Image segmentation using Machine Learning

Part I. A machine learning spin on threshold-based image segmentation

In Part I of this exercise, we will return to a simple method for image classification that we've used in previous modules: intensity-based thresholding. We will compare the performance of a manually assigned threshold to that of one that is computationally inferred, learning the basics of logistic regressions (a common ML framework) along the way.

Part II. Multi-parametric image segmentation

In Part II, we will build upon the simple thresholding approach, introducing a couple of common transforms that can be used to extract additional information about the structure of an image that we will then use to improve our classification algorithm.

Part III. Getting a bigger hammer

In Part III, we'll build on work from the previous section, introducing how to employ a Random Forest classifier in Matlab and learning how to add additional features to the classification scheme.

% House keeping commands
clear % clear all variables
close all % close all open figure windows
addpath('utilities') % add path to utilities folder
plot_cmap = brewermap(9,'Set1'); % define colormap to use for line and scatter plots along the way
cmap_im = flipud(brewermap(128,'RdYlBu')); % define colormap for color-scaled image visualization
cmap_bin = [0 0 0 ; 1 1 1]; % binary colormap for labeled images
figPath = 'slides_and_figures/';
mkdir(figPath);

Warning: Directory already exists.

Part I. A machine learning spin on threshold-based image segmentation

Previously, we have employed a simple thresholding approach to estimate how the total surface area of a bacterial colony changed over time. This kind of approach is a great place to start, but often it is not feasible to manually specify a threshold for each image of interest and, more fundamentally, a simple threshold approach may not be enough to achieve satisfactory classification in many cases. First, let's experiment letting a simple algorithm choose the intensity threshold for us.

Training, validation, and testing sets

In developing a classifier, the basic goal build something that can generalize effectively; that is, to build a classification scheme based on a set of data with known labels that we can then confidently apply to data for which the labels are unknown. Thus, it is common practice to divide one's labeled training data into three pieces: a training set, a validation set, and a testing set.

The training set is what we use to develop our classification procedure. We then test the accuracy of our procedure using the validation set. A final model is typically developed by iterating back and forth between
training and validation. Meanwhile, the testing set is kept "locked away" until we are ready to test the final version of the model. In essence, the validation and testing sets both serve to test the generality of a classifier; however, we hold the test set in reserve to ensure that it can serve as truly independent test of model performance. See this link for more details.

For the purposes of this tutorial, we will focus on segmenting a single image of *E. coli* cells. Thus, we will need to divide the pixels within this image into training, testing, and validation groups.

Open phase contrast colony image (the file with the suffix "_phase") using the imread function. The images are contained in the "training_data" folder (which should be in your current folder); therefore we must tell matlab to navigate there.

```
phase_image = imread('training_data/phase.tif');
phase_image = mat2gray(phase_image); % convert to grayscale
raw_image_fig = figure;
colormap(cmap_im)
imagesc(phase_image) % Display image with color scaling
h = colorbar;
ylabel(h,'pixel values');
title('Raw Image')
% save
saveas(raw_image_fig,[figPath 'phase_heatmap.tif'])
```

In this case--and this is critical--we also have access to a binary image that indicates the "ground truth" identity (is it inside or outside a bacterial cell?) for each pixel. Our aim will be to develop a machine learning-based classifier that is capable of inferring these labels as accurately as possible from the raw image. Note that, since we are explicitly showing our classifiers the correct labels for each pixel, this constitutes an kind of supervised machine learning.

Load the labeled image (suffix "_feature") that corresponds to the phase image loaded above. The labeled image contains two values that correspond to regions inside and outside of bacterial cells. Initially these values are "255" (inside) and "0" (outside). For convenience, let's binarize it such that the final labeled image has "1's" for inside regions and "0's" for outside regions.

```
lb_image = imread('training_data/feature.tif');
lb_image = mat2gray(lb_image); % convert to grayscale (puts all pixels between 0 and 1)
lb_image_fig = figure;
imshow(lb_image,'InitialMagnification','fit') % Display the image
title('Classified Image (ground truth)')
saveas(lb_image_fig,[figPath 'ground_truth_labels.tif'])
```
Manual thresholding

Now that we've loaded the raw and labeled images, the first order of business is to divide the pixels within the raw image into training, validation, and testing sets. We will use a couple of built-in matlab functions to randomly generate these assignments.

```matlab
n_pixels = numel(phase_image); % get number of pixels
% let's randomly assign a 1,2, or 3 (train, val, or test) to each pixel
id_vec_raw = repelem(1:3,n_pixels/3); % generate a vector of repeating 1's, 2's, and 3's
set_assignment_vec = id_vec_raw(randperm(n_pixels));
```

Let's focus on the training set first. We can use these labels to separate the pixels in our original image into those corresponding to interior and exterior regions.

```matlab
% generate vectors of "inside" and "outside" pixels. Recall that 1's in assignment_vec indicate
label_vec = lb_image(:)';
inside_pix_vec = phase_image(label_vec==1&set_assignment_vec==1);
outside_pix_vec = phase_image(label_vec==0&set_assignment_vec==1);
% compare pixel intensities between the two groups
edges = linspace(0,1,50);
hist_fig = figure;
colormap(plot_cmap)
hold on
histogram(inside_pix_vec,edges,'Normalization','probability','FaceColor',plot_cmap(1,:))
histogram(outside_pix_vec,edges,'Normalization','probability','FaceColor',plot_cmap(2,:))
legend('interior', 'exterior')
```
As we might expect from the raw image, the pixels in the bacterial interiors tend to have lower values than those from the background, since the cells are opaque. Note also that there is quite a bit of overlap between the two groups: no matter what we do, pixel intensity alone will not be enough to perfectly separate these groups. We’ll return to this issue in a bit.

For now, let’s see what we can achieve using a simple intensity threshold. Inspect the histogram using the “Data Cursor” option and make a guess about the best threshold to use in order to distinguish between inside and outside pixels.

thresholdManual = .17; % guess at threshold

Let’s examine how well our choice of threshold did at segmenting the image.

Decent but nothing to write home about. Let’s be a bit more quantitative about the performance. There are a number of metrics we could use to characterize performance. Two intuitive choices are the fraction of false positives (FP) (0’s classified as 1’s) and false negatives (FN) (1’s as 0’s) our classifier returns. We will calculate these values for the validation set.

% extract validation vectors
man_thresh_vec_val = imThreshManual(set_assignment_vec==2); 
true_lb_vec_val = lb_image(set_assignment_vec==2);

% calculate number of false positives and false negatives
fp_frac_man = sum(man_thresh_vec_val==1 & true_lb_vec_val==0) / sum(true_lb_vec_val==0);  
fn_frac_man = sum(man_thresh_vec_val==0 & true_lb_vec_val==1) / sum(true_lb_vec_val==1);

% print
disp(['% false positives (manual): ' num2str(100*fp_frac_man)])
So it appears our manual threshold leads to a very high false negative rate: almost 30% of the interior pixels are being misclassified. This phenomenon is clear from a comparison of our classified image with the ground truth labels. The threshold performs very poorly in the center of clusters of bacteria. Let's see if we can improve upon this by using an optimization algorithm to select the threshold.

### Computational thresholding

We will use a logistic regression to estimate the optimal threshold computationally. Logistic regressions are widely used to infer how one or more variables (in our case, pixel value) predict outcomes that our binary in nature—in our case interior or exterior (1 or 0). For our problem, the model takes the following form:

\[
\ln \frac{p_0}{p_1} = \beta_0 + \beta_1 X_i
\]

Here \( p_1 \) and \( p_0 \) indicate the probability that a pixel is inside and outside of a bacterial cell, respectively, and \( X_i \) indicates that pixel's intensity. The object is to infer the regression coefficients (\( \beta_0 \) and \( \beta_1 \)) that do the best job predicting a pixel's class (inside or outside) base upon it's intensity. Matlab has a built-in function for inferring logistic regression parameters: \textit{mnrfit}.

```matlab
class_vec_train = categorical(lb_image(set_assignment_vec==1)); % cast class labels as categorical variables	pixel_values_train = phase_image(set_assignment_vec==1); % convert 2D image into vector
% conduct inference
beta_vec = mnrfit(pixel_values_train,class_vec_train);
```

"beta_vec" contains the estimate values for \( \beta_0 \) and \( \beta_1 \). These estimates are obtained using a common optimization approach known as Maximum Likelihood estimation. The specifics of the optimization routine aren't critical to understanding how to apply logistic regressions to data, so we won't go any further into the specifics here. With these parameter estimates in hand, the next step will be to compare matlab's estimate of the threshold with our manual guess. To find threshold corresponding to the inference, we need to solve for when the ratio between \( p_1 \) and \( p_0 \) is 1:

\[
\ln 1 = \beta_0 + \beta_1 X_c
\]

\[
0 = \beta_0 + \beta_1 X_c
\]

\[
X_c = -\frac{\beta_0}{\beta_1}
\]

We can do this programmatically:

```matlab
thresholdComputational = - beta_vec(1) / beta_vec(2);
```
Now let's compare the computational and manual thresholds:

```matlab
cp_hist_fig = figure; hold on
histogram(inside_pix_vec,edges,'Normalization','probability','FaceColor',plot_cmap(1,:))
histogram(outside_pix_vec,edges,'Normalization','probability','FaceColor',plot_cmap(2,:))
xlabel('pixel intensity')
ylabel('number of pixels')

% add ref lines
line([thresholdManual,thresholdManual],ylim,'Color','black','LineWidth',2)
line([thresholdComputational,thresholdComputational],ylim,'Color','blue','LineWidth',2)
legend('interior', 'exterior','manual threshold','computational threshold')
xlim([0 .8])
box on
grid on

set(gca,'Fontsize',12)
```

Unless you went completely rogue in your manual estimate, you should find that, visually at least, it appears that the algorithm matches the manual guess quite closely. Let's compare the manually and computationally classified images.

```matlab

% generate classified image using computational threshold
imThreshComputational = phase_image < thresholdComputational;

% display computationally classified image
comp_fig = figure;
isshow(imThreshComputational,'InitialMagnification','fit')

% force image to be square
ax = gca;
ax.DataAspectRatio = [1 1 1];
title('Computational Threshold')
saveas(comp_fig,[figPath 'classified_image_1param.tif'])
```
Let's now perform a more quantitative comparison. Start by calculating the true and false positive fractions.

% extract validation vectors
comp_thresh_vec_val = imThreshComputational(set_assignment_vec==2);
true_lb_vec_val = lb_image(set_assignment_vec==2);

% calculate number of false positives and false negatives
fp_frac_comp = sum(comp_thresh_vec_val==1 & true_lb_vec_val==0) / sum(true_lb_vec_val==0);
fn_frac_comp = sum(comp_thresh_vec_val==0 & true_lb_vec_val==1) / sum(true_lb_vec_val==1);

% print
disp(['% false positives (computational): ' num2str(100*fp_frac_comp)]);

% false positives (computational): 5.0997

disp(['% false negatives (computational): ' num2str(100*fn_frac_comp)]);

% false negatives (computational): 26.5778

Now that we've quantified the performance of the manual and computational thresholds, it makes sense to ask how close they are to the globally optimal solution. Since we're only dealing with a single parameter at this point, it is trivial to calculate the true and false positive rates as a function of threshold, in order to see where our estimates fall.

% calculate error rate as a function of decision threshold for validation image
thresh_vec = linspace(0,1,1000); % vector of possible threshold values
fp_vec_val = NaN(size(thresh_vec)); % vector of false positive fractions
fn_vec_val = NaN(size(thresh_vec)); % vector of false negative fractions
for i = 1:numel(thresh_vec)
    lb_vec_temp = phase_image(set_assignment_vec==2)<thresh_vec(i);
    fp_vec_val(i) = sum(lb_vec_temp==1 & true_lb_vec_val==0) / sum(true_lb_vec_val==0);
    fn_vec_val(i) = sum(lb_vec_temp==0 & true_lb_vec_val==1) / sum(true_lb_vec_val==1);
end

% make plot
error_fig = figure;
colormap(cmap_im)
hold on
plot(100*fp_vec_val,100*fn_vec_val,'Color','black')
scatter(100*fp_frac_man,100*fn_frac_man,'filled','MarkerEdgeAlpha',0)
scatter(100*fp_frac_comp,100*fn_frac_comp,'filled','MarkerEdgeAlpha',0)
xlabel('false positive rate (%)')
ylabel('false negative rate (%)')
grid on
legend('error trend','manual solution', 'computational solution','Location','southeast')
title('Error rate vs. Decision Threshold (validation set)')
% flip y axis
set(gca, 'YDir','reverse')
axis([0 60 0 60])
set(gca,'Fontsize',12)

We see that the computational solution decreases the false positive rate, but at the cost of significantly increasing the rate of false negatives. The dynamics of this trade-off between false negatives and false positives will vary depending on our application. Regardless, it is clear that there is an inherent limit to the performance of any classification scheme that uses pixel intensity alone. Graphically speaking, there is a lot of space in the upper left hand corner of the trade-off space that we cannot access due to the significant overlap in pixel intensities between classes.

A natural next step is to move from one classification axis (intensity) to two or more predictors. And while we could make a respectable guess at the optimal threshold for a single parameter, the value of having a computational procedure for doing this will become clear as the number of variables increases.

Part II: Multi-parametric image segmentation

Introduction to image filters

It is possible to extract useful information about the structure and content of an image by observing how it changes when we subject it to different kinds of mathematical transformations. These transformations are applied by passing a small matrix, a filtering kernel, over the image. It is not our purpose here to give a detailed account of the mechanics of image filtering. For us, what matters is that applying various filters to an image provide additional information that can be used to augment our simple threshold-based classifier.

The best way to get a sense for how these filtering kernels work is to apply them to an image. In order to augment the information provided by raw pixel intensities, it seems intuitive to employ some metric that
captures information about the shape of the objects depicted in the image. The Laplace Filter commonly used for edge detection in image analysis. Let's see what happens when we apply it to our training image.

```matlab
% create Gaussian kernel
h_Laplace = fspecial('laplacian',.2);
% visualize
kernel_fig = figure;
colormap(cmap_im)
imagesc(h_Laplace);
h = colorbar;
ylabel(h, 'pixel value')
title('Laplace filter')
hold off

% apply it to image
Laplace_image = imfilter(phase_image,h_Laplace);
% compare
raw_fig = figure;
colormap(cmap_im)
imagesc(phase_image)
h = colorbar;
ylabel(h, 'pixel value')
% force image to be square
ax = gca;
ax.DataAspectRatio = [1 1 1];
title('Original')

lap_gif = figure;
colormap(cmap_im)
imagesc(Laplace_image)
% force image to be square
ax = gca;
ax.DataAspectRatio = [1 1 1];
title('Filtered (Laplace)')
h = colorbar;
ylabel(h, 'pixel value')
caxis([-0.05 .05])
```
Note how the Laplace filter emphasizes the boundaries of cells. Intuitively, it seems like this effect might be useful for classification; however, we can also see that the filter amplifies background noise in the image. In order to circumvent this problem, it is common practice to "smooth" the image using a Gaussian averaging kernel prior to applying the Laplace filter. This is known as Laplace of Gaussian (LoG) filtering. Let’s first see what the Gaussian kernel does on its own.

```matlab
% create Gaussian kernel
hsize = 5; % dimensions of kernel
sigma = 2; % sigma of gaussian (in pixels)
h_Gauss = fspecial('gaussian',hsize,sigma);
% visualize
kernel_fig = figure;
colormap(cmap_im)
imagesc(h_Gauss);
h = colorbar;
ylabel(h,'pixel value')
title('Gaussian Smoothing Kernel')
```
% apply it to image
Gauss_image = imfilter(phase_image,h_Gauss);

Gauss_fig = figure;
colormap(cmap_im)
imagesc(Gauss_image)
title('Filtered (Gaussian)')
% force image to be square
ax = gca;
ax.DataAspectRatio = [1 1 1];
h = colorbar;
ylabel(h,'pixel value')
As expected, Gaussian filtering returns a blurred version of the original image. Now we will combine the Gaussian and Laplace kernels.

```matlab
% use a slightly smaller Gaussian kernel for milder blurring
hsize = 5; % dimensions of kernel
sigma = 2; % sigma of gaussian (in pixels)
h_Gauss = fspecial('gaussian',hsize,sigma);
Gauss_image = imfilter(phase_image,h_Gauss);
% apply Laplace kernel to Gauss-smoothed image
LoG_image = imfilter(Gauss_image,h_Laplace);

Laplace_fig = figure;
colormap(cmap_im);
imagesc(Laplace_image)
title('Laplace')
h = colorbar;
ylabel(h,'pixel value')
caxis([-0.05,0.05])
% force image to be square
ax = gca;
ax.DataAspectRatio = [1 1 1];
```
LoG_fig = figure;
colormap(cmap_im);
imagesc(LoG_image)
title('Laplace of Gaussian (LoG)')
% force image to be square
ax = gca;
ax.DataAspectRatio = [1 1 1];
h = colorbar;
ylabel(h,'pixel value')
caxis([-0.05,.05])
set(gca,'Fontsize',12)
saveas(LoG_fig,[figPath 'LoG_ft_image.tif'])
This looks better. The LoG filter clearly returns a less noisy image, while preserving the boundary-amplifying properties of the Laplace filter. Now let's see if we can but it to use.

**Classification with multiple parameters**

Let's first get a sense for how our two metrics, raw and LoG pixel intensities, correspond to our two classes.

```matlab
% generate vector of labels
lb_vec = lb_image(:);

class_fig2 = figure;
colormap(plot_cmap)
hold on
scatter(phase_image(lb_vec==0&set_assignment_vec==1),LoG_image(lb_vec==0&set_assignment_vec==1),20,...
'MarkerFaceColor',plot_cmap(1,:),'MarkerFaceAlpha',.1,'MarkerEdgeAlpha',0)
scatter(phase_image(lb_vec==1&set_assignment_vec==1),LoG_image(lb_vec==1&set_assignment_vec==1),20,...
'MarkerFaceColor',plot_cmap(2,:),'MarkerFaceAlpha',.1,'MarkerEdgeAlpha',0)
legend('exterior','interior')
xlabel('raw intensity')
ylabel('LoG')
title('2 Parameter Scatter (training set)')
grid on
box on
set(gca,'Fontsize',12)
saveas(class_fig2,[figPath 'two_param_scatter.tif'])
```
We can see that, in addition to their relative separation along the "raw intensity" axis, interior and exterior pixels also separate to a degree along the LoG axis. This suggests that adding the LoG dimension should improve our segmentation. Let's give it a try. The procedure is essentially the same as for a single parameter.

```matlab
% "Response" vector
class_vec_train = categorical(lb_image(set_assignment_vec==1))'; % cast labels as categorical variables
% Predictor vectors
raw_pixel_values = phase_image(set_assignment_vec==1); % convert 2D image into vector
LoG_pixel_values = LoG_image(set_assignment_vec==1); % convert 2D image into vector
pd_mat = [raw_pixel_values' LoG_pixel_values']; % concatenate predictor variables
% conduct inference
beta_vec_2param = mnrfit(pd_mat,class_vec_train); % conduct inference
```

As before, we can calculate the inferred decision threshold--this time a line across 2D parameter space--from the inferred $\beta$ parameters.

\[
0 = \beta_0 + \beta_1 X_{1c} + \beta_2 X_{2c}
\]

\[
X_{1c} = -\frac{\beta_0 + \beta_2 X_{2c}}{\beta_1}
\]

Let's plot this decision line to see how it intersects with the raw data.

```matlab
% generate vectors of x1 and x2
LoG_vec = linspace(min(LoG_image(:)),max(LoG_image(:)));
```
raw_vec = - (beta_vec_2param(1) + beta_vec_2param(3) * LoG_vec) ./ beta_vec_2param(2);

% plot
class_line_fig2 = figure;
colormap(plot_cmap)
hold on
scatter(phase_image(lb_image==0),LoG_image(lb_image==0),20,'MarkerFaceColor',plot_cmap(1,:), 'MarkerFaceAlpha', 0.1, 'MarkerEdgeAlpha', 0)
scatter(phase_image(lb_image==1),LoG_image(lb_image==1),20,'MarkerFaceColor',plot_cmap(2,:), 'MarkerFaceAlpha', 0.1, 'MarkerEdgeAlpha', 0)
plot(raw_vec, LoG_vec,'Color','black','LineWidth',2)
legend('outside','inside','decision line')
xlabel('raw intensity')
ylabel('LoG')
title('2 Parameter Scatter (training image)')
xlim([0 1])
grid on
saveas(class_line_fig2,[figPath 'two_param_scatter_line.tif'])
set(gca,'Fontsize',12)

Looks reasonable. Now let's see if adding the LoG filter actually leads to an appreciable improvement in our segmentation. To do this, we need to generate a classified image by determining whether each pixel falls above or below the decision line. Essentially, if \( \beta_0 + \beta_1 X_{1c} + \beta_2 X_{2c} < 0 \), then \( p_1 > p_0 \) (because the left side of the regression is \( \ln \frac{p_0}{p_1} \)) and we classify the pixel as "inside" (and vice versa).

Let's calculate the true and false positive rates and compare them to those achieved using the threshold alone.

% calculate thresholded image
imThresh2Param = (beta_vec_2param(1) + beta_vec_2param(2) * ... 
        phase_image + beta_vec_2param(3) * LoG_image) < 0;

% extract validation vectors
thresh_vec_val_2param = imThresh2Param(set_assignment_vec==2);
true_lb_vec_val = lb_image(set_assignment_vec==2);

% calculate number of false positives and false negatives
fp_frac_param2 = sum(thresh_vec_val_2param==1 & true_lb_vec_val==0) / sum(true_lb_vec_val==0);
fn_frac_param2 = sum(thresh_vec_val_2param==0 & true_lb_vec_val==1) / sum(true_lb_vec_val==1);

% print error rates to console
% print
disp(['% false positives (2 parameter): ' num2str(100*fp_frac_param2)])
% false positives (2 parameter): 3.894

disp(['% false negatives (2 parameter): ' num2str(100*fn_frac_param2)])
% false negatives (2 parameter): 19.4555

% make plot
error_fig = figure;
colormap(cmap_im)
hold on
plot(100*fp_vec_val,100*fn_vec_val,'Color','black')
scatter(100*fp_frac_man,100*fn_frac_man,'filled','MarkerEdgeAlpha',0)
scatter(100*fp_frac_comp,100*fn_frac_comp,'filled','MarkerEdgeAlpha',0)
scatter(100*fp_frac_param2,100*fn_frac_param2,'filled','MarkerEdgeAlpha',0)
xlabel('false positive rate (%)')
ylabel('false negative rate (%)')
grid on
legend('error trend','manual threshold', 'computational threhold','2 parameter fit', 'Location','southeast')
title('Error rate vs. Decision Threshold (validation set)')
% flip y axis
set(gca, 'YDir','reverse')
xlim([0 20])
ylim([0 30])
Great! We’ve managed to drop our error rate to a respectable 8%. Nonetheless, this result may still leave something to be desired for certain applications. In developing a classifier, it is important to know how good is “good enough”. This will vary widely depending on the application. Often, it is a mistake to chase the “perfect” classification scheme, when one that is right most of the time would be perfectly sufficient.

Finally, let’s compare our latest segmented image with the ground truth.

```python
param2_comparison_fig = figure;
imshow(imThresh2Param,'InitialMagnification','fit')
title('Classified Image (2 param model)')
saveas(param2_comparison_fig,[figPath 'classified_image_2param.tif'])
```
raw_fig = figure;
imshow(lb_image,'InitialMagnification','fit')
title('Ground Truth')
% make some additional filters

% Gaussian Blur
GB1 = imgaussfilt(phase_image,1);
GB3 = imgaussfilt(phase_image,3);
GB5 = imgaussfilt(phase_image,5);
GB7 = imgaussfilt(phase_image,7);

% Difference of Gaussian
DoG15 = GB5 - GB1;
DoG37 = GB7 - GB3;

% generate first and second derivativ images
imDGB1 = imgradient(GB1);
imDGB5 = imgradient(GB5);

% Now generate array of predictors
tr_filt = set_assignment_vec==1;
pd_mat_full = [phase_image(:) LoG_image(:) ...
                GB1(:) GB3(:) GB5(:) GB7(:) DoG15(:) DoG37(:) imDGB1(:) imDGB5(:)];

NumTrees = 50; % set number of decision trees
% instruct matlab to train logistic regression classifier
beta_vec_full = mnrfit(pd_mat_full(tr_filt,:),class_vec_train);

Warning: Maximum likelihood estimation did not converge. Iteration limit
exceeded. You may need to merge categories to increase observed counts.

% instructs matlab to train ensemble of trees
RFMdl = TreeBagger(NumTrees,pd_mat_full(tr_filt,:),class_vec_train);

logfile = mnrval(beta_vec_full,pd_mat_full);
[~, rf_class_prob_array] = predict(RFMdl,pd_mat_full);
% class prob array contains 1 row per pixel, on column per class
% reshape predictions into image dims
imLogProb = reshape(log_class_prob_array(:,1),size(phase_image));
imRFProb = reshape(rf_class_prob_array(:,1),size(phase_image));
imThreshLog = imLogProb < .5;
imThreshRF = imRFProb < .5;
% calculate the false positive and negative rates
thresh_vec_val_RF = rf_class_prob_array(set_assignment_vec==2,1)' < .5;
thresh_vec_val_Log = log_class_prob_array(set_assignment_vec==2,1)' < .5;

fp_frac_RF = sum(thresh_vec_val_RF==1 & true_lb_vec_val==0) / sum(true_lb_vec_val==0);
fn_frac_RF = sum(thresh_vec_val_RF==0 & true_lb_vec_val==1) / sum(true_lb_vec_val==1);

fp_frac_log = sum(thresh_vec_val_Log==1 & true_lb_vec_val==0) / sum(true_lb_vec_val==0);
fn_frac_log = sum(thresh_vec_val_Log==0 & true_lb_vec_val==1) / sum(true_lb_vec_val==1);

% make new iteration of fp fn figure

% make plot
error_fig = figure;
colormap(cmap_im)
hold on
plot(100*fp_vec_val,100*fn_vec_val,'Color','black')
scatter(100*fp_frac_man,100*fn_frac_man,'filled','MarkerEdgeAlpha',0)
scatter(100*fp_frac_comp,100*fn_frac_comp,'filled','MarkerEdgeAlpha',0)
scatter(100*fp_frac_param2,100*fn_frac_param2,'filled','MarkerEdgeAlpha',0)
scatter(100*fp_frac_log,100*fn_frac_log,'filled','MarkerEdgeAlpha',0)
scatter(100*fp_frac_RF,100*fn_frac_RF,'filled','MarkerEdgeAlpha',0)
xlabel('false positive rate (%)')
ylabel('false negative rate (%)')
grid on
legend('error trend','manual threshold', 'computational threshold','2 parameter fit',...
    'logistic (all params)', 'random forest (all params)', 'Location','southeast')
title('Error rate')
% flip y axis
set(gca, 'YDir','reverse')
xlim([0 20])
ylim([0 30])
param2_fig = figure;
imshow(imThresh2Param,'InitialMagnification','fit')
% title('2 Parameter Logistic Regression')

rf_fig = figure;
imshow(imThreshRF,'InitialMagnification','fit')
% title('Multiparametric Random Forest')
saveas(rf_fig,[figPath 'multi_rf_classified.tif'])
true_fig = figure;
imshow(lb_image,'InitialMagnification','fit')
% title('Ground Truth')