt.subplot(211)
plt.plot(np.array(gr)/1000, CG_densi)
plt.xlabel('Genomic coordinate (Kbp)')
plt.ylabel('Windowed CG density (%)')

plt.title('Choice of binning scheme')
plt.subplot(212)
# plt.figure()
#which()
ct,edges,_ = plt.hist(CG_densi,bins = np.linspace(0,1,101),normed=1)
mids = (edges[:-1] + edges[1:])/2
egs = mids[np.where( ct == 0 )]
egs = egs[(egs > 0.3) & (egs < 0.6)]
scheme1 = egs.copy()
scheme2 = np.array([0.26] + egs[:-1].tolist() )
plt.vlines(scheme1, 0,10,'g',label = 'scheme1')
plt.vlines(scheme2 - 0.0005,0,7,'r',label = 'scheme2')
plt.ylabel('Normalised count')
plt.xlabel('CG density (%)')
plt.xlim(0,0.7)
plt.legend()
print 'Scheme 1:'
print "The border of bins are:", scheme1

print 'Scheme 2:'
print "The border of bins are:", scheme2

figname = 'Q5_fig1.pdf'
print "Saving figure to: %s"%figname
plt.savefig(figname)

# In[50]:

plt.figure(figsize = [6,4])
coded_CG_densi = classify(CG_densi,**construct_LR(scheme1))
_ = plt.hist(coded_CG_densi,alpha = 0.3, bins = np.arange(-0.5,5.5,0.25),label = 'scheme1')
coded_CG_densi = classify(CG_densi,**construct_LR(scheme2))
_ = plt.hist(coded_CG_densi,alpha = 0.3,bins = np.arange(-0.5,5.5,0.25),label ='scheme2')
plt.legend()
plt.ylabel('Count of emission')
plt.xlabel('Emission index')

figname = 'Q5_fig2.pdf'
print "Saving figure to: %s"%figname
plt.savefig(figname)

# In[9]:

mdl = util.Var_hmm.read_model(fname = 'Q3.hmm',
           emission_list = [str(x) for x in range(1,6)])
print mdl

lst = []
print '\nScheme 1'
coded_CG_densi = classify(CG_densi,**construct_LR(scheme1))
mdl.emission_likelihood([coded_CG_densi],debug = 1)
lst.append(coded_CG_densi)

print '\nScheme 2'
coded_CG_densi = classify(CG_densi,**construct_LR(scheme2))
_ = mdl.emission_likelihood([coded_CG_densi],debug = 1)
lst.append(coded_CG_densi)

print >>open('Q3.echain','w'),util.mat2str(np.vstack(lst))
#mdl.unicode()
```

9.2.5 Filename:“Q6.py”

```
########## Functionality test for BW-algo
##### Although there seems to be numeric instability, the c_(m+1) = d_m  is visually observable

import util
from forward import *
from bulk import chains
import matplotlib.pyplot as plt
h1 = util.Var_hmm.read_model(fname ='Q3.hmm')
chs = chains(fname = 'Q3.echain',model = h1)
i = 0
plt.figure(figsize = [12,8])
#for N in np.arange(2,100,20).astype(int):
N = 100
if 1:
    i += 1
    echain = chs.echains[0][:N]
    h1.reset()
    cs = [h1._forward(E,return_c=1) for E in echain ]
    cs = np.round(cs[:],decimals=5)
    print 'shape',cs.shape
    print 'c_m:',cs[:30]
    cs = cs[1:]
    print cs.size

#    print np.log(cs)
    h1.reset()
    ds = [h1._backward(E,return_d=1) for E in echain[::-1] ]
    ds =  np.round(ds[::-1],decimals=5)
    print 'd_m',ds[:30]
    ds = ds[:-1]
#     print(zip(cs,ds)[:10])
    C = np.corrcoef(cs,ds)
    print C[0,1]
    plt.figure(figsize = [12,5])
    plt.subplot(1,1,i)
    clab = '$c_{m+1}$'
    dlab = '$d_{m}$'
    plt.plot(cs,label = clab)
    plt.plot(ds,label = dlab)
#     plt.ylabel()
    plt.xlabel('time step')
    plt.legend()
figname = 'Q6_FBQC.pdf'
print "Saving figure to: %s"%figname
plt.savefig(figname)

###########################


import copy
mdl = util.Var_hmm.read_model(fname = 'Q3.hmm',
       emission_list = [str(x) for x in range(1,6)])

plt.figure(figsize = [12,5])
print "Scheme 1"
chs = chains(fname = 'Q3.echain',model = copy.copy(mdl))
chs.single_chain_BW(cidx = 0)
chs.model.save_model('Q6_scheme1.hmm')
plt.plot(chs.logLs[0],label = 'scheme_1')

print "Scheme 2"
chs = chains(fname = 'Q3.echain',model = copy.copy(mdl))
chs.single_chain_BW(cidx = 1)
chs.model.save_model('Q6_scheme2.hmm')
plt.plot(chs.logLs[0])
plt.plot(chs.logLs[0],label = 'scheme_2')

plt.xlabel('Iteration number')
plt.ylabel('$\log{P(Y_0^N)}$')
plt.legend()

figname = 'Q6_BWQC.pdf'
print "Saving figure to: %s"%figname
plt.savefig(figname)
```

9.2.6 Filename:“Q7.py”

```
import util, util_genome
from forward import *
import matplotlib.pyplot as plt
#%matplotlib inline

if __name__ == '__main__':
    seqs = util_genome.read_fasta('Saccharomyces_cerevisiae.R64-1-1.dna.chromosome.III.fa')
    gr,chunks = util_genome.to_chunks(seqs[0][1],100,100)
    CG_densi = [util_genome.CG_density(x) for x in chunks]

if __name__=='__main__':
    print '\nViterbi MLE for scheme 1'
#     data = {'Q6_scheme1.hmm':None,'Q6_scheme2.hmm':None}
    h1 = util.Var_hmm.read_model(fname ='Q6_scheme1.hmm')
    echain = util.read_chain('Q3.echain',dtype=int)[0]
    mle_seq1 = h1.MLE_viterbi(echain,debug = 1)

    print '\nViterbi MLE for scheme 2'
    h2 = util.Var_hmm.read_model(fname ='Q6_scheme2.hmm')
    echain = util.read_chain('Q3.echain',dtype=int)[1]
    mle_seq2 = h2.MLE_viterbi(echain,debug = 1)
    gr = np.array(gr)/1000.

    plt.figure(figsize = [10,8])
    plt.subplot(211)
#     plt.figure(figsize = [10,4])
```

```
    plt.plot(gr,mle_seq1,  label = 'internal_state_scheme1')
    plt.plot(gr,mle_seq2/2.,  label = 'internal_state_scheme2')
    plt.plot(gr,CG_densi,label = 'local_CG_density')
    plt.xlim(0,5E1)
    plt.xlabel('Genomic Position (Kbp)')
    plt.legend()
#    plt.savefig('Q7_50K.pdf')

#     plt.figure(figsize = [10,4])
    plt.subplot(212)
    plt.plot(gr,mle_seq1,  label = 'internal_state_scheme1')
    plt.plot(gr,mle_seq2/2.,  label = 'internal_state_scheme2')
    plt.plot(gr,CG_densi,label = 'local_CG_density')
    plt.xlim(0,10)
    plt.xlabel('Genomic Position (Kbp)')
    plt.legend()
    plt.savefig('Q7.pdf')
#     plt.savefig('Q7')
```