**Functions**

Import\_functions

1. get(health,technique,heart\_mode,param,mask='myo'):

* Pulls data from csv files

Inputs:

* health: 'Healthy' or 'HCM'
* technique: 'SE' or 'STEAM'
* heart\_mode: 'Systole' or 'Diastole'
* param: params2d = ['CR', 'E2A', 'E2A\_alt', 'E3A', 'FA', 'HA', 'IA', 'MD', 'mode', 'negative\_eigenvalues', 'norm', 'S0', 'TA']
* mask: any of the masks available in the mask folders

Outputs:

* Returns: list with two entries. The first entry is a list of the images, the second entry is a list of the masks
* Dont unpack the output, if you see a function has a data input, it means you need to pass in this list of two entries

1. rotate\_imgs\_auto(data)

* Automatically rotating images

Input: list of images and masks

Output: list of images and masks rotated

1. mask(data)

* Masks the images with the mask obtained from get

Input: list of images and masks

Output: list of masked images and masks

1. crop(data)

* Crops the images and masks by removing all columns and rows that contain only 0's

Input: list of images and masks

Outputs: list of cropped images and masks

1. pp(data)

* Pre-Processing: Rotates, Masks, Crops the input images and masks

Input: list of images and masks

* Output: list of rotated, masked, cropped images and masks

Helper\_functions

1. helper\_cmaps(imgs)

* Configures plt.imshow() cmap settings for plotting. Matches the vmax and vmin across all images
* Use like this: plt.imshow(img, \*\*helper\_cmaps(imgs))
* Or if single image: plt.imshow(img, \*\*helper\_cmaps([img]))

Input: List of images

1. pims(imgs,title=None,figsize = (10,6))

* Plotting Helper Function for plotting multiple figures at once
* Plots up to 15 images from a list

Input:

* imgs: list of images
* title: title of plot
* figsize: figure size of plot

1. to\_dist(imgs)

* Converts a list of images to a distribution

Inputs: list of images

Outputs: 1d distribution of the values in the image

1. plot\_dists(dist\_h,dist\_uh,bin\_n,labels=['Healthy','HCM'],ax = None)

* Plot Distributions obtained from to\_dist(imgs)

Inputs:

* dist\_h,dist\_uh: the two distributions you want to plot. (N,) Numpy array of pixel values
* bin\_n: number of bins to use in histogram
* labels: labels assigned to dist\_h,dist\_uh respectively
* ax: axis to plot to, if no axis set, new axis created

1. plot\_ims(set\_nan=True, \*args)

* Plotting function helper

Inputs:

* set\_nan: sets the 0 values in the image to nan so no background is present.
* args: list of images

unwrap\_functions

1. pick\_centre\_mean\_(alg\_mask)

* Calculates the centre of the image using the centre of mass

1. pick\_centre\_middle\_(alg\_mask)

* Calculates the centre of the image by taking the central point

1. unwrap\_inner\_(img1,mask1,pick\_centre\_algorithm=pick\_centre\_mean\_)

* Computes the inner unwrap of an image

Output:

* list of numpy arrays containing information on the pixels in each layer. Each numpy array is of size Nx4.
* N: number of points in layer i, ouput[i]
* 4: holds the pixel value, pixel angle, pixel i and j coordinate

1. get\_contour\_values(alg\_mask,bordered\_mask,bordered\_imag,angles)
2. add\_diagonal\_elements\_contour\_mask(contour\_mask)
3. unwrap\_outer\_(img1,mask1,pick\_centre\_algorithm = pick\_centre\_mean\_)

* Computes the outer unwrap of an image

Output:

* list of numpy arrays containing information on the pixels in each layer. Each numpy array is of size Nx4.
* N: number of points in layer i, ouput[i]
* 4: holds the pixel value, pixel angle, pixel i and j coordinate
* this is referred to as contour\_values frequently

1. correct\_end\_vals\_(out)

* Correction for the number of layers in the image
* algorithm responsible for correcting the layers if the last layers have too few values in them

1. correct\_N\_(inn,out)

* get the Number of layers estimated in an image after correction has been applied

1. get\_val\_layer\_(i,j,layers)

* returns the pixel value and angle of a pixel at position i,j in the original image from the list of numpy arrays

1. vals\_to\_layer\_map\_(image,inn,out,custom\_tol=None)

* This takes the list of numpy arrays that contain the layer information and builds the final depth layer map

1. layer\_map\_to\_contour\_values\_(image,layer\_map,inn)

* from the final layer\_map, calculates the contour\_values from the layer\_map and image

1. vals\_to\_barcode\_updated\_(contour\_values,tol)

* converts the contour\_values into a barcode Method 1

1. vals\_to\_barcode\_updated\_method2\_(contour\_values,N=None)

* converts the contour\_values into a barcode Method 2

1. get\_inner\_outer\_unwraps(data)

* Returns the inner and outer unwraps of the data

Input: data: list of images and masks

Output: oups\_inner,oups\_outer: returns the contour\_values of the inner and outer unwrap

1. uw2(data,num\_M=None,num\_layers=None,return\_contours=False,return\_layer\_maps=False)

* this is the main unwrap function responsible for most of the work. It takes a bit long

on a large amount of images so I recommend unwrapping before doing heavy computations.

Or use jupyter notebook.

Input:

* data: list of images and masks
* num\_M: Number of points in the circumfrential direction of barcode. If left blank, will take the value of the largest
* by pixel count. Doesnt need to be set if return\_contours or return\_layer\_maps == True
* num\_layers: number of layers to have in the final image. set to 3 to retrieve inner, middle, outer sections
* if left as None, will automatically select it based on the corrected lengths of the inner and outer unwrap
* return\_contours: returns the contour\_values of the image after final depth calculation
* return\_layer\_maps: returns the layer\_maps of the image
* if both return\_contours and return\_layer\_maps is set to False, the unwrapped barcode is returned

1. get\_layer\_maps(data,num\_layers=None)

* Retrieves the layer\_maps of all images in data

Input:

* data: list of images and masks
* num\_layers: number of layers to have in the final image. set to 3 to retrieve inner, middle, outer sections
* if left as None, will automatically select it based on the corrected lengths of the inner and outer unwrap

Output: oups

1. get\_num\_layers(data)

* returns the number of layers in an image after correction. Use for approximating size

Inputs: data: list of images and masks

Outputs: Number of layers in each image

1. get\_angles1\_(img1,mask1,pick\_centre\_algorithm=pick\_centre\_mean\_)

* returns angle maps of the image

1. get\_angle\_maps(data,minn=None,maxx=None)

* returns the angle maps of all images in data, can set minn and maxx to only select specific regions.(120,300 roughly for septum)

Inputs:

* data: list of images and masks
* minn,maxx: minimum and maximum angles to retrieve

1. calculate\_average\_angle(imgs)

* given a set of images, calculates the periodic average angle in it used for Figure of diastolic and systolic E2A in septum

stats\_functions

The first four functions in this file are responsible for computing which distribution an image matches better

They have a general format that needs to be followed to be able to include your own.

So the general format is for inputs:

img: the input image, can be a distribution or a 2d image

health: the health of this image. 0: Healthy 1: HCM

h\_dist: healthy distribution to compare to

uh\_dist: unhealthy distribution to compare to

n\_bins: number of bins to use if needed

then the ouputs are: [oup1,oup2]

oup2 is always the prediction of the model. 0:healthy 1:HCM

oup1 has a set format. The first entry is 1 if the prediction is correct. The last entry is always 1 to count the number of times

the image was used in the dataset. All entries in the middle handle any test statistics that want to be stored per image

1. log\_like(img,health,h\_dist,uh\_dist,n\_bins)
2. custom\_r2(img,health,h\_dist,uh\_dist,n\_bins)
3. mann\_whitney(img,health,h\_dist,uh\_dist)
4. calculate\_zscore(imgs,dist,U)
5. KS2(img,health,h\_dist,uh\_dist)
6. run\_all\_tests\_loocv(imgs\_h,imgs\_uh,bin\_n=30)

* This is the main function used to run LOOCV statistical tests on each image in imgs\_h and imgs\_uh

Input:

* imgs\_h,imgs\_uh = list of healthy and unhealthy images
* bin\_n = number of bins for LL and R2 test

1. run\_all\_tests\_kfold(imgs\_h,imgs\_uh,test\_size,num\_runs,bin\_n)

* Similar run\_all\_tests\_loocv but selects multiple individuals for the test set.

Input:

* imgs\_h,imgs\_uh: list of healthy and HCM images
* test\_size: Number of individuals to put into test set [healthy\_test\_number,HCM\_test\_number]
* num\_runs: how many times to rerun the train-test split and statistical test workflow
* bin\_n: number of bins for LL and R2 test

1. set\_acc(arrs)
2. get\_performance\_metrics(df)
3. get\_metrics(output,test,pri=None)

local\_SD\_analysis\_functions

1. local\_SD\_(window,min\_points)

* calculates the non-periodic standard deviation of a window in the image

Inputs: min\_points: minimum number of points required in the window to calculate the SD, if less, oup is set to 0

1. circular\_SD\_(window,min\_points)

* calculates the periodic standard deviation of a window in the image

Inputs: min\_points: minimum number of points required in the window to calculate the SD, if less, oup is set to 0

1. image\_SD\_calc\_function\_(img,mask,n,periodic\_var=False,min\_num\_points=5)

* calculates the local SD over a single image

1. calc\_SD(data, n, periodic\_SD=False,min\_num\_points=5)

* calculates the local variance over all images in data

Inputs:

* data: list of images and masks
* n: size of filter (nxn)
* periodic\_var: bool, if True, will calculate the periodic SD, if False, will calculate the classic SD
* min\_num\_points=5: minimum number of pixels required in a filter to compute the SD.

Cross Correlation Functions

1. group\_unwraps\_(countour\_value)

* helper\_function for when youre working with the inner, middle outer region. The default uw2 method returns the contour\_values output as a list of Nx3, where N is the number of images and 3 is the number of regions. This just converts it to a 3XN list so that extracting all images for a particular region can be done through oup[i]

Input: contour\_value: contour\_value output from uw2 function

1. split\_unwraps(countour\_value,test\_idx)

* Split the unwraps into train and a singular test image for LOOCV

1. cross\_correlate\_images(img1,img2,change\_angles,filter\_settings=None,plot\_vals=False)

Note when the contour\_values are talked about, it may seem difficult to retrieve them

They are ultimately just a numpy array where each row is a pixel\_value / angle pair you can just use the unwrap algorithm and set it the number of layer to 1 to retrieve it for the full images or you can unwrap into multiple regions and then split\_unwraps to retrieve the training or testing data to use for cross correlation

This is the cross correlation of two images. Only really applicable to E2A or IA parameter

img1: contour\_values of first image to compare: generally put systole here

img2: contour\_values of second image to compare: generally put diastole here

change\_angles: [bool1,bool2], changes the range of the angle parameters of [img1,img2]

respectively from -90-90 to 0-90

filter\_settings: savgol filter was implemented but not mentioned in report, can be used to

smoothen the distributions in the columnwise direction

filter\_settings = [window\_length,polyorder]

plot\_vals: if False, doesn't plot anything. If given a value, will plot the:

histograms of the distributions of img1,img2

2d PDFs of img1 and img2

Angle shift over the circumfrential position of image

Titles are set to the value of plot\_vals

Returns: (hist1,hist2),(oup1,oup2,oup3,(x,angle\_shifts))

hist1, hist2: 2D histograms of img1 and img2

oup1: full columnwise crosscorrelation

oup2: only maximum values of cross correllations

oup3: maximum values but set to 1

x: x positions of angle\_shift

angle\_shifts: angle\_shift at each discrete point along the circumfrential position

AI\_methods

1. gen\_data\_split(healthy\_images,unhealthy\_images,test\_size,repeats=False)

* splits data for training

Input:

* healthy\_images,unhealthy\_images: list of images
* test\_size: how many images to take from both healthy and hcm for testing
* repeats: how many times to repeat the unhealthy training images to account for the class imbalance

Output:

* a lists of training and testing data.
* y data is categorical, meaning the output is [0,1] or [1,0] rather than [0] or [1]

1. normalise(healthy\_images,unhealthy\_images,minn=-90,maxx=90)

* normalises values to range of -1 to 1 for each image. If angle parameter, dont pass values into minn or maxx
* If you want it to be automtically selected, set minn,maxx == None

Input:

* healthy\_images,unhealthy\_images: list of healthy and HCM images
* minn,maxx: values in the original images mapped to -1 and 1

1. standardise(healthy\_images,unhealthy\_images)

* standardises images using zscore formula

1. resize\_images(X\_train,X\_test)

* Resizes the images to a constant size. The largest image had a dimension of 63, so images were chosen to be resized to 64 by 64.

Inputs: X\_train,X\_test: list of input data.

Outputs: new\_X\_train,new\_X\_test: numpy arrays of size Mx64x64x1, where M is the number of individuals in training and test set

1. build\_model\_cnn(patience,start\_from)

Inputs:

* patience: how many epochs the model waits before stopping if it hasnt reached a new minimum of validation loss
* start\_from: the number of epochs to wait before starting early stopping criterion

Outputs:

* model: Compiled TensorFlow Model
* callback: Callback to use during training

1. train\_model(model, callback, data, epochs)

* trains model and evaluates

Inputs:

* model: Compiled TensorFlow Model
* callback: Callback to use during training
* data: [X\_train,y\_train,X\_test,y\_test]
* epochs: number of cycles to train the neural network. Effectively how many times it sees the data.

Outputs:

* history: Training and Validation Loss and Accuracy
* info\_df: Information dataframe containing final test and train losses and accuracies

1. plotHistory(history)

* pass in history from run\_model to plot accuracy and loss vs epoch for training and validation

1. save\_model(name,model,data,history)

* saves the model, training and validation data, and training history

Inputs:

* name: name of model to save. Will be save in ./Models/name/
* model: TensorFlow model to save
* data: training and test data used
* history: training and validation accuracy and loss during training

1. load\_model(name)

* loads the model, training and validation data, and training history

Input: name: name of model to load. Will be loaded from ./Models/name/

Output:

* model: TensorFlow model to load
* data: training and test data used to train model
* history: training and validation accuracy and loss during training

Modules that need to be downloaded:

* pickle
* numpy
* tensorflow (NB: use pip instead of conda for keras)
* sklearn
* PIL
* skimage