Algorithms and Implementation details

October 30, 2014

1 TICA

Literature: G. Pérez-Hernández, F. Paul, T. Giorgino, G. De Fabritiis and F. Noé: Identification of slow molecular order parameters for Markov model construction Citation: J. Chem. Phys. 139, 015102 (2013); doi: 10.1063/1.4811489

We consider a d-dimensional vector of input data, called $\mathbf{r}(t) = (r_i(t))_{i=1,\ldots,D}$. Here, t is an integer from $\{1...N\}$ denoting the time step. We assume that the data is mean-free, i.e. from a general input vector $\tilde{\mathbf{r}}(t)$, we first obtain:

$$\mathbf{r}(t) = \tilde{\mathbf{r}}(t) - \langle \tilde{\mathbf{r}}(t) \rangle_t$$

where $\langle \tilde{\mathbf{r}}(t) \rangle_t$ is the data mean.

We first compute the covariance matrices from the data:

$$c_{ij}(\tau) = \langle r_i(t)\, r_j(t+\tau) \rangle_t$$

where τ is the lag time. We will need two matrices for TICA, for the choices $\tau = 0$ and another positive value of τ. $\langle \cdot \rangle_t$ denotes the time average. We can evaluate it as follows:

$$c_{ij}(\tau) = \frac{1}{N-\tau-1} \sum_{t=1}^{N-\tau} r_i(t)\, r_j(t+\tau).$$

It is easy to verify that $C(0)$ is a symmetric matrix. For algebraic reasons we will need that $C(\tau)$ is also symmetric, which is not automatically guaranteed. Therefore we enforce symmetry from a data-computed matrix $C_d(\tau)$:

$$C(\tau) = \frac{1}{2} \left(C_d(\tau) + C_d^\top(\tau) \right).$$

Now we solve the generalized eigenvalue problem:

$$C(\tau)\, U = C(0)\, U\, \Lambda$$

where U is the eigenvector-matrix containing the independent components (ICs) in the columns and Λ is a diagonal eigenvalue matrix. This problem can be solved by an appropriate generalized eigenvalue solver (directly), or in two steps. The two step procedure is called AMUSE algorithm and works as follows:

1. Solve the simple PCA Eigenvalue problem $\mathbf{C}(0)\, \mathbf{W} = \mathbf{W}\, \mathbf{\Sigma}$, where $\mathbf{W}$ is the eigenvector matrix with principal components and $\mathbf{\Sigma}$ are their variances (diagonal Eigenvalue matrix).
2. Transform the mean-free data $\mathbf{r}(t)$ onto principal components $\mathbf{y}(t) = \mathbf{W}\, \mathbf{r}(t)$.
3. Normalize the principal components: $\mathbf{y}'(t) = \mathbf{\Sigma}^{-1} \mathbf{y}(t)$
4. Compute the symmetrized time-lagged covariance matrix of the normalized PCs: $\mathbf{C}^y_{sym}(\tau) = \frac{1}{2} \left[\mathbf{C}^y(\tau) + (\mathbf{C}^y(\tau))^\top \right]$
5. Perform an eigenvalue decomposition of $\mathbf{C}^y_{sym}(\tau)$ to obtain the eigenvector matrix $\mathbf{V}$ and project the trajectory onto the dominant eigenvectors to obtain $\mathbf{z}(t)$.

In summary, we can write the transformation equation in three linear transforms:

$$\mathbf{z}^\top(t) = \mathbf{r}^\top(t)\mathbf{U} = \mathbf{r}^\top(t)\mathbf{W}\mathbf{\Sigma}^{-1}\mathbf{V}.$$

TICA will be used as a dimension reduction technique. Only the dominant TICA components will be used to go to the next step. In order to reduce the dimension, use only a few columns of $\mathbf{U}$.

Implementation notes:

- For large data sets, the construction of the covariance matrices is a very most time-consuming procedure. Think about how that can be done efficiently. In particular, consider the case that not the entire data set can be kepts in memory.
- How many dimensions can we deal with before the eigenvalue problem(s) become too large to solve?
- Eigenvalue solvers do generally not guarantee to provide the eigenvalues in a particular order. If you want to use TICA as a dimension reduction technique, you might have to reorder the eigenvalues and eigenvectors yourself, such that the dominant eigenvalues come first.
- Check every step of what you're doing. Is the data really mean-free / really normalized after the appropriate steps? Does the transformation make sense? How does the transformed data look like?
- Think of a good test case.

2 k-Means

k-Means is perhaps the famous out of many clustering algorithms. Clustering algorithms are used to classify or discretize data. Note the two different purposes:

1. Classification: We try to group data into *a few* classes that should be as distinct as possible. The purpose of the algorithm is to distinguish data points in such a way that it is easy to tell data points in different classes apart.
2. Discretization: We don't mind if there are many clusters. We just want to cluster finely enough such that clearly distinct data ends up in different clusters. This is usually just a data processing step in a larger pipeline.

Here we will use clustering for discretization. Therefore we don't have to worry too much that the the number of clusters, k, is chosen correctly. It just has to be big enough.

Consider that we have input data in the form $\mathbf{x}(t) = (x_i(t))_{i=1,...,d}$. Consider the following:

- We need to define a distance metric in order to measure distances. Usually we will use the normal Euclidean metric:
$$d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_2$$
- There are two results of the clustering. Result one are the cluster centers:
$$\mathbf{Y} = \{\mathbf{y}_1, ..., \mathbf{y}_k\}$$
Result two is the assignment of the data:
$$s(t) = (s_1, ..., s_N)$$
where each data point is assigned to one of the cluster centers. This is done by a Voronoi partition, i.e. each data point is assigned to the nearest center in the above distance metric.

The k-Means algorithm iterates the following steps:

- Initialization: Pick k input data points at random and set them as initial cluster centers, $\mathbf{Y}$.
- Assigment step:
  - For each data point, find the nearest cluster center and assign the data point to it:

$$s(t) = \arg\min_s d(\mathbf{y}_s, \mathbf{x}(t))$$

  - Note that this assignment step creates a Voronoi partition!
- Update step:
  - For each cluster center, update its position by setting it to the mean of the data assigned to it:

$$\mathbf{y}_s = \frac{1}{n_s} \sum_{i \in S} \mathbf{x}_i$$

    where S is the set of data points assigned to cluster s.

Implementation notes:

- When do you consider the algorithm to be converged?
- The maximum number of iterations is until convergence. But since many iterations might be needed to convergence and in principle we don't need the algorithm to converge if we want it to just do discretization (as opposed to classficiation), we can also have the number of iterations as an input parameter
- Think about what happens if the entire set of input data cannot be kept in memory. How do we need to design the algorithm in order to deal with that?

3 Hidden Markov Model

Estimation of a Hidden Markov model consists of estimating two quantities:

- The hidden transition matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ which governs the switching dynamics between n hidden states
- The output matrix $\mathbf{B} \in \mathbb{R}^{n \times m}$ which governs the translation of the n hidden system states to m observable system states.

The determination of n is a difficult problem of statistical inference and we will ignore it for now. Instead we will just set n to a value that we know works well for the data.

We will use the EM (Expectation-Maximization) Algorithm to do the estimation. EM is actually a class of algorithms and here we will use the so-called Baum-Welch implementation of it. The expectation step consists of running two algorithms, the forward and backward algorithm, and will provide us with a model of the hidden trajectory, i.e. statistics that contain our belief at which hidden state the system is at time t given the evidence that we have so far. The maximization step will then produce an estimate of the quantities $\mathbf{A}$ and $\mathbf{B}$.

3.1 Expectation step - forward algorithm

Let $\alpha_t(i)$ be the forward variable, defined as

$$\alpha_t(i) = \mathbb{P}(o_1, o_2, ..., o_t, s_t = S_i \mid \Lambda)$$

We solve for $\alpha_t(i)$ inductively as follows:

1. Initialization:
$$a_1(i) = \pi_i b_i(o_1)$$
for all i

2. Induction:
$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{n} \alpha_t(i)\, a_{ij}\right] b_j(o_{t+1})$$

Implementation notes:

- If you implement the algorithm exactly as above, you will run into underflow problems, because the α_t values will become smaller an smaller. Two procedures are common to avoid this problem: (1) rescaling of α_t values such that they sum to 1 for every t. If you choose this option, you have to remember the scaling factors in order to evalues the likelihood (below) correctly. (2) work in the log-space $\ln \alpha_t$. If you choose this option you have to adapt all calculations with α_t.

3.2 Expectation step - Likelihood

The likelihood can be computed from the α-Variables:

$$\mathbb{P}(O \mid \Lambda) = \sum_{i=1}^{N} \alpha_N(i)$$

Implementation notes:

- You have to adapt the calculation of the likelihood if you have rescaled α_t or worked in the log-space.

3.3 Expectation step - Backward algorithm

Definition:
$$\beta_t(i) = \mathbb{P}(o_{t+1}, o_{t+2}, ..., o_N \mid s_t = S_i, \Lambda)$$
We can solve for $\beta_t(i)$ inductively as follows:

1. Initialization:
$$\beta_N(i) = 1$$
for all i

2. Induction:
$$\beta_t(i) = \sum_{j=1}^{n} a_{ij} b_j(o_{t+1)} \beta_{t+1}(j)$$

3.4 The hidden trajectory

You might be interested at which hidden state the system is at a given time t. This can now be computed. We define
$$\gamma_t(i) = \mathbb{P}(s_t = S_i \mid O, \Lambda)$$
the probability that the system is at hidden state i at time t. It is given by:

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{N} \alpha_t(j)\beta_t(j)}$$

You can see that the normalization is not an issue in this equation. $\gamma_t(i)$ will be a normalized probability. If you want you can not compute the path of maximum probability, either by a simple arg-max or by the Viterbi algorithm. This is optional - do it if you have time and interest.

3.5 Maximization step

Now we are in a position to compute the maximum-likelihood values of $\mathbf{A}$ and $\mathbf{B}$. We need an intermediate quantity, the expected number of transitions from state i to j at time t:

$$\xi_t(i,j) = \frac{\alpha_t(i)\, a_{ij}\, b_j(o_{t+1})\, \beta_{t+1}(j)}{\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_t(i)\, a_{ij}\, b_j(o_{t+1})\, \beta_{t+1}(j)}.$$

From this we can also compute the probability to be in state i:

$$\gamma_t(i) = \sum_{j=1}^{n} \xi_t(i,j)$$

Now we compute the transition matrix:

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{N-1} \xi_t(i,j)}{\sum_{t=1}^{N-1} \gamma_t(i)}$$

As a small modification to the normal Baum-Welch algorithm we assume that we have a stationary process, i.e. that our starting probability is identical to the stationary probability of transition matrix $\mathbf{A}$. We compute it as stationary eigenvector of $\mathbf{A}$:

$$\boldsymbol{\pi}^\top = \boldsymbol{\pi}^\top \mathbf{A}.$$

Finally we compute the output probability matrix as:

$$\hat{b}_{ij} = \frac{\sum_{t=1, o_t=j}^{N} \gamma_t(i)}{\sum_{t=1}^{N} \gamma_t(i)}.$$

3.6 Iteration

In each iteration, the Likelihood $\mathbb{P}(O \mid \Lambda)$ should increase. Iterate the EM-steps until convergence

Implementation notes:

- What is the most time-consuming part of the algorithm? How do we determine this technically?
- How can we speed the algorithm up?
- Where are the numerical bottlenecks, i.e. where do you expect underflows, overflows, cancelleation, etc.?
- What is a good convergence criterion?