Investigating Conditional Random Fields for Atlas-free MR Image Segmentation

Dominic Padova (dpadova1@jhu.edu)

ABSTRACT

Medical image segmentation aims to parcellate and label unseen images into biologically relevant structures. A daunting bottleneck in MRI analysis of the brain is tissue segmentation, in which the current norm is the time-consuming and subjective manual segmentation by experts. Therefore, an approach that can accurately segment brain tissues with minimal user input is highly desirable. Conditional random fields (CRFs) are discriminative undirected graphical modes that can directly model the conditional probability distribution for the object classification label given the training set. A CRF tissue classifier that could automatically label and segment soft tissue and cerebrospinal fluid/air given 2D brain MR images would serve as an alternative to segmentation models that require assumptions about the class conditional density distribution. Since this probabilistic model does not make assumptions about the distribution of the data (i.e. the priors do not have to be modeled), other input features besides voxel intensity can be used for learning tissue classifications (e.g. neighboring voxel correlations). Future work will involve improving the model with feature engineering, tuning the statistical techniques to sufficiently handle segmentation label variability (e.g. label fusion across imaging modalities), and to extend the model to 3D data.

1. Introduction

Tissue classification and segmentation of magnetic resonance (MR) brain images is an important task in medical image analysis that can aid in the diagnosis and prognosis of disease, as well as in medical decision processes. Two vetted approaches to computational brain parsing are atlas-based methods and probabilistic segmentation methods. Atlas-based methods utilize clinical expert-labeled images as references to segment the image into anatomically significant regions, and as such they incorporate high-levels of domain knowledge. More specifically, the label images are deformably registered to a target patient image, and through the nonlinear image warping, the target image can be labeled and segmented. One notable example is Multi-Atlas Label Fusion (MALF) approach. A popular MALF approach is to employ a weighted voting scheme, where larger weights are given to atlases which are more similar to the target image [1]. A more effective approach is to use the Large Deformation Diffeomorphic Metric Mapping (LDDMM) algorithm to non-linearly warp the target image to multiple atlases, and then use an Expectation Maximization (EM) algorithm for maximum a posteriori (MAP) estimation problems to update and fuse the likelihoods of the labels on structure-specific basis [2,3]. However, atlas-based approaches rely on the human knowledge of the definition of brain structures and tissues, which vary from expert to expert [6].

While atlas-based methods work well on normal brains, they can fail to quantitatively describe shape and size differences in abnormal brains, such as those undergoing the neurodegenerative changes of Alzheimer's disease, Huntington's disease, etc. The problem arises from the mathematical treatment of atlas-to-target registration: the normal brain atlas is mapped bijectively to the target image. This one-to-one mapping models the characteristics of all regions as being normal to some degree, and hence hinders the breadth of shape variability. This can decrease the ability of the algorithms to discriminate healthy characteristics from pathological ones [9]. Notably, the Random Orbit Model addresses the problem of identifying structural differences between normal and abnormal despite great biological variability via diffeomorphic transformations of characterized atlases [2,10].

On the other hand, probabilistic models do not rely on pre-delineated anatomical sections to segment brains. These methods can model the noise in images, handle the complex distributions of intensities in MR images, and tend to be simpler to implement. Probabilistic graphical models can compactly capture structural dependencies, context, and uncertainty in data as relationships between random variables in a graph. Markov Random Fields (MRFs) are a classification technique that model dependencies in the labels of an immediate neighborhood of voxels [4], and they have been used in medical image segmentation tasks. MRFs are generative probabilistic graphical models that are at times outmatched by their discriminative analogs, Conditional Random Fields (CRFs). For instance, when trying to label voxels as a certain tissue-type, MRFs generate all possibilities of image configurations and underlying tissue types, and it becomes computationally cumbersome. Thus, a discriminative approach using CRFs, which identifies the voxel-wise tissue type, is more parsimonious. For a thorough introduction to CRFs, see [7,8].

In this paper, we use Conditional Random Fields (CRFs) to mine and extract relevant information from MR brain images for tissue segmentation. The task can be stated as follows: Given a training set of raw MRI brain data, its corresponding ground-truth tissue-labeled images, and a CRF with a pre-ordained structure, we must compute (i.e. learn) the parameters of the CRF, and then use this model to segment and label tissues in a given testing set of semi-raw MRI brain data.

1.1. Markov Random Fields

Markov Random Fields (MRFs) are generative probabilistic graphical models that have been used for medical image processing tasks. They have the beneficial capability to classify one element based on the labels in a given immediate neighborhood. Note that this neighborhood can be defined to be a sequence, a 2D image, or a 3D volume. In other words, MRFs avoid the problem of statistically modeling the data as independent and identically distributed (i.i.d.)—an assumption that violates known probabilistic dependence amongst neighborhood labels. MRFs model the joint probability density of the features of a set of voxels $\boldsymbol{x} = \{x_1,\dots,x_n\}$ and their corresponding labels $\boldsymbol{y} = \{y_1,\dots,y_n\}$:

$$p(\boldsymbol{x},\boldsymbol{y}) = p(\boldsymbol{x}|\boldsymbol{y})p(\boldsymbol{y}).$$

Modeling the joint probability is a very computationally burdensome task. Normally, to remedy this, a factorized form is assumed:

$$p(\boldsymbol{x}|\boldsymbol{y}) = \prod_i p(x_{\boldsymbol{i}} \,|\, y_{\boldsymbol{i}})\,.$$

However, there is a tradeoff: making the model more tractable disallows modeling more complex relationships between features and labels [4].

1.2. Conditional Random Fields

Conditional Random Fields (CRFs) are discriminative undirected graphical modes that can directly model the conditional probability distribution for the object classification label given the training set. In contrast to MRFs, CRFs do not model the joint likelihood $p(\boldsymbol{x},\boldsymbol{y})$. Instead, they model the posterior probability of the labels given the voxel features $p(\boldsymbol{y}|\boldsymbol{x})$. In this formulation, $\boldsymbol{x}$ represents the set of random variables taking evidence data values, and $\boldsymbol{y}$ represents the set of hidden random variables for labels. As a result, there are no assumptions made about the distribution of the data (i.e. it avoids modelling $p(\boldsymbol{x}|\boldsymbol{y})$). Further, more complex dependencies can be captured between elements and labels, the labels in a neighborhood, and the labels of neighboring elements and their features. Moreover, more complex relationships between labels and other user-specified features of the evidence can be captured. These points are important in medical image processing and analysis because the intensity distributions of the image voxels are complex (there are multiple types of tissues contributing to the signal in each voxel), and it is likely not appropriate to assume a simplified factorized form of $p(\boldsymbol{x}|\boldsymbol{y})$) [4].

2. Problem Formulation

We define a graph $G = (V,E)$, where V denotes the set of all vertices $V_i \in V$, where each $y_{\boldsymbol{i}} \in \boldsymbol{y}$ corresponds to a vertex $V_{\boldsymbol{i}}$, and where E is the set of edges $e \equiv (i,j) \in E$ which represent the pairwise connection between two labels $y_{\boldsymbol{i}}$ and $y_{\boldsymbol{j}}$. We can formulate this as a factor graph with two sets of binary feature functions: (i) $f_k(y_{\boldsymbol{i}},x_{\boldsymbol{i}})$, which represents the link between the hidden label variable and its corresponding observation data; and (ii) $f_k(y_{\boldsymbol{i}},y_j,x_{\boldsymbol{i}})$, which represents the link between two labels $y_{\boldsymbol{i}}$ and $y_{\boldsymbol{j}}$ and the first label's corresponding observation data. Using a simplified factor graph notation for the probability of the latent variables $\boldsymbol{y}$ conditioned on the evidence $\boldsymbol{x}$, we have:

$$p(\boldsymbol{y}|\boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} e^{\left\{\sum_{(i,j)\,\in\,E,k} \lambda_k f_k(y_i,y_j,x_i)\right\}},$$

where the partition function is

$$Z(\boldsymbol{x}) = \sum_{\boldsymbol{y}} e^{\left\{\sum_{(i,j)\,\in\,E,k} \lambda_k f_k(y_i, y_j, x_i)\right\}}.$$

In the above expressions, parameters λ_k are weights for each feature f_k, where subscript k is a variable which indexes each feature function in the set.

The data mining task is to find the label y_i that maximizes the conditional probability of the latent variables $\boldsymbol{y}$ conditioned on the evidence $\boldsymbol{x}$:

$$\hat{\boldsymbol{y}} = \arg\max_{\boldsymbol{y}} P(\boldsymbol{y}|\boldsymbol{x})$$

This corresponds to the maximum a posteriori (MAP) assignment. This all follows from [9].

3. Methods

3.1. Datasets

We used the IBSR V1.0 and IBSR V2.0 datasets taken from the Internet Brain Segmentation Repository (IBSR) at http://www.nitrc.org/projects/ibsr. IBSR V1.0 is comprised of 20 low resolution, normal brains, and IBSR V2.0 is comprised of 18 high resolution 1.5mm T1-weighted scans. Also included in the datasets are expert labeled "ground-truth" images, which delineate the three tissue types and the background. All scans have been positionally normalized into the Talairach orientation (through rotation only) and processed by the Center for Morphometric Analysis (CMA) AutoSeg bias field correction routines. The MR brain data sets and their manual segmentations were provided by the Center for Morphometric Analysis at Massachusetts General Hospital and are available at http://www.cma.mgh.harvard.edu/ibsr/.

3.2. CRF Segmentation Implementation Details

The CRF tissue segmentation procedure is carried out in three main phases: 1) Feature Extraction, 2) Training, and 3) Inference. All computations were performed on a TOSHIBA laptop with INTEL i7 CPU @ 2.4 GHz, 4 cores, 6GB RAM. Training and inference were carried out using the UGM package [11]—a package which can handle learning a large number of CRF parameters.

We extract from each image three features: (i) voxel intensity, (ii) mean intensity of the 3x3 neighborhood of each voxel, and (iii) the position of the voxel (i.e. Euclidean distance from the voxel to the center of the image). Then we append a bias feature to the beginning of each voxel's feature vector. Thus, there are four inputs: a bias term/constant, and the three image features (i), (ii), and (iii) [9].

The goal of the training phase is to learn the weight λ_k for each feature f_k, where subscript k is a variable which indexes each feature function in the set [9]. Each observation x_i is a feature vector corresponding to each voxel in the image slices that includes the voxel's own intensity value, the average intensity of its 8 neighbors, if looking at single slices, or its 26 neighbors if looking at the entire volume (i.e. the average intensity of the 3-by-3-by-3 neighborhood around the voxel), the position of the voxel (taken as the Euclidean distance from voxel to the image center), and, optionally, the multiscale texture statistics of each voxel. Each voxel has a corresponding hidden variable $\boldsymbol{y_i}$, which can take one of four categorical label values $\{BG \rightarrow 1, CSF \rightarrow 2, GM \rightarrow 3, WM \rightarrow 4\}$ (i.e. tissue types → integer labels), where WM is the white matter, GM is the gray matter, CSF is the cerebrospinal fluid, and BG is background (or equivalently, none of the above). Also, the string labels are mapped to integer labels—not one-hot-encoded—for computation, as denoted in the set notation. Note that each hidden variable is connected to either 8 or 26 other hidden nodes in the neighborhood, depending on if the user is looking at the individual slices or volumes. In this work, we define the neighborhood to be 3x3, and as such connect label nodes to eight others.

The goal of the inference phase is to classify the tissue located at each voxel based on the models learned parameters [9]. To do this, we must compute the MAP estimate over each of the different voxels according to

$$\hat{\boldsymbol{y}} = \arg\max_{\boldsymbol{y}} P\left(y_i = \hat{y}_i \,\middle|\, x_i\right), \forall i.$$

We do this MAP estimate of the labels by iterated conditional models (ICM) with 30 restarts [9, 12]. After we yield out estimated images, we compare them to the "ground-truth" using the DICE coefficient [9]. The DICE coefficient is a measure of overlap of regions, and is defined analogously to the F1-score in terms of receiver operating characteristics (ROCs). We define the DICE coefficient as follows:

$$\frac{2(TP)}{(TP + FP) + (TP + FN)},$$

where TP is the true positive rate, FP is the false positive rate, and FN is the false negative rate [9].

4. Tutorial

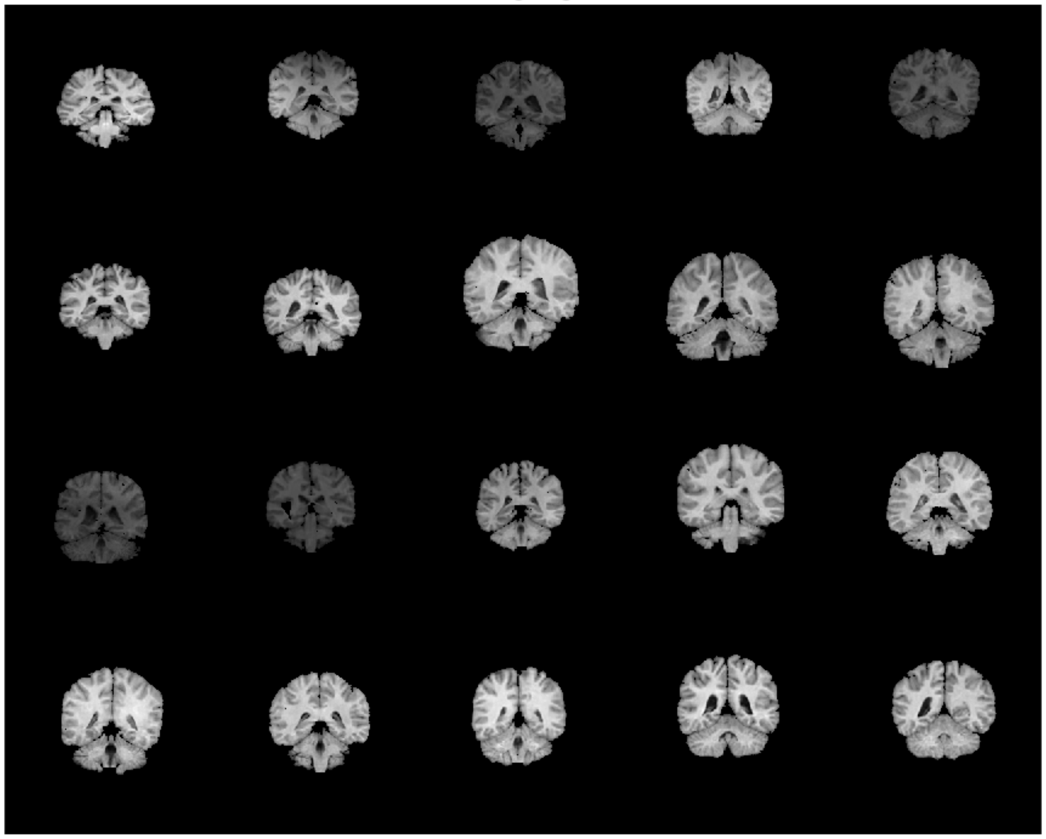
Loading and Visualizing the Data

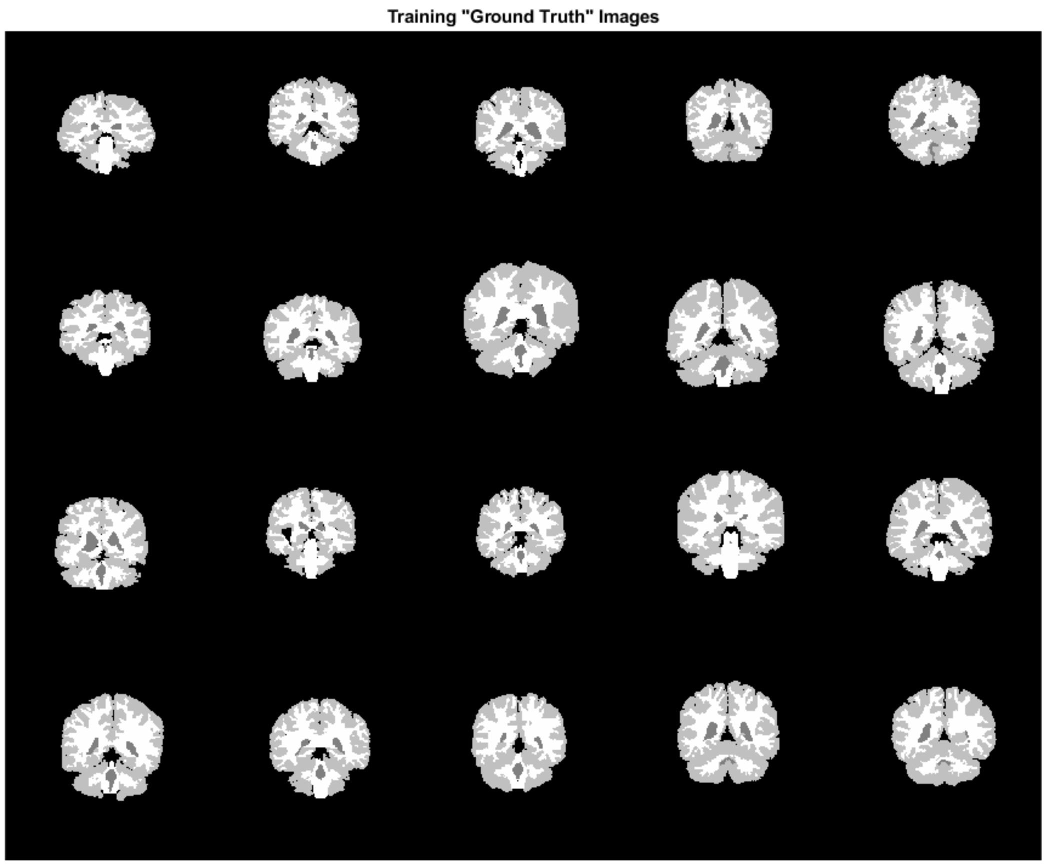
```
clear all, clc, close all;

fprintf('\nLoading Data...\n');
```

```
dirname = 'C:\Users\domin_000\Desktop\Hopkins Stuff\625.492.81 Probabilistic Graphical Models\CRFs';
cd(dirname);
load IBSR_V1_images.mat
load IBSR_V1_labels.mat
load IBSR_V2_images.mat
load IBSR_V2_labels.mat

Ytrain = zeros(size(IBSR_V1_labels{1},1),size(IBSR_V1_labels{1},2),length(IBSR_V1_labels),'int32');
Xtrain = zeros(size(IBSR_V1_images{1},1),size(IBSR_V1_images{1},2),length(IBSR_V1_images),'int32');
for ii = 1:length(IBSR_V1_labels)
    Ytrain(:,:,ii) = IBSR_V1_labels{ii}(:,:,20);
    Xtrain(:,:,ii) = IBSR_V1_images{ii}(:,:,20);
end

% Visualize the training images with contrast enhancement
figure(1),montage(reshape(uint8(imrotate(Xtrain,-90)),[256,256,1,20]),[]);
title('Training Images');

% Visualize the training labels
figure(2),montage(reshape(uint8(imrotate(Ytrain,-90)),[256,256,1,20]),[]);
title('Training "Ground Truth" Images');


Ytest = zeros(size(IBSR_V2_labels{1},1),size(IBSR_V2_labels{1},2),length(IBSR_V2_labels),'int32');
Xtest = zeros(size(IBSR_V2_images{1},1),size(IBSR_V2_images{1},2),length(IBSR_V2_images),'int32');
for ii = 1:length(IBSR_V2_labels)
    Ytest(:,:,ii) = IBSR_V2_labels{ii}(:,:,20);
    Xtest(:,:,ii) = IBSR_V2_images{ii}(:,:,20);
end
```

Here is what the training data, Xtrain and Ytrain (20 2D slices), looks like:

Training Images





Feature Extraction

After loading the data, the first course of action is to extract from each image the three features: (i) voxel intensity, (ii) mean intensity of the 3x3 neighborhood of each voxel, and (iii) the position of the voxel (i.e. Euclidean distance from the voxel to the center of the image). Then we append a bias feature to the beginning of each voxel's feature vector. In essence, there are four inputs: a bias term/constant, and the three image features (i), (ii), and (iii). We do this for both the training and testing images so that we can build their respective graphs later.

```
fprintf('Computing image features...\n');

% Build feature matrix
voxelDistance = zeros(size(Xtrain));
meanNeighborIntensities = zeros(size(Xtrain));
tic;
for i = 1:size(Xtrain,1)
    for j = 1:size(Xtrain,2)
```

```
        for k = 1:size(Xtrain,3)
            x_center = ceil(size(Xtrain(:,:,k),2)/2);
            y_center = ceil(size(Xtrain(:,:,k),1)/2);
            voxelDistance(i,j,k) = sqrt((x_center - i)^2 + (y_center - j)^2);
            kkNeighbors = 8;
            kk = [1 1 1; 1 0 1; 1 1 1]/kkNeighbors;
            meanNeighborIntensities(:,:,k) = conv2(Xtrain(:,:,k),kk,'same');
        end
    end
end
toc;

X_1 = [Xtrain(:), meanNeighborIntensities(:), voxelDistance(:)];
X_1 = reshape(X_1, [size(Xtrain,3),size(X_1,2),size(Xtrain,1)*size(Xtrain,2)]);
X_1 = double(X_1);

% Build feature matrix
voxelDistance = zeros(size(Xtest));
meanNeighborIntensities = zeros(size(Xtest));
tic;
for i = 1:size(Xtest,1)
    for j = 1:size(Xtest,2)
        for k = 1:size(Xtest,3)
            x_center = ceil(size(Xtest(:,:,k),2)/2);
            y_center = ceil(size(Xtest(:,:,k),1)/2);
            voxelDistance(i,j,k) = sqrt((x_center - i)^2 + (y_center - j)^2);

            kkNeighbors = 8;
            kk = [1 1 1; 1 0 1; 1 1 1]/kkNeighbors;
            meanNeighborIntensities(:,:,k) = conv2(Xtest(:,:,k),kk,'same');
        end
    end
end
toc;

X_2 = [Xtest(:), meanNeighborIntensities(:), voxelDistance(:)];
X_2 = reshape(X_2, [size(Xtest,3),size(X_2,2),size(Xtest,1)*size(Xtest,2)]);
X_2 = double(X_2);
```

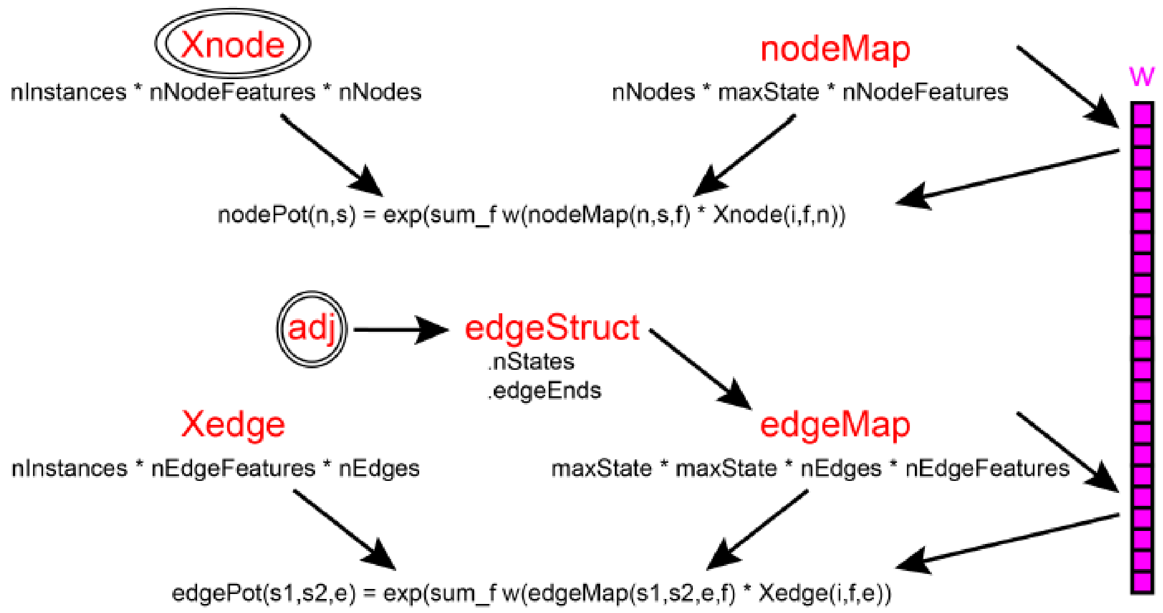
Note: one easy way to compute the mean intensity of a 2D neighborhood is with Matlab's conv2() function. If we wanted to perform a 3D convolution, we could do this in Fourier space as well. For 3D images, and higher dimensional datasets, it may be useful to employ the Fourier transform and take advantage of the FFT algorithm, since the number of operations is proportional to NlogN, whereas in standard convolution the number of operations required to convolve them in the straightforward manner is proportional to N^2. I will employ this approach in future work

Building the CRF

Before we build the training CRF, we must put the training data in the UGM format and define the preliminary variables. This involves mapping the input labels to integers 1-to-K, where K is the number of labels. Note that this is not meant to be a 1-of-K encoding/one-hot-encoding. The minimum integer label must start at 1 because the value is used as an index later, and Matlab indexes beginning at 1.

```
cd('C:\Users\domin_000\Desktop\Hopkins Stuff\625.492.81 Probabilistic Graphical Models\CRFs\UGM_2011\UGM\');
addpath(genpath(pwd));

% White Matter, Gray Matter, Cerebrospinal Fluid, Background
labels = {'WM'; 'GM'; 'CSF'; 'BG'};

% Put into UGM format
fprintf('Putting MRI data into UGM format...\n');

[nRows, nCols, nInstances] = size(Xtrain);
nNodes = nRows*nCols;
nStates = 4;
maxState = max(nStates); % Maximum number of states that any node can take

ytrain = reshape(Ytrain, [size(Ytrain,3),size(Ytrain,1)*size(Ytrain,2)]);
tissue_vals = unique(ytrain);
for i = 1:length(ytrain(:))
   if ytrain(i) == tissue_vals(1)
      ytrain(i) = 1;
   elseif ytrain(i) == tissue_vals(2)
      ytrain(i) = 2;
   elseif ytrain(i) == tissue_vals(3)
      ytrain(i) = 3;
   elseif ytrain(i) == tissue_vals(4)
      ytrain(i) = 4;
   end
end
```

Before we describe how to construct the CRF, it will be instructive to see pictorially how the UGM software compiles the information.


Xnode
nInstances * nNodeFeatures * nNodes
nodeMap
nNodes * maxState * nNodeFeatures
w
nodePot(n,s) = exp(sum_f w(nodeMap(n,s,f) * Xnode(i,f,n))
adj
edgeStruct
.nStates
.edgeEnds
Xedge
nInstances * nEdgeFeatures * nEdges
edgeMap
maxState * maxState * nEdges * nEdgeFeatures
edgePot(s1,s2,e) = exp(sum_f w(edgeMap(s1,s2,e,f) * Xedge(i,f,e))

In order to build the CRF graph, we must fill in a sparse matrix with constant values indicating the edge connections between nodes. We also omit the edges protruding out from the boundary nodes along the periphery of the image because there are no nodes on the other side to connect them to. The edge structure ends up being a lattice of ones with zeros along the first and last rows and zeros along the first and last column for our model. Essentially, this adjacency matrix defines a lattice-structured relationship between the labels. Note that each row of edgeStruct.edgeEnds gives the two nodes associated with an edge. We can also check our results for this setup to see if they are expected. For instance, how many edges are in an n x n grid? Well, there are 2*n^2 - 2*n edges. So, for 256 x 256 grid like the one we have for each same-sized image, n=256, and we have 130,560 edges and 65,536 nodes. We can see this is correct by checking nEdges = edgeStruct.nEdges.

Make edgeStruct

```
fprintf('Making Edge Structure to Track Edge Information...\n');
tic;
adj = sparse(nNodes,nNodes);

% Add Down Edges
index = 1:nNodes;
exclude = sub2ind([nRows nCols],repmat(nRows,[1 nCols]),1:nCols); % No Down edge for last
row
index = setdiff(index,exclude); % Find the values in index that are not in exclude.
adj(sub2ind([nNodes nNodes],index,index+1)) = 1;

% Add Right Edges
```

```
index = 1:nNodes;
exclude = sub2ind([nRows nCols],1:nRows,repmat(nCols,[1 nRows])); % No right edge for last column
index = setdiff(index,exclude);
adj(sub2ind([nNodes nNodes],index,index+nCols)) = 1;

% Add Up/Left Edges
adj = adj+adj';
edgeStruct = UGM_makeEdgeStruct(adj,nStates);
nEdges = edgeStruct.nEdges; % Number of edges
toc;
```

After constructing the edge structure, we want to build the matrices that represent the features that affect the node potentials and the edge potentials called Xnode and Xedge, respectively. Ultimately, Xnode will be an nInstances-by-nNodeFeatures-by-nNodes array and Xedge will be an nInstances-by-nEdgeFeatures-by-nEdges array, both in double format.

To make Xnode, we append a constant bias term to the beginning of the voxel feature vector, and we statistically standardize the columns of the feature vector (i.e. we standardize each feature across the nodes). As a result, the mean and standard deviation of each column are zero and one, respectively. Note that the bias reflects any effects on the states that are independent of the features.

Make Xnode

```
fprintf('Compiling features that affect the node potentials...\n');
tied = 1; % indicates to standardize each feature across nodes
Xnode = [ones(nInstances,1,nNodes), UGM_standardizeCols(X_1,tied)];
nNodeFeatures = size(Xnode,2); % 4 input features
```

Then we construct the node map encoding CRF node features. Each node can take on any one of four states (i.e. classes): {Background (or none of the above), Cerebrospinal Fluid, Gray Matter, and White Matter}. Since the CRF features parameterize over the observations and possible classes, the maximum number of node parameters is 4*4=16. In other words, each node has a parameter for each combination of input (and the constant) feature and output class.

```
row_first = sub2ind([nRows nCols],repmat(1,[1 nCols]),1:nCols);

col_first = sub2ind([nRows nCols],1:nRows,repmat(1,[1 nRows]));
```

```
row_last = sub2ind([nRows nCols],repmat(nRows,[1 nCols]),1:nCols);

col_last = sub2ind([nRows nCols],1:nRows,repmat(nCols,[1 nRows]));

NonBoundaryNodes = setdiff(1:nNodes,[row_first, row_last, col_first, col_last]);

BoundaryNodes = unique([row_first, row_last, col_first, col_last]);



% Make nodeMap
fprintf('Making Node Feature Map...\n');
nodeMap = zeros(nNodes,maxState,nNodeFeatures,'int32');
featNum = 1;
for f = 1:nNodeFeatures
   for s = 1:maxState
      for n = 1:length(NonBoundaryNodes)
         nodeMap(NonBoundaryNodes(n),s,f) = featNum;
      end
      featNum = featNum+1; % Should be 16 node features at the end
   end
end
nNodeParams = max(nodeMap(:));
fprintf('\nThe total number of node parameters is %d.\n', nNodeParams)
```

To make Xedge, we tie all the edge parameters together, so that each edge shares parameters, and the number of neighbors each node has doesn't change the number of total parameters in the model.

Make Xedge

```
% Make Xedge
fprintf('Preparing features that affect the edge potentials...\n');
sharedFeatures = ones(nNodeFeatures,1); % each edge shares parameters
Xedge = UGM_makeEdgeFeatures(Xnode,edgeStruct.edgeEnds,sharedFeatures);
nEdgeFeatures = size(Xedge,2);
```

Then, we can construct the edge map encoding CRF edge features. Each edge has the four inputs from each node, and how they affect the probability of each class, which gives us 4*4*4=64 edge parameters. If we follow Magnano et al. (2014), we need to create the “boundary” edge parameters. The “boundary” edge features correspond to the nodes on the periphery of the image, so they do not have neighbors on certain sides. Phrased differently, we treat the boundaries as edge features, rather than node features. Consequently, the code treats these boundaries as nodes that have edges which have no node on the other side, instead of treating these nodes as not having some edges. Thus, we need to calculate a separate set of edge parameters/features for these single-sided edges. Each single-sided edge has four inputs from one node, and as there is only a single side, we get 4*4=16 boundary edge parameters.

```
% Find the edges associated with the boundary nodes
BoundaryEdges = cell(length(BoundaryNodes),1);
BoundaryEdgeIdxs = [];
for i = 1:length(BoundaryNodes)
   node = BoundaryNodes(i);
   % fprintf('Here are the edge numbers associated with node %d\n',node);
   edges = UGM_getEdges(node,edgeStruct);
   BoundaryEdges{i} = edges;
   BoundaryEdgeIdxs = [BoundaryEdgeIdxs; edges(:)];
%     fprintf('These are the edgeEnds associated with these edges\n');
%     edgeEnds = edgeStruct.edgeEnds(edges,:)
%
%     fprintf('Here are the neighbors of node %d\n',node);
%     neighbors = edgeEnds(edgeEnds ~= node)
end
BoundaryEdgeIdxs = unique(BoundaryEdgeIdxs);
NonBoundaryEdgeIdxs = setdiff(1:nEdges,BoundaryEdgeIdxs);

% Make edgeMap
edgeMap = zeros(maxState,maxState,nEdges,nEdgeFeatures,'int32');
fb = 1;
for edgeFeat = 1:nEdgeFeatures
   for s = 1:maxState
      for n = 1:length(BoundaryEdgeIdxs)
         edgeMap(s,s,BoundaryEdgeIdxs(n),edgeFeat) = fb;
      end
      fb = fb+1; % Should be 16 boundary edge features (instead of node features) at the end
   end
end
BoundaryParams = edgeMap(:,:,BoundaryEdgeIdxs,:);
nBoundaryEdgeParams = max(BoundaryParams(:));
fprintf('\nThe total number of boundary edge parameters is %d.\n', nBoundaryEdgeParams)


ssf = 1;
for edgeFeat = 1:nEdgeFeatures
   for s1 = 1:maxState
      for s2 = 1:maxState
```

```
            for k = 1:length(NonBoundaryEdgeIdxs)
%                edgeMap(s1,s2,NonBoundaryEdgeIdxs(k),edgeFeat) = nNodeParams+ssf;
                edgeMap(s1,s2,NonBoundaryEdgeIdxs(k),edgeFeat)                          =
nNodeParams+nBoundaryEdgeParams+ssf;
            end
            ssf = ssf+1; % Should be 64 edge features at the end
        end
    end
end
nEdgeParams = max(edgeMap(:));
fprintf('\nThe total number of edge parameters is %d.\n', nEdgeParams)

nParams = max([nNodeParams;nEdgeParams;nBoundaryEdgeParams]);
fprintf('\nDesired total number of model parameters: 96.')
fprintf('\nActual total number of model parameters: %d.\n', nParams)


save('mri_crf_attempt01.mat'); % save progress
```

Overall, if we use the approach in [9], the maximum number of parameters for the CRF is 16 node parameters + 16 boundary edge parameters + 64 node parameters = 96 parameters. It's important to note that CRFs enable compact parameterization of the imaging data—in this case, data in three million voxels per 3D image volume is expressed as 96 model parameters useful for the prediction of brain tissue labels!

However, if we neglect modeling the boundary edge parameters, we get 80 parameters after making the CRF maps. The former approach will likely give better results, but here is the code to do it anyway.

```
% Make the nodeMap and edgeMap
fprintf('\nMaking CRF maps...\n')
ising = 0; % Use full potentials; Don't use Ising-like potentials
tied = 1; % Each node/edge shares parameters
paramLastState = 1;
[nodeMap,edgeMap]                                                                        =
UGM_makeCRFmaps(Xnode,Xedge,edgeStruct,ising,tied,paramLastState);
nParams = max([nodeMap(:);edgeMap(:)]);

fprintf('\nThe total number of model parameters should be 96.')
fprintf('\nThe total number of model parameters is %d.\n', nParams)
```

Training the CRF

To train the CRF, we employ the stochastic gradient descent (SGD) algorithm: at each iteration we pick a random training example, evaluate the gradient on that training example, and take a small step in the direction of the negative gradient, where the step size is fixed at a small constant value. Here, the objective function to be minimized is the negative log likelihood. CRFs directly model the conditional probability $p(Y \mid X) = \frac{1}{Z(w,X)} e^{w^T F(Y,X)}$. The negative log likelihood (NLL) function is given by

$$f(w) =-\frac{1}{n}\sum_{i=1}^{n} - w^T F(Y_i,X_i) + \log((Z(w,X_i)),$$

and the gradient of the NLL is given by

$$\nabla_f f(w) = -\frac{1}{n}\sum_{i=1}^{n} F(Y_i,X_i) + E_{Y|X}[F_f(Y_i,X_i)].$$

Note that NLL parameterization has the property of convexity, that this formulation has a partition function and marginals for each example i, and it maintains a maximum entropy interpretation [15]. In our implementation, inference is a sub-routine of learning the CRF parameters. We use loopy belief propagation (LBP) for estimating the partition function during training and the marginal probabilities during training.

Some helpful notes here are that: (i) nodePot is size [nNodes-by-maxState] and edgePot is size [maxState-by-maxState-by-nEdges]; (ii) UGM_LogConfigurationPotential called in UGM_CRF_NLL needs max values of the input training labels be maxState and the values must either be real positive integers or logicals, so this is why we need to map the image values to integers 1-4.

Training

```
% Free up some memory
clearvars -except ytrain ytest Xtest Ytest X_2 maxState nParams nInstances nRows nCols Xnode
Xedge nodeMap edgeMap edgeStruct

stepSize = 1e-4; % We use a small, fixed stepsize
w = zeros(nParams,1);
maxIter = 100;
fprintf('Estimation: Training for %d passes...\n',maxIter);

tic;
for iter = 1:maxIter*nInstances
    i = ceil(rand*nInstances);
    funObj                                                                                    =
@(w)UGM_CRF_NLL(w,Xnode(i,:,:),Xedge(i,:,:),ytrain(i,:),nodeMap,edgeMap,edgeStruct,@U
GM_Infer_LBP);
```

```
    [f,g] = funObj(w);
    fprintf('Iter = %d of %d (fsub = %f)\n',iter,maxIter*nInstances,f);

    w = w - stepSize*g;
end
toc;

tic;
maxState = 4;
nodeBelmat = zeros(nRows*nCols,maxState,nInstances);
for i = 1:nInstances
    [nodePot,edgePot]                                                          =
UGM_CRF_makePotentials(w,Xnode,Xedge,nodeMap,edgeMap,edgeStruct,i);
    nodeBel = UGM_Infer_LBP(nodePot,edgePot,edgeStruct);
    nodeBelmat(:,:,i) = nodeBel;
    figure(i*10);
    imagesc(reshape(nodeBel(:,2),[nRows,nCols]));
    colormap(gray);
    title(['LBP node marginals with truncated stochastic gradient parameters' num2str(i)]);
end
toc;

save('mri_crf_attempt01.mat'); % save progress
```

Magnano et al. (2014) [9] trained their CRF for 300-500 iterations over the data comprised of full image volumes. On their computer with a 4-core Intel i5-347OS 2.9 GHz processor and 31 GB of RAM, training took about 40 hours. Since my computer is older and less compute-friendly these days (7-year-old TOSHIBA laptop with INTEL i7 CPU @ 2.4 GHz, 4 cores, 6GB RAM), I figured working with single slice images rather than whole image volumes might help out on this front. Spoiler alert: I did not get good results, and have several issues to resolve. I attempted 100 (~13 min.), 200 (~26 min.), 800 (~2 hr.), and 4000 iterations (~26 hr.) over 20 training images, each trial yielding nonsensical images after the inference phase. I shall only show two examples of what failed MAP estimated images look like at the end.

After training, we build the CRF using the testing data in the same way we did using the training data.

```
cd('C:\Users\domin_000\Desktop\Hopkins      Stuff\625.492.81      Probabilistic      Graphical
Models\CRFs\UGM_2011\UGM\');
addpath(genpath(pwd));

labels = {'WM'; 'GM'; 'CSF'; 'BG'};
```

```
% Put into UGM format
fprintf('Putting Testing data into UGM format...\n');
[nRows, nCols, nInstances2] = size(Xtest);
nNodes = nRows*nCols;
nStates = int32(length(labels)); % Number of states that each node can take
maxState = max(nStates);

ytest = reshape(Ytest, [size(Ytest,3),size(Ytest,1)*size(Ytest,2)]);
ytest = ytest + 1;
```

Make edgeStruct

```
fprintf('Making Edge Structure to Track Edge Information...\n');
tic;
adj = sparse(nNodes,nNodes);

% Add Down Edges
index = 1:nNodes;
exclude = sub2ind([nRows nCols],repmat(nRows,[1 nCols]),1:nCols); % No Down edge for last
row
index = setdiff(index,exclude); % Find the values in index that are not in exclude.
adj(sub2ind([nNodes nNodes],index,index+1)) = 1;

% Add Right Edges
index = 1:nNodes;
exclude = sub2ind([nRows nCols],1:nRows,repmat(nCols,[1 nRows])); % No right edge for last
column
index = setdiff(index,exclude);
adj(sub2ind([nNodes nNodes],index,index+nCols)) = 1;

% Add Up/Left Edges
adj = adj+adj';
edgeStruct2 = UGM_makeEdgeStruct(adj,nStates);
nEdges2 = edgeStruct2.nEdges; % Number of edges

toc;
```

Make Xnode, Xedge

```
% Add bias (which reflects any effects on the states that are independent
% of the features) and Standardize Columns
fprintf('Compiling features that affect the node potentials...\n');
tied = 1; % indicates to standardize each feature across nodes
Xnode2 = [ones(nInstances2,1,nNodes), UGM_standardizeCols(X_2,tied)];
nNodeFeatures2 = size(Xnode2,2); % 4 input features
```

```
% Make Xedge
% tie all the edge parameters together, so the number of neighbors each
% node has doesn't change the number of total parameters in the model.
fprintf('Preparing features that affect the edge potentials...\n');
sharedFeatures2 = ones(nNodeFeatures2,1); % each edge shares parameters
Xedge2 = UGM_makeEdgeFeatures(Xnode2,edgeStruct2.edgeEnds,sharedFeatures2);
nEdgeFeatures2 = size(Xedge2,2);

row_first = sub2ind([nRows nCols],ones([1 nCols]),1:nCols);
col_first = sub2ind([nRows nCols],1:nRows,ones([1 nRows]));
row_last = sub2ind([nRows nCols],repmat(nRows,[1 nCols]),1:nCols);
col_last = sub2ind([nRows nCols],1:nRows,repmat(nCols,[1 nRows]));
NonBoundaryNodes = setdiff(1:nNodes,[row_first, row_last, col_first, col_last]);
BoundaryNodes = unique([row_first, row_last, col_first, col_last]);

% Make nodeMap
fprintf('Making Node Feature Map...\n');
nodeMap2 = zeros(nNodes,maxState,nNodeFeatures2,'int32');
featNum = 1;
for f = 1:nNodeFeatures2
    for s = 1:maxState
        for n = 1:length(NonBoundaryNodes)
            nodeMap2(NonBoundaryNodes(n),s,f) = featNum;
        end
        featNum = featNum+1; % Should be 16 node features at the end
    end
end
nNodeParams2 = max(nodeMap2(:));
fprintf('\nThe total number of node parameters is %d.\n', nNodeParams2)

% Find the edges associated with the boundary nodes
BoundaryEdges = cell(length(BoundaryNodes),1);
BoundaryEdgeIdxs = [];
for i = 1:length(BoundaryNodes)
    node = BoundaryNodes(i);
    % fprintf('Here are the edge numbers associated with node %d\n',node);
    edges = UGM_getEdges(node,edgeStruct2);
    BoundaryEdges{i} = edges;
    BoundaryEdgeIdxs = [BoundaryEdgeIdxs; edges(:)];
%     fprintf('These are the edgeEnds associated with these edges\n');
%     edgeEnds = edgeStruct.edgeEnds(edges,:)
%
%     fprintf('Here are the neighbors of node %d\n',node);
%     neighbors = edgeEnds(edgeEnds ~= node)
```

```
end
BoundaryEdgeIdxs = unique(BoundaryEdgeIdxs);
NonBoundaryEdgeIdxs = setdiff(1:nEdges2,BoundaryEdgeIdxs);

% Make edgeMap
edgeMap2 = zeros(maxState,maxState,nEdges2,nEdgeFeatures2,'int32');
fb = 1;
for edgeFeat = 1:nEdgeFeatures2
   for s = 1:maxState
      for n = 1:length(BoundaryEdgeIdxs)
         edgeMap2(s,s,BoundaryEdgeIdxs(n),edgeFeat) = fb;
      end
      fb = fb+1; % Should be 16 boundary edge features (instead of node features) at the end
   end
end
BoundaryParams2 = edgeMap2(:,:,BoundaryEdgeIdxs,:);
nBoundaryEdgeParams2 = max(BoundaryParams2(:));
fprintf('\nThe total number of boundary edge parameters is %d.\n', nBoundaryEdgeParams2)


ssf = 1;
for edgeFeat = 1:nEdgeFeatures2
   for s1 = 1:maxState
      for s2 = 1:maxState
         for k = 1:length(NonBoundaryEdgeIdxs)
            edgeMap2(s1,s2,NonBoundaryEdgeIdxs(k),edgeFeat)                    =
nNodeParams2+nBoundaryEdgeParams2+ssf;
         end
         ssf = ssf+1; % Should be 64 edge features at the end
      end
   end
end
nEdgeParams2 = max(edgeMap2(:));
fprintf('\nThe total number of edge parameters is %d.\n', nEdgeParams2);

nParams2 = max([nNodeParams2;nEdgeParams2;nBoundaryEdgeParams2]); % this is 96
fprintf('\nDesired total number of model parameters: 96.');
fprintf('\nActual total number of model parameters: %d.\n', nParams2); % this is 96

save('mri_crf_attempt01.mat');
```

Again, if we wish to use the 80-parameter method, we do this:

```
% Make the nodeMap and edgeMap
fprintf('\nMaking CRF maps.\n')
ising = 0; % Use full potentials; Don't use Ising-like potentials
tied = 1; % Each node/edge shares parameters
paramLastState = 1;
[nodeMap2,edgeMap2]                                                        =
UGM_makeCRFmaps(Xnode2,Xedge2,edgeStruct2,ising,tied,paramLastState);
nParams2 = max([nodeMap2(:);edgeMap2(:)]);
```

Inference

After the CRF is prepared, we perform MAP inference via iterated conditional modes (ICM). To find the image with high probability (ideally the maximum probability), we use ICM. ICM is an inference algorithm that maximizes local conditional probabilities sequentially. In essence, it is an application of coordinate-wise gradient ascent [Bishop p.389, 14]. All the "messages" that get passed from one node to the next "comprise a single value consisting of the new state of the node for which the conditional distribution is maximized" [Bishop p.415]. Basically, the way it works is: until convergence, for each node in the set of nodes (this is the "iterated" part), we set all other nodes as fixed (this is the "conditional" part, and solve for the non-fixed node (this is the "modes" part) [16]. ICM can be viewed as a greedy approximation to Gibbs sampling [Bishop p.546]. Since this a greedy algorithm, it can get stuck in local minima, which will likely be bad. To address this, we restart the ICM algorithm with different initializations—we chose 30 restarts following [9]. Last, these yMAP values are labels 1-4, so we map them back to the original unique voxel values in the "ground-truth" labeled testing images for visualization of our results.

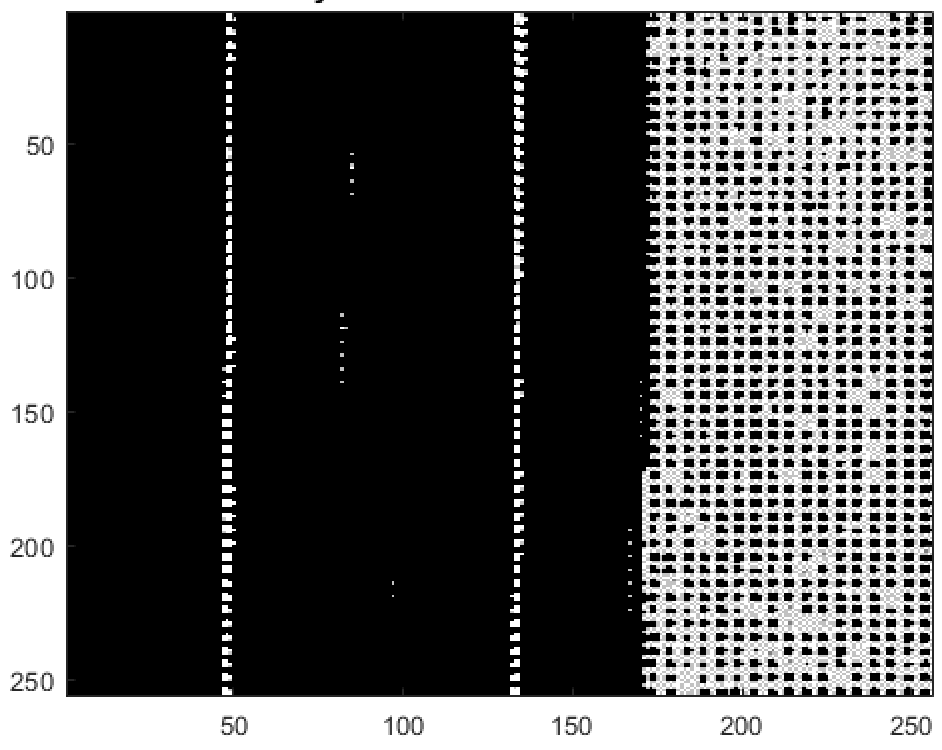
Step 6: Inference Phase: Perform MAP estimate of tissue labels

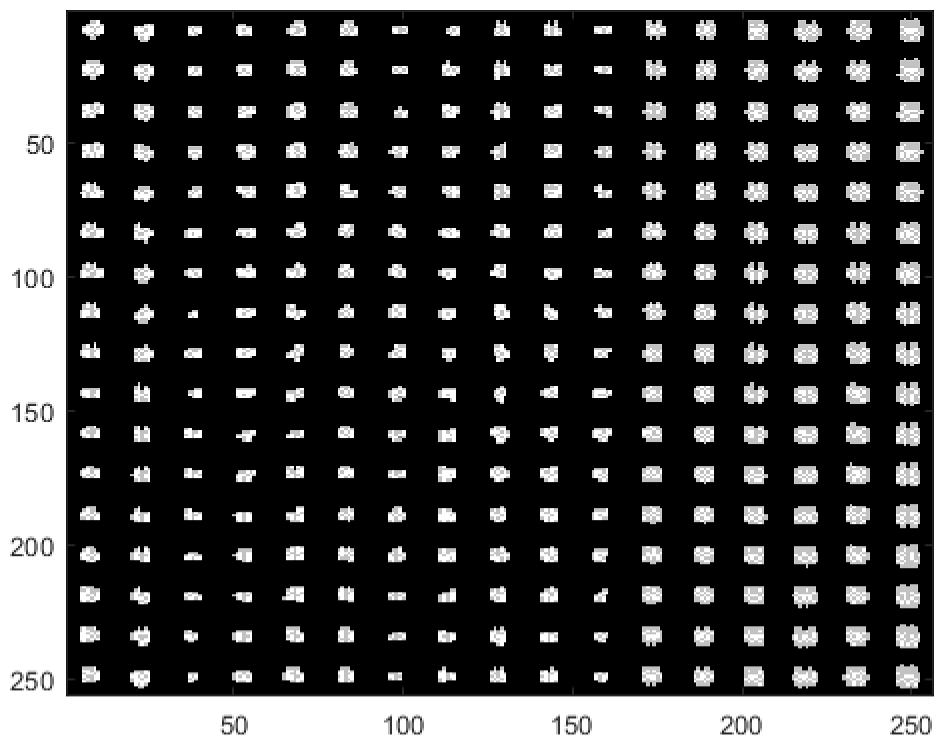
```
% Free up some Memory
clearvars -except maxState ytest nParams2 nNodes nInstances2 w Xnode2 Xedge2 nodeMap2
edgeMap2 edgeStruct2

fprintf('\nBeginning Inference Phase...\n');
nRestarts = 30;

% Compute test error
fprintf('Testing: MAP inference using ICM w/%d restarts...\n',nRestarts);
yMap_cell = cell(1,nInstances);
tic;
for i = 1:nInstances
   [nodePot2,edgePot2]                                                     =
UGM_CRF_makePotentials(w,Xnode2,Xedge2,nodeMap2,edgeMap2,edgeStruct2,int32(i));
```

```
   yMAP = UGM_Decode_ICMrestart(nodePot2,edgePot2,edgeStruct2,nRestarts); % this is
ICMrestartDecoding

   yMAP_cell{i} = yMAP;
   % Map the labels back to image pixel values
   % {“Background” = 0; “CSF” = 128; “GM” = 192; “WM” = 254} to a tissue label index {1; 2;
3; 4}?
   tissue_vals = unique(yMAP);
   for ii = 1:length(yMAP(:))
      if yMAP(ii) == tissue_vals(1)
         yMAP(ii) = 0;
      elseif yMAP(ii) == tissue_vals(2)
         yMAP(ii) = 128;
      elseif yMAP(ii) == tissue_vals(3)
         yMAP(ii) = 192;
      elseif yMAP(ii) == tissue_vals(4)
         yMAP(ii) = 254;
      end
   end

   figure(11*i),
   imagesc(squeeze(reshape(yMAP,[256,256,1]))), colormap(gray);

end

toc; % Last run took ~40secs.

fprintf('ICM Procedure Finished\n');

save('mri_crf_attempt01.mat');
```

Here are the (bad) estimated images that I got. The second image looks kind of interesting—like a bunch of small brains as nodes in the image. I don’t even know where to begin to critique them, so I will not.

yMAP after ICM w/Restarts





Now that the testing phase is complete, we can evaluate our segmentation accuracy with ROCs, but most importantly, the DICE coefficient (this is basically the F1 score).

Get Testing ROCs

```
fprintf('\nGetting the Testing ROCs ...\n')
Y_unique = unique(ytrain(:));
confusionMatrix = zeros(length(Y_unique));
for i = 1:length(Y_unique)
   for j = 1:length(Y_unique)
      v = (ytrain == Y_unique(i)) & (yMAP_cell{:}(:) == Y_unique(j));
      confusionMatrix(i,j) = sum(v);
   end
end

TP_test = zeros(1, length(Y_unique));
TN_test = zeros(1, length(Y_unique));
FP_test = zeros(1, length(Y_unique));
FN_test = zeros(1, length(Y_unique));
for i = 1:length(Y_unique)
   TP_test(i) = confusionMatrix(i,i);
   FN_test(i) = sum(confusionMatrix(i,:)) - confusionMatrix(i,i);
   FP_test(i) = sum(confusionMatrix(:,i)) - confusionMatrix(i,i);
   TN_test(i) = sum(confusionMatrix(:)) -...
      TP_test(i) - FP_test(i) - FN_test(i);
end

fprintf('Total TP = %d\n', sum(TP_test))
fprintf('Total FN = %d\n', sum(FN_test))
fprintf('Total FP = %d\n', sum(FP_test))
fprintf('Total TN = %d\n', sum(TN_test))

P_test = TP_test + FN_test;
N_test = FP_test + TN_test;
Accuracy_test = (TP_test)./(P_test + N_test);
Error_test = (FP_test)./(P_test + N_test);
Sensitivity_test = (TP_test)./P_test;
Specificity_test = (TN_test)./N_test;
FPR_test = 1 - Specificity_test;                 % False Positive Rate
Precision_test = (TP_test)./(TP_test + FP_test); % Also Positive Predictive Value
NPV_test = (TN_test)./(TN_test + FN_test);       % Negative Predictive Value
% beta = 1;
% F1_score = ((1+(beta^2))*(Sensitivity.*Precision)) ./ ((beta^2)*(Precision+Sensitivity));
F1_score_test = (2*TP_test) ./ (2*TP_test + FP_test + FN_test); % this def. avoids division by 0

% Compute the DICE coefficients in terms of ROCs
```

```
DICEcoeffs = 2 .* TP_test ./ ((TP_test + FP_test) + (TP_test + FN_test));

% n = size(tp, 1);
% AUC = sum((fp(2:n) - fp(1:n-1)).*(tp(2:n)+tp(1:n-1)))/2;

% Individual class results as struct
TestingROCstruct_1.TruePositive = TP_test';
TestingROCstruct_1.TrueNegative = TN_test';
TestingROCstruct_1.FalsePositive = FP_test';
TestingROCstruct_1.FalseNegative = FN_test';
TestingROCstruct_1.AccuracyTot = Accuracy_test';
TestingROCstruct_1.ErrorTot = Error_test';
TestingROCstruct_1.Sensitivity = Sensitivity_test';
TestingROCstruct_1.Specificity = Specificity_test';
TestingROCstruct_1.FPR = FPR_test';
TestingROCstruct_1.Precision = Precision_test';
TestingROCstruct_1.NPV = NPV_test';
TestingROCstruct_1.F1_score = F1_score_test';
TestingROCstruct_1.DICE = DICEcoeffs';

% Display the class-wise ROCs in a table
labels = {'BG','CSF','GM','WM'};
ROCIndividualparams = struct2table(TestingROCstruct_1,'RowNames',labels);
disp(ROCIndividualparams)

% % Display the table "ROCIndividualparams" in a figure
% figure(7)
% uitable('Data',ROCIndividualparams{:,:},...
%     'ColumnName',ROCIndividualparams.Properties.VariableNames,...
%     'RowName',ROCIndividualparams.Properties.RowNames,...
%     'Units', 'Normalized', 'Position',[0, 0, 1, 1]);

% Overall class results as struct
TestingResults_1.TestingSetAccuracy = TestingSetAccuracy;
% TestingResults.MeanSquaredError = mean(cost_test);
TestingResults_1.Sensitivity = mean(Sensitivity_test);
TestingResults_1.Specificity = mean(Specificity_test);
TestingResults_1.Precision = mean(Precision_test);
TestingResults_1.NPV = mean(NPV_test);
TestingResults_1.FPR = mean(FPR_test);
TestingResults_1.F1_score = mean(F1_score_test);
TestingResults_1.DICE = mean(DICEcoeffs);

% Display the overall Results in a table
disp(TestingResults_1)
```

```
ROCsOverallparams = struct2table(TestingResults_1);
disp(ROCsOverallparams)

% figure(8)
% uitable('Data',ROCsOverallparams{:,:},...
%     'ColumnName',ROCsOverallparams.Properties.VariableNames,...
%     'Units', 'Normalized', 'Position',[0, 0, 1, 1]);

% Save to excel spreadsheet
writetable(ROCIndividualparams,'myResults.xlsx','Sheet',1)
writetable(ROCsOverallparams,'myResults.xlsx','Sheet',2)
```

5. Discussion

I think that using the full image volumes along with considering a 3D voxel neighborhood (e.g. 3x3x3, 26 neighbors) is a good idea--better than using individual slices and a 2D neighborhood. Further, using the full 96 parameters specified in Magnano et al. (2014) would be superior to the 80-parameter approach. Using the extra parameters may capture the relationships within the image data better. Although I am disappointed that my implementation did not work as hoped, I am enthusiastic that I will be able to fix any errors and get decent segmentations. In the future, I will incorporate the use of full image volumes, 96 model parameters, the mean intensity of a 3D neighborhood, and perhaps include more image features into the feature vector (e.g. gradients, texture statistics, etc.). Additionally, I will compare this improved method to the Atlas-based (e.g. MRICloud, SPM) and Atlas-free software routines (e.g. FSL).

6. Appendix:

I plan to use texture statistics in the future:

```
% Set the wavelet name
wname = 'db4';

% Perform three levels of discrete 2D wavelet decomposition
[cA1,cH1,cV1,cD1] = dwt2(img, wname);
[cA2,cH2,cV2,cD2] = dwt2(cA1, wname);
[cA3,cH3,cV3,cD3] = dwt2(cA2, wname);

% Build extracted feature vector
dwt_features = [cA3,cH3,cV3,cD3];

% Reduce the dimensionality of the features (not sure if this is necessary
% or desired since CRF reduces dimensionality via parameterization)
```

```
pca_coefficients = pca(dwt_features);

% Create gray-level co-occurrence matrix from image to examine texture
gclm = graycomatrix(pca_coefficients);

% Get image and texture statistics
stats = graycoprops(gclm,{'contrast',...
    'correlation', 'energy', 'homogeneity'});
Contrast = stats.Contrast;
Correlation = stats.Correlation;
Energy = stats.Energy;
Homogeneity = stats.Homogeneity;
Mean = mean2(pca_coefficients);
Standard_Deviation = std2(pca_coefficients);
Entropy = entropy(pca_coefficients);
RMS = mean2(rms(pca_coefficients));
Variance = mean2(var(double(pca_coefficients)));
a = sum(double(pca_coefficients(:)));
Smoothness = 1-(1/(1+a));
Kurtosis = kurtosis(double(pca_coefficients(:)));
Skewness = skewness(double(pca_coefficients(:)));

% Construct image feature vector
feature_vec = [Contrast, Correlation, Energy, Homogeneity,...
    Mean, Standard_Deviation, Entropy, RMS, Variance,...
    Smoothness, Kurtosis, Skewness];
```

References:

[1] Wang, H., Suh, J. W., Das, S. R., Pluta, J., Craige, C., & Yushkevich, P. A. (2013). Multi-Atlas Segmentation with Joint Label Fusion. IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(3), 611–623. http://doi.org/10.1109/TPAMI.2012.143

[2] Tang X, Oishi K, Faria AV, Hillis AE, Albert MS, et al. (2013) Bayesian Parameter Estimation and Segmentation in the Multi-Atlas Random Orbit Model. PLOS ONE 8(6): e65591. https://doi.org/10.1371/journal.pone.0065591

[3] Tang X, Yoshida S, Hsu J, Huisman TAGM, Faria AV, et al. (2014) Multi-Contrast Multi-Atlas Parcellation of Diffusion Tensor Imaging of the Human Brain. PLOS ONE 9(5): e96985. https://doi.org/10.1371/journal.pone.0096985

[4] http://papersdb.cs.ualberta.ca/~papersdb/uploaded_files/268/paper_FINAL.pdf

[5] https://webdocs.cs.ualberta.ca/~btap/Papers/dana2007.pdf

[6] https://braingps.mricloud.org/docs/atlasrepodocs/AtlasDescriptionAndProtocolV7a.pdf

[7] C. Sutton and A. McCallum. An introduction to conditional random fields, 2010. arxiv:1011.4088.

[8] Hanna M. Wallach. Conditional Random Fields: An Introduction. Technical Report MS-CIS-04-21. Department of Computer and Information Science, University of Pennsylvania, 2004.

[9] Magnano C. S., Soni A., Natarajan S., Kunapuli.G. (2014). A graphical model approach to ATLAS-free mining of MRI images.

[10] Tang X, Crocetti D, Kutten K, Ceritoglu C, Albert MS, Mori S, Mostofsky SH, Miller MI (2015c): Segmentation of brain magnetic resonance images based on multi-atlas likelihood fusion: Testing using data with a broad range of anatomical and photometric profiles. Front Neurosci 9:61.

[11] M. Schmidt. UGM: A Matlab toolbox for probabilistic undirected graphical models. http://www.cs.ubc.ca/~schmidtm/Software/UGM.html, 2007.

[12] J. Besag. On the Statistical Analysis of Dirty Pictures. Journal of the Royal Statistical Society. Series B (Methodological), 48(3):259{302, 1986.

[13] Pattern Recognition and Machine Learning by Christopher M. Bishop, Springer, 2006 [ISBN 978-0387-31073-2]

[14] J. Kittler and J. Foglein. Contextual Classification of Multispectral Pixel Data. ¨ IMage and Vision Computing, 2(1):13–29, 1984

[15] Schmidt, M. Training MRFs, CRFs, and SSVMs. http://www.cs.ubc.ca/labs/lci/mlrg/slides/2015_MLRG_Learn.pdf

[16] Martinez, J. Approximate decoding: ICM, block methods, alpha-beta swap & alpha-expansion. http://www.cs.ubc.ca/labs/lci/mlrg/slides/approximate_decoding.pdf

Additional References:

[17] Iglesias, J. E., & Sabuncu, M. R. (2015). Multi-Atlas Segmentation of Biomedical Images: A Survey. Medical Image Analysis, 24(1), 205–219. http://doi.org/10.1016/j.media.2015.06.012

[18] Yang, Zhengyi, Choupan, Jeiran, Sepehrband, Farshid, Reutens, David and Crozier, Stuart

(2013) Tissue classification for PET/MRI attenuation correction using conditional random field and image fusion. International Journal of Machine Learning and Computing, 3 1: 87-92. doi:10.7763/IJMLC.2013.V3.278

[19] Hu Y.-C., Grossberg M., Mageras G., "Semi-automatic medical image segmentation with adaptive local statistics in conditional random fields framework," in 30th Annual Int. Conf. of the IEEE EMBS 2008, pp. 3099–3102 (2008).http://dx.doi.org/10.1109/IEMBS.2008.4649859

[20] Bauer S., Nolte LP., Reyes M. (2011) Fully Automatic Segmentation of Brain Tumor Images Using Support Vector Machine Classification in Combination with Hierarchical Conditional Random Field Regularization. In: Fichtinger G., Martel A., Peters T. (eds) Medical Image Computing and Computer-Assisted Intervention – MICCAI 2011. MICCAI 2011. Lecture Notes in Computer Science, vol 6893. Springer, Berlin, Heidelberg