

Approaches to the Network-Optimal Partition Problem for fMRI Data

Benjamin Yackley, Terran Lane

Abstract

Although much MRI research has been performed using existing atlases of brain regions, it is possible that these regions, which are anatomically determined, are not the truly optimal way to divide brain activity into parts. The goal of this research is to discover a way to take existing fMRI data and use it as a basis for a clustering method that would divide the brain into functional partitions independent of anatomy. Since Bayesian networks over anatomical regions have been shown to be informative models of brain activity, this paper describes attempts to solve a mathematical problem which is an abstraction of the above goal: to partition a group of variables (the voxels of the MRI data) into clusters such that the optimal Bayesian network given the clustering is as “good” as possible.

1 Introduction

Functional magnetic-resonance imaging, or fMRI, is a valuable tool in neuroscience; it allows researchers to discover patterns of activity within regions of the brain over time. With current technology, we can divide the brain into approximately 30,000 voxels — rectangular regions in 3-dimensional space — and probe their activity over the course of a task. The temporal resolution of fMRI is poor; one frame of the whole brain is captured every 1–2 seconds, but its spatial resolution is very good, and activity can be localized to relatively small parts of the brain; the size of a single voxel is only a few millimeters on each side. This is, of course, much larger than a single neuron, but still good enough to detect interesting patterns. This is in contrast with EEG techniques, which have very good temporal resolution but very poor spatial resolution relative to fMRI.

fMRI studies can be used to localize activity within specific brain regions. This can be used not only to pinpoint certain areas of the brain as responsible for kinds of behavior, but also to analyze patterns of activity between regions. For instance, breaking neural activity apart into regions and finding an optimal Bayesian network over these regions given the data can be used to show differences in the activity patterns of healthy and demented patients[1]. However, these regions are determined anatomically — by the positions of physical wrin-

kles and folds in the brain itself¹. It's possible that these regions are not optimal, and that in fact the brain can be divided more intelligently into regions by considering the function of groups of voxels and separating these optimally so that, when brain activity is partitioned using these new regions, the already-found patterns (the Bayesian networks, for instance) would stand out more clearly and perhaps even new patterns could be found.

2 The Main Problem(s)

The difficult part of this problem is that it's composed of two parts that depend heavily on one another, neither of which produces an optimal solution unless the other is already known. One of these parts is the search for an optimal network given the clustering, which is a straightforward Bayes Net search, assuming the clusters are known. The other is to search for an optimal clustering that will produce the best network possible, and this is far less straightforward. We could add the condition that the network structure over the clusters is known beforehand (a structure which somehow exists independent of any data; think of it as a network with "hollow" nodes) and then search for a way to place the variables into the clusters so that the resulting network has as high a score as possible; this is the way pursued in section 4.

It also seems possible to combine these two using a kind of alternating-projections method; first, the variables would be clustered at random and then an optimal network found over those clusters. Then, that network would be used as the basis for a re-clustering, and then the new clusters used to find a network, and so on. Although each step would seem to improve one side of this dual problem, it is not known if this would actually converge on the "true" solution or whether it would get stuck in a local maximum.

3 Searching for Partitions using MCMC

The first approach we tried was to use a Markov Chain Monte Carlo technique (specifically, the Metropolis algorithm[5]) to explore the space of all possible partitionings. Ideally, this would be structured so that the score of a given partitioning would reflect its suitability as a basis for finding Bayesian networks over that data. One way to think about this score is to use the log-likelihood score of a "typical" network found given that partitioning and the fixed set of data, but this is not the only possibility.

3.1 The MCMC framework

To serve this purpose, we created a set of Java classes (as described in Section 5) to represent a generalized MCMC process. In our particular case, there were three potential operations that could be performed on a given set of partitions.

¹ One common way to do this is called the Talairach atlas; see for instance the Talairach book[6] for details.

The first and most straightforward was to select two partitions and merge them together. The second was to move a single voxel from one partition into another. Finally, the third would split a partition in two by assigning each voxel in the original partition to either its original partition or a newly created one with equal probability. Given these operations, then, the goal was to find a scoring function that would allow us to traverse the space of all partitions in search for its mode. Because the motivation is to find a partitioning that would result in the “best” possible Bayesian Network over the partitions, it seemed natural to use the likelihood score of the optimal network over the partitions as the score of the partitioning itself (see section 5.2 for details). However, comparing the scores of two networks with different numbers of nodes or based on different data (as would arise from any partitioning with the same number of partitions but a different assignment) is not very well-defined. Nevertheless, we decided to use a similar method to score partitions; instead of finding the best possible network over a given partitioning of voxels, which could potentially take a long time, we instead sampled a number of networks at random and used the maximum score found as the partition’s score.

3.2 Chow-Liu Trees

Sampling networks at random had two major flaws. One was that it was slow; scoring a single network takes a few seconds, so sampling even 100 networks would require several minutes of computation for each step in an MCMC process with potentially thousands of steps, and 100 is a small number of samples to take considering that the number of possible networks grows exponentially with the number of nodes. This ties in with the other reason, which is that random sampling of networks is growingly unlikely to get a good estimate of how the best network looks, if the number of nodes keeps increasing. For small toy problems this doesn’t matter much, but actual brain data consists of approximately 30,000 voxels and would be divided, if the anatomical regions are anything to go by, into over 100 clusters. This would result in an intractably huge space to sample from.

The next attempt to fix this was to use a Chow-Liu tree[2]. Chow-Liu trees can be computed relatively quickly, and they are the optimal network structures within the subspace of Bayesian networks which have a tree topology. Such a tree would serve as a baseline for determining how good a clustering would be; even if it wasn’t the best possible network, it was still, in theory, equally non-optimal for any possible clustering, and would serve just as well as a score.

4 Searching for Partitions using ICA

Independent component analysis is another widely-used tool to separate data into a number of discrete partitions. The goal of ICA is that the mutual information between partitions is minimized; it has successfully been used to separate audio data into independent channels, for example. One implementation of this

is the FastICA algorithm by Hyvarinen[4], which uses the concept of negentropy² to derive an iterative algorithm to solve the problem of partitioning variables into clusters. However, these clusters are completely unrelated to one another (in fact, that's the whole goal of ICA — to make the clusters as unrelated as possible), whereas our goal is to find a partitioning where the clusters fit into a given Bayesian network as well as possible.

As mentioned before, this leads to a framework where we assume some given network topology and attempt to cluster our known variables into the initially-empty nodes. This requires varying the ICA algorithm; instead of finding the arrangement which minimizes the mutual information between partitions, we now wish to minimize the mutual information between independent nodes while maximizing it between connected nodes (the directionality of the edges is irrelevant here). The optimization problem Hyvarinen uses to define ICA is:

$$\begin{aligned} & \text{given } \mathbf{s} = \mathbf{W}\mathbf{x}, \\ & \text{maximize } \sum_{i=1}^n J_G(\mathbf{w}_i) \text{ wrt. } \mathbf{w}_i, i = 1, \dots, n \\ & \text{under the constraint } E \{(\mathbf{w}_k^T \mathbf{x})(\mathbf{w}_j^T \mathbf{x})\} = \delta_{jk} \end{aligned}$$

Here, \mathbf{x} denotes a zero-mean vector of observed data and \mathbf{s} the vector of data transformed such that there is minimal mutual information between its elements. The weight matrix \mathbf{W} has rows \mathbf{w}_i , and $J_G(\mathbf{w}_i)$ denotes the negentropy of a given row of the matrix subject to a given approximator G . The delta function δ_{jk} is defined as $\delta_{jk} = 1$ if and only if $j = k$, and $\delta_{jk} = 0$ otherwise.

4.1 Altering ICA

It would seem, then, that the right thing to do would be to change the expression being maximized into something of the form

$$\sum_{i=1}^n J_G(\mathbf{w}_i) + \sum_{(u \rightarrow v) \in E} I(u; v) \quad (1)$$

The notation $(u \rightarrow v) \in E$ means “pairs of variables u and v such that there is an edge between them in the network”, but what this does to the rest of the algorithm is uncertain. However, we can at least rewrite this expression in terms of negentropy. Using the formula for mutual information $I(X; Y) = H(X) + H(Y) - H(X, Y)$, we obtain:

$$\sum_{i=1}^n J_G(\mathbf{w}_i) + \sum_{(u \rightarrow v) \in E} I(u; v)$$

² Negentropy is defined as $J(\mathbf{y}) = H(\mathbf{y}_{gauss}) - H(\mathbf{y})$, where \mathbf{y}_{gauss} is a Gaussian random vector with the same covariance matrix as \mathbf{y} . The entropy of such a vector is a constant given a particular covariance matrix, so $H(\mathbf{y}_{gauss})$ is a direct function of \mathbf{y} .

$$= \sum_{i=1}^n J_G(\mathbf{w}_i) + \sum_{(u \rightarrow v) \in E} (H(u) + H(v) - H(u; v)) \quad (2)$$

From the formula for negentropy, $J(x) = H(x_{gauss}) - H(x)$, and so $H(x) = H(x_{gauss}) - J(x)$, and we obtain:

$$= \sum_{i=1}^n J_G(\mathbf{w}_i) + \sum_{(u \rightarrow v) \in E} (H(u_{gauss}) + H(v_{gauss}) - H(u; v_{gauss}) - J_G(u) - J_G(v) + J_G(u; v)) \quad (3)$$

Unfortunately, there doesn't seem to be any way to simplify this sum; everything on the inside depends on the topology of the network. For instance, even though $H(u_{gauss})$ is a constant depending only on the covariance of u , it gets added to the overall sum a number of times equal to how many edges in the network begin at u . There should be a way to rewrite u and v in terms of the \mathbf{w}_i ; since u and v represent two of the nodes in our network which are filled in by some combination of voxels, and that combination is determined by the \mathbf{w}_i . However, more work needs to be done on exactly what this relation is and how it changes the expression above. If we call the resulting value $N(\mathbf{w}_i)$ (equal to the second of the two sums in the above equation), our minimization problem can be stated:

$$\text{maximize } \sum_{i=1}^n J_G(\mathbf{w}_i) + N(\mathbf{w}_i) \text{ wrt. } \mathbf{w}_i, i = 1, \dots, n$$

$$\text{under the constraint } E \{(\mathbf{w}_k^T \mathbf{x})(\mathbf{w}_j^T \mathbf{x})\} = \delta_{jk}$$

5 The Java Implementation

This section of the paper will not only describe the structure of the Java classes used to implement the ideas in this paper, but the equations used in them to compute the needed priors and likelihoods. The code is available at <http://www.cs.unm.edu/~benj/part/partitioning-java.zip>, with corresponding Javadoc at <http://www.cs.unm.edu/~benj/part/doc/>.

5.1 The MCMCProcess Class

This class is the main engine that drives a Markov Chain Monte Carlo process; while it's used here for the purpose of searching over the spaces of partitions as well as in a Bayesian network structure search class, it can be adapted to run an MCMC process of any type by defining appropriate actions and targets.

An MCMC process is here defined as a set of actions and a target. The actions are all single instances of classes which implement the MCMCACTION interface, while the target can be any object at all. The MCMCPROCESS class

does not directly refer to the target; instead, it uses the `SETTARGET` method on each action to make them refer to the proper object (see the `MAINDRIVER` class for an example of this).

5.1.1 The MCMCAction Interface

This interface defines six methods which all possible actions defined in this framework must contain. The important ones are:

`void reset()` This method “resets” an action by assigning it a random argument. For instance, in a Bayesian network search framework, if this is the “add edge” action, this would set a pair of fields to a randomly chosen source and destination node. This is the first method called on each action on every round of the MCMC process.

`double getChance()` This method is used to weight each action appropriately; since the list of actions only contains one of each “verb” (e.g. “merge” or “add edge”) without a corresponding argument (e.g. “region 5”, “nodes 2 and 7”), then to truly choose an action at random, this should return a number equal to (or at least related somehow) to the number of possible different actions belonging to each verb.³ This mechanism is also how we detect invalid actions; if the arguments of this action (as set by the `RESET` method) make it invalid, this should return 0.

`double propose()` This method will return the difference in the target’s score (where “score” is defined however is appropriate for what type of object the target is) caused by hypothetically taking the action (with the arguments as above). This could come from a mathematical analysis of the target and what effect the action would have on it, or more straightforwardly by making a copy of the target, applying the action to the copy, and taking the difference of the scores on the original and the copy. This copy can then be saved around for use in the `PERFORM` method. Alternatively,⁴ an “undo” function could be implemented on the target object that will roll back the change caused by any of the MCMC-related actions.

`void perform()` This method performs the action, changing the target object and preparing it for the next MCMC step. If there’s a saved “hypothetical” instance from when the `PROPOSE` method was called, this could simply assign that instance to be the new target for all future actions.

The other two methods are `TOSTRING`, which should just be defined in any reasonable way, and `GETTARGET`, which is simply an accessor method that returns the object which is the action’s target.

³ I didn’t actually take this into account in my own code, and only caught it recently. This may have been a problem, but there seemed to have been bigger ones.

⁴ This is the route that I take currently.

5.2 The BayesStructure Class

This class represents the structure of a Bayesian network. Edges can be added or removed, and the network can also be linked to a data set for purposes of computing the score. We use a Parzen windows-based estimator[3]; the equation given in the source paper is

$$p^M(x_i|\mathcal{P}_i) = \frac{\sum_{k=1}^D G((x_i, \mathcal{P}_i); (x_i^k, \mathcal{P}_i^k), \sigma_i^2)}{\sum_{k=1}^D G(\mathcal{P}_i; \mathcal{P}_i^k, \sigma_i^2)} \quad (4)$$

In other words, the probability of a specific node x taking on a specific value x_i , given its parents \mathcal{P} taking on their specific values \mathcal{P}_i , is equal to the quotient of two sums; the upper is the sum over all time steps of the value of a Gaussian at the point (x_i, \mathcal{P}_i) , with the mean of that Gaussian at the corresponding point for that time step and the standard deviation being a tunable parameter. The denominator is a similar sum over all time steps, but with one fewer dimension; it's a sum over Gaussians based only on the parents.

This score is multiplied by the prior $p(G)$, the probability of the graph independent of any data. This is calculated as follows:

$$p(G) = \beta^E (1 - \beta)^{\frac{n(n+1)}{2} - E} \quad (5)$$

In this equation, n is the number of nodes in the graph, making $\frac{n(n+1)}{2}$ the total number of possible edges that respect a given ordering. E is the number of edges actually present in the graph, and β is a tunable parameter, equal to the probability of a single edge being present. This is a straightforward binomial distribution.

5.3 The BayesNode Class

This class represents a single node in a Bayesian network. This is where the guts of loop detection exist; the static `PATHFROM` method checks whether a path exists between two nodes in the same network, which is used in the `MCADD` method to see if a proposed edge should be marked as invalid (and the action given a weight of zero). This could potentially be streamlined; there are spectral methods to detect loops in directed graphs which should be much faster than the search performed here.

5.4 The BayesStructureSearch Class

This class is meant to encapsulate the search for an optimal Bayesian network; it uses an `MCMCPROCESS` instance internally, and has only one public method apart from a main method used for testing:

```
search(double[][] data, double beta, double sigma) This method will (in theory)
    search for the optimal Bayesian network over the given data (where, for
    example, data[0][1] represents the value of node 0 at time step 1).
```

5.5 The ChowLiuTree Class

This class will search for a Chow-Liu tree given a 2D array of data, as well as a number of equal-sized bins to break the data into for quantization purposes (used to compute mutual information) and a threshold value to allow edges with only at least a certain amount of mutual information to be included — this is mainly used to break edges between what should be independent components of the graph, allowing for Chow-Liu tree-based clustering to be performed. The score of a Chow-Liu tree is currently defined here as equal to the sum of the mutual information scores of all of its edges, although a large commented-out block of code is provided that, when uncommented, will instead define the tree's score as the log-likelihood of the Bayesian network defined by this tree over the provided data.

5.6 The Ops Class

This abstract class contains a number of convenient methods that are used elsewhere. Like Java's built-in Math class, Ops is never actually instantiated. All of these are explained in the Javadoc, but one deserves special attention:

`double gaussian(double[] x, double[] mu, double sigma)` This method computes the value of a spherical Gaussian distribution at the vector x (represented as an array of doubles), given the vector μ as its mean and the value σ as its variance. The formula it uses, which can be straightforwardly derived from the equation for a general multidimensional Gaussian, is:

$$G(x; \mu, \sigma) = \frac{1}{\sqrt{\sigma}(2\pi)^{N/2}} \exp\left(-\frac{\|x - \mu\|_2^2}{2\sigma}\right), \quad x, \mu \in \mathbb{R}^N \quad (6)$$

5.7 The FakeData Class

This class is used to generate random data for use with clustering algorithms. Its two main methods, `GENERATEARRAY` and `GENERATEFILE` both work the same way but produce different output; `GENERATEARRAY` returns a two-dimensional array of doubles, while `GENERATEFILE` returns nothing but creates a new text file with the data in it, suitable for reading by any program that can parse lines of space-separated values into an array. Both of these methods take similar arguments; a number of fake nodes to generate based on random linear combinations of an underlying set of hidden nodes, a number of time steps to generate data for (data at each time step is independent of all time steps), and an amount of Gaussian noise to add to each observation. The number of clusters can be adjusted by changing a constant within the source file (obviously, this is something that can and should be changed for flexibility; this should really be an argument to the generation methods).

5.8 The OriginalData Class

This class is used to encapsulate a 2-dimensional array of doubles, allowing it to be constructed from an array of ints (something that Java oddly doesn't do automatically) or read from a file such as one that would have been the output of the FAKEDATA class. It also provides a method to get back a column of the data at once; Java provides a way to extract a row of a 2-D array — `array[n]`, if `array` is two-dimensional, will provide that array's *n*th row — but no equivalent to what might be written, mixing Java and Matlab notation, as `array[:,n]`. This functionality is also present in the Ops class as `extract`, which takes an array and a numeric index as arguments; the version in OriginalData just takes a numeric index. `data.getSeries(n)` is equivalent to `Ops.extract(data.getData(),n)`.

References

- [1] J. Burge, V.P. Clark, T. Lane, H. Link, and S. Qiu. Bayesian Classification of FMRI Data: Evidence for Altered Neural Networks in Dementia. *submission to Human Brain Mapping*, 2004.
- [2] C. Chow and C. Liu. Approximating discrete probability distributions with dependence trees. *Information Theory, IEEE Transactions on*, 14(3):462–467, 1968.
- [3] R. Hofmann and V. Tresp. Discovering Structure in Continuous Variables Using Bayesian Networks. *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*, pages 500–506, 1996.
- [4] A. Hyvarinen. Fast and robust fixed-point algorithms for independent component analysis. *Neural Networks, IEEE Transactions on*, 10(3):626–634, 1999.
- [5] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087, 1953.
- [6] J. Talairach and P. Tournoux. *Co-planar stereotaxic atlas of the human brain*. Thieme Medical Publishers New York, 1988.