In [ ]:

```python
import numpy as np
from numpy.typing import NDArray
import nibabel as nib
import os
import scipy.io
from scipy.optimize import minimize
from tqdm import tqdm

rng = np.random.default_rng(seed=42)


SUBJECT_IDS_REMOVE = np.array([43607, 44631, 74067, 78004, 21511, 98394, 67358, 17(
# subject id 43607, error on loading: True, shape: (0,)
# subject id 44631, error on loading: False, shape: (96, 96, 51, 8)
# subject id 74067, error on loading: False, shape: (96, 96, 51, 2)
# subject id 78004, error on loading: False, shape: (96, 96, 51, 7)
# subject id 21522 and 98394 aren't extreme pre term, nor full term
# subject ids: [67358, 48996, 17065] have high nonmonotonic percentages , 67358, 1


def T2_estimation(img1: NDArray, img2: NDArray, mask: NDArray, TE_1: float, TE_2:
    """estimate T2 directly using 2 signal images from 2 different TEs

    Args:
        img1 (NDarray): any shape, but same as other inputs
        img2 (NDarray): any shape, but same as other inputs
        mask (NDArray): any shape, but same as other inputs
        TE_1 (float): TE timing of img1
        TE_2 (float): TE timing of img2

    Return:
        T2 (NDarray): [same shape as inputs]
    """
    assert img1.shape == img2.shape, 'input shapes need to be equal'
    assert mask.shape == img1.shape, 'input shapes need to be equal'

    T2 = (TE_2 - TE_1) / np.log(img1 / img2)
    T2 = mask * T2
    T2 = np.where(np.isnan(T2), 0, T2)

    return T2


def lsqr_weighted(data: NDArray, mask: NDArray, TE: NDArray):
    """calculate T2 and S0 from the signal images of single slice through multiple
    assumes there is only one T2 compartment

    Args:
        imgs (NDarray): [...,t]
        mask (NDArray): [...]
        TE (NDarray): [t] 1D array

    Return:
        T2 (NDarray): [...]
        S0 (NDarray): [...]
    """
    assert TE.shape[0] == data.shape[-1], 'number of TE timings needs to be the sar

    eps = 1
    data_shape = data.shape[:-1]
    data_flat = data.reshape((-1,data.shape[-1]))
    mask_flat = mask.reshape((-1))
```

```
    nb_data = data_flat.shape[0]
    TE_num =  TE.shape[0]
    X = np.zeros(shape=(nb_data, 2))

    G = np.ones(shape=(TE_num, 2))
    G[:,1] = TE
    for i in range(nb_data):
        A = data_flat[i,:]
        if A.min() > eps and mask_flat[i] > 0:
            W = np.diag(A**2)
            invmap = np.linalg.pinv(G.T @ W @ G) @ G.T @ W
            X[i,:] = invmap @ np.log(A)

    T2 = (-1) / X[:,1]
    T2 = np.where(T2 > 0, T2, 0).reshape(data_shape)
    S0 = np.exp(X[:,0]).reshape(data_shape)
    return T2, S0


def model_one_compartment(T2: NDArray, S0: NDArray, TE: NDArray) -> NDArray:
    """calculate the signal assuming 1 compartment model

    Args:
        T2 (NDArray): any size of array, T2 at each cell
        S0 (NDArray): any size of array, S0 at each cell
        TE (NDArray): 1D array of TE values

    Return:
        signal (NDArray): array of model estimations same size as T2 image but with
    """
    assert T2.shape == S0.shape, 'parameter input shapes T2 and S0 need to be the s

    signal = np.zeros(shape=(T2.shape))
    T2_inv = 1/T2
    # TE: [t] - TE times, 1D array
    # T2: [h,w,d] - T2 value at every voxel compartment
    # S0: [h,w,d] - S0 value at each voxel
    TE_div_T2 = np.einsum('t,...->...t', TE, T2_inv)    # [h,w,d,t]
    TE_div_T2_exp = np.exp(-TE_div_T2) # [h,w,d,t]
    signal = np.einsum('...,...t->...t', S0, TE_div_T2_exp) # [h,w,d] * [h,w,d,t]
    return signal


def model_one_compartment_voxel(T2: float, S0: float, TE: NDArray) -> float:
    T2 = np.array([T2]).reshape(1,1)
    S0 = np.array([S0])

    signal = model_multi_compartment(T2, S0, TE).flatten()
    return signal


def model_multi_compartment(T2: NDArray, S0: NDArray, TE: NDArray, v: NDArray=np.ar
    """calculate the signal with a multi compartment model

    Args:
        T2 (NDArray): [...,v] any size of array, T2 at each cell compartment
        S0 (NDArray): [...] any size of array, S0 at each cell
        v (NDArray): [...,v] proportion of each compartment at each voxel
        TE (NDArray): [t] 1D array of TE values

    Return:
        signal (NDArray): [...,t] array of model estimations same size as T2 image
    """
    # assert T2.shape[:-1] == S0.shape, 'parameter input shapes T2 excluding v dime
```

```
    assert T2.shape == v.shape, 'T2 and v have to be the same shape. There are the
    assert np.isclose(np.prod(v.sum(axis=-1).flatten()), 1), f'every voxel compartr

    signal = np.zeros(shape=(S0.shape))
    T2_inv = 1/T2
    # TE: [t] - TE times, 1D array
    # T2: [h,w,d,v] - T2 value at every voxel compartment
    # S0: [h,w,d] - S0 value at each voxel
    # v: [h,w,d,v] - compartment split at each voxel
    TE_div_T2 = np.einsum('t,...->...t', TE, T2_inv)    # [h,w,d,v,t]
    TE_div_T2_exp = np.exp(-TE_div_T2) # [h,w,d,v,t]
    signal_per_compartment = np.einsum('...,...vt->...vt', S0, TE_div_T2_exp) # [h
    signal = np.einsum('...vt,...v->...t', signal_per_compartment, v) # [h,w,d,v,t
    return signal


def RMSE(signal_actual: NDArray, signal_estimate: NDArray, dim: int=-1) -> NDArray
    """Calculate the Root Mean Square error between the true value and estimate val
    Assumes the errors are normal so cannot be transformed to log-scale
    Give dimension (dim) that estimates for the same parameters are on, eg. dim=2 t

    Args:
        signal_actual (NDArray): True signal values
        signal_estimate (NDArray): Estimated signal values from model
        dim (_type_, optional): Dimension of time in signal. Defaults to 3:int.

    Returns:
        RMSE (NDArray): float at each voxel with error
    """
    assert signal_actual.shape == signal_estimate.shape, 'estimate and actual shape

    squared_diff = (signal_actual - signal_estimate)**2

    # calculate the number of voxels excluding the t dimension
    n = np.prod(signal_actual.shape[-1])
    RMSE = np.sqrt(np.sum(squared_diff, axis=dim) / n)
    return RMSE


def SSD(signal_actual: NDArray, signal_estimate: NDArray, dim: int=-1, is_normalis
    """Calculate the sum of squared difference between the true value and estimate
    Assumes the errors are normal so cannot be transformed to log-scale

    Args:
        signal_actual (NDArray): True signal values
        signal_estimate (NDArray): Estimated signal values from model
        dim (_type_, optional): Dimension of time in signal. Defaults to 3:int.
        is_normalised (book): return SSD per voxel if True (default False)

    Returns:
        SSD (NDArray): Total SSD
    """
    assert signal_actual.shape == signal_estimate.shape, 'estimate and actual shape

    squared_diff = (signal_actual - signal_estimate)**2

    # calculate the number of voxels excluding the t dimension
    if is_normalised:
        n = np.prod(signal_actual.shape[-1])
    else:
        n = 1
    SSD = np.sum(squared_diff, axis=dim) / n
    return SSD
```

```python
def minimize_given_problem(problem, X0: NDArray, args_list: list, is_solve_for: ND
    """Given parameters for scipy.optimize.minimize, minimize over all given starti

    Args:
        problem (_type_): _description_
        x0 (NDArray): _description_
        args (list): _description_
        solve_for (NDArray): _description_

    Returns:
        _type_: _description_
    """
    solve_shape = X0.shape[:-1] # the last dimension is for different variables opt
    nb_params = X0.shape[-1]
    nb_args = len(args_list) # number of arguments
    if is_solve_for is None:
        is_solve_for = np.ones(shape=solve_shape)
    assert is_solve_for.shape == solve_shape, 'shape of voxels to solve for needs t

    params = []
    for param_id in range(X0.shape[-1]):
        params.append(np.zeros(shape=solve_shape))

    for index in tqdm(np.ndindex(solve_shape), total=np.prod(solve_shape), ascii=Tr
        if is_solve_for[index]:
            x0 = X0[index]
            args = []
            for arg_id in range(nb_args):
                args.append(args_list[arg_id][index])
            args = tuple(args)
            solution = minimize(**problem, x0=x0, args=args, method='L-BFGS-B')
            for param_id in range(nb_params):
                params[param_id][index] = solution['x'][param_id]

    return params


def objective_one_compartment(x, signal, TE_times):
    """objective used to minimize for one compartment model

    Args:
        x (list): S0, T2
        signal (NDArray): 1D array of signals for each TE
        TE_times (NDArray): 1D array

    Returns:
        SSD (float): estimated signal error to true signal
    """
    S0, T2 = x
    vox_est = model_one_compartment_voxel(T2, S0, TE_times)
    res = SSD(signal_actual=signal, signal_estimate=vox_est, dim=-1)
    return res


def objective_two_compartment(x, signal, TE_times, v):
    """objective used to minimize for one compartment model with v fixed

    Args:
        x (list): S0, T2_0, T2_1
        signal (NDArray): true signal for each TE
        TE_times (NDArray): 1D array of TE times
        v (NDArray): 1D array of v0, v1
```

```
    Returns:
        SSD: estimated signal error to true signal
    """
    S0, T2_0, T2_1 = x
    S0 = np.array([S0])
    T2 = np.array([T2_0, T2_1]).reshape(1,2)
    v = v.reshape(1,2)
    vox_est = model_multi_compartment(T2=T2, S0=S0, TE=TE_times, v=v).flatten()
    res = SSD(signal_actual=signal, signal_estimate=vox_est, dim=-1)
    return res


def objective_three_compartment(x, signal, TE_times, S0, T2_0, T2_1, WM_GM_factor)
    """objective used to minimize for three compartment, only free params are T_2,

    Args:
        x (list): T_2, v_2
        signal (NDArray): true signal for each TE
        TE_times (NDArray): 1D array of TE times
        S0 (float): S0 for original signal
        T2_0 (float): T2 value for first compartment (WM)
        T2_1 (float): T2 value for second compartment (GM)
        WM_GM_ratio (float): 1D array of v0/v1 ratio - between [0,1000]

    Returns:
        SSD: estimated signal error to true signal
    """
    T2_2, v_2 = x
    S0 = np.array([S0])
    T2 = np.array([T2_0, T2_1, T2_2]).reshape(1,3)
    v_1 = WM_GM_factor * (1 - v_2)
    v_0 = 1 - v_1 - v_2
    v = np.array([v_0, v_1, v_2]).reshape(1,3)
    vox_est = model_multi_compartment(T2=T2, S0=S0, TE=TE_times, v=v).flatten()
    res = SSD(signal_actual=signal, signal_estimate=vox_est, dim=-1)
    return res


def objective_four_compartment(x, signal, TE_times, S0, T2_0, T2_1, WM_GM_factor):
    """objective used to minimize for three compartment, only free params are T_2,

    Args:
        x (list): T2_2, T2_3, z, r (z = v_2 + v_3), v2 = r * v_3
        signal (NDArray): true signal for each TE
        TE_times (NDArray): 1D array of TE times
        S0 (float): S0 for original signal
        T2_0 (float): T2 value for first compartment (WM)
        T2_1 (float): T2 value for second compartment (GM)
        WM_GM_ratio (float): 1D array of v0/v1 ratio - between [0,1000]

    Returns:
        SSD: estimated signal error to true signal
    """
    eps = 1e-3
    T2_2, T2_3, z, r = x
    S0 = np.array([S0])
    T2 = np.array([T2_0, T2_1, T2_2, T2_3]).reshape(1,4)
    if WM_GM_factor == 0:
        WM_GM_factor = 1e-3
    v_1 = WM_GM_factor * (1 - z)
    v_0 = (v_1 / WM_GM_factor) * (1 - WM_GM_factor)
    v_2 = r * z
    v_3 = 1 - v_0 - v_1 - v_2
    v = np.array([v_0, v_1, v_2, v_3]).reshape(1,4)
```

```
    vox_est = model_multi_compartment(T2=T2, S0=S0, TE=TE_times, v=v).flatten()
    res = SSD(signal_actual=signal, signal_estimate=vox_est, dim=-1)
    return res


def objective_two_v_compartment(x, signal, TE_times):
    """objective to minimize two compartment model including v0

    Args:
        x (list): S0, T2_0, T2_1, v0
        signal (NDArray): true signal for each TE
        TE_times (NDArray): 1D array of TE times

    Returns:
        SSD: error between true and estimate signal
    """
    S0, T2_0, T2_1, v0 = x
    S0 = np.array([S0])
    T2 = np.array([T2_0, T2_1]).reshape(1,2)
    v = np.array([v0, 1 - v0]).reshape(1,2)
    vox_est = model_multi_compartment(T2=T2, S0=S0, TE=TE_times, v=v).flatten()
    res = SSD(signal_actual=signal, signal_estimate=vox_est, dim=-1)
    return res


def create_problem_to_minimize(problem: str, bounds=None):
    if problem == 'one_compartment':
        if bounds is None:
            bounds = [(0, 15000), (20, 2000)]
        problem_object = {'fun': objective_one_compartment, 'bounds': bounds}
    elif problem == 'two_compartment':
        if bounds is None:
            bounds = [(0, 15000), (20, 2000), (20,2000)]
        problem_object = {'fun': objective_two_compartment, 'bounds': bounds}
    elif problem == 'two_compartment_v':
        if bounds is None:
            bounds = [(2000, 15000), (20, 200), (20,2000), (0, 1)]
        problem_object = {'fun': objective_two_v_compartment, 'bounds': bounds}
    elif problem == 'three_compartment':
        if bounds is None:  # T2_2, v_2
            bounds = [(0, 35), (0,1)]
        problem_object = {'fun': objective_three_compartment, 'bounds': bounds}
    elif problem == 'four_compartment':
        if bounds is None:  # T2_2, T2_3, z, r
            bounds = [(0, 35), (60,2000), (0,1), (0,1)]
        problem_object = {'fun': objective_four_compartment, 'bounds': bounds}
    else:
        print('ERROR: problem_string description doesnt match')
    return problem_object


def is_monotonic_index(data:NDArray) -> NDArray:
    """given NDArray of shape [...,i], return a bool array of indexes that are monc

    Args:
        data (NDArray): [...,i], checking if each signal is monotonic over last dir

    Returns:
        NDArray: [...] bool for each voxel is monotonic
    """
    data_diff = np.zeros(shape=(*data.shape[:-1], data.shape[-1]-1))    # one less
    for t in range(data_diff.shape[-1]):
        data0 = data[...,t]
        data1 = data[...,t+1]
```

```
        data_diff[...,t] = data1 - data0
        data_diff_increasing_bool = data_diff > 0
        is_data_mono = np.where(data_diff_increasing_bool.sum(axis=-1) == 0, True,

        return is_data_mono


def AIC(signal_actual:NDArray, signal_est:NDArray, N:int) -> NDArray:
    """Given true and estimated signal, with the number of estimated parameters in

    Args:
        signal_actual (NDArray): [...,t] True signal
        signal_est (NDArray): [...,t] estimated signal
        N (int): number of estimated parameters in the model

    Returns:
        AIC (NDArray): AIC of the model at each voxel (different fitted parameters
    """
    N = N + 1
    sum_squared_diff = SSD(signal_actual, signal_est, is_normalised=False)  # [...
    K = signal_actual.shape[-1] # t
    AIC = 2 * N + K * np.log(sum_squared_diff / K)  # [...]
    return AIC



class MRIDataLoader():
    """Load MRI data using class"""
    def __init__(self, root_dir='data\EPICure_qt2/', file_start_name='Epicure'):
        """Class to load MRI data easily

        Args:
            root_dir (str): folder relative location, end in /
            file_start_name (str, optional): eg. 'Epicure' or 'case'
        """
        self.root_dir = root_dir
        self.file_start_name=file_start_name
        self.roi_dict = {"brain":0,"CSF":1,"GM":2,"WM":3,"DeepGM":4,"brainstem":5}
        self.roi_id_dict = {0:'brain', 1:'CSF', 2:'GM', 3:'WM', 4:'DeepGM', 5:'bra
        if file_start_name=='Epicure':
            self.subject_ids = self.get_subject_ids()


    def get_subject_ids(self, subject_char_start=7, subject_id_length=5):
        """Return all the subject ids in the data directory

        Args:
            subject_char_start (int): where in the file name does the subject ID st
            subject_id_length (int): how many digits is the subject ID

        Returns:
            subject_ids (numpy array): 1D array of subject ids
        """
        subject_ids_set = set()

        # loop over the files in the directory
        for filename in os.listdir(self.root_dir):
            # check if the file is a file (not a directory)
            if os.path.isfile(os.path.join(self.root_dir, filename)):
                # extract the characters from 8 to 12 and append them to the list
                try:
                    start = subject_char_start
                    end = start + subject_id_length
                    subject_ids_set.add(int(filename[start:end]))
```

```
                except:
                    pass

        subject_ids = np.sort(np.array(list(subject_ids_set)))

        # remove the ids that error when loading, or don't have all 10x TEs
        for subject_id_del in SUBJECT_IDS_REMOVE:
            subject_id_del_index = np.squeeze(np.argwhere(subject_ids == subject_i
            subject_ids = np.delete(subject_ids, subject_id_del_index)

        return subject_ids


    def get_TE_times(self):
        """return TE times

        Returns:
            numpy array: 1D array
        """
        TE_times = np.loadtxt(self.root_dir + 'TEs.txt', delimiter=' ')
        return TE_times


    def get_roi_dicts(self):
        """Return the roi dict and the roi_id_dict
        """
        return self.roi_dict, self.roi_id_dict


    def get_img(self, subject_id, file_type, is_normalise=False):
        """Return 3D brain image of type specified

        Args:
            subject_id (int): subject number
            file_type (string): name of particular part of img, eg. 'qt2', or 'seg1
            is_normalise (boolean): returns the img normalised between [0,1] if Tru

        Return
            img (numpy array): [96, 96, 51, X] - 3D brain img, potentially 4D if se
        """
        # par1 is GM cortical labels for temporal/occipital/parietal/frontal (left
        # For par2, these are WM labels for genu/splenium of corpus callosum and pc
        file_dict = {'signal':'qt2', 'mask':'mask1', 'seg1':'qt2_seg1', 'seg':'qt2_
        try:
            file_type = file_dict[file_type]
        except:
            file_type = file_type
        if subject_id < 1000:
            subject_id = self.subject_ids[subject_id]
        file_name = self.file_start_name + str(subject_id) + '-' + file_type + '.ni
        data_nifti = nib.load(self.root_dir + file_name)
        img = data_nifti.get_fdata()
        if is_normalise:
            img = self.normalise_img(img)

        # if getting seg1 then swap brain mask for background mask
        if file_type == 'qt2_seg1':
            # make each compartment sum to one (or be less than one)
            img = np.clip(img, a_min=0, a_max=1)
            norm_inv = 1 / np.where(img.sum(axis=-1) > 1, img.sum(axis=-1), 1)
            img = np.einsum('...i,...->...i', img, norm_inv)

            # change the first segmentation to be the brain segmentation
            mask = self.get_img(subject_id=subject_id, file_type='mask', is_normali
```

```
            img[:,:,:,0] = mask
        return img


    def get_info_dict(self):
        """get the info.mat file - only available for EPICure data folder

        Returns:
            info (dict): dictionary of meta data about the subjects
        """
        file_name = self.root_dir + 'info.mat'
        info = scipy.io.loadmat(file_name)
        return info


    def get_info(self, info_id: int):
        """given info id according to info_dict['label'] return that info for all s
        1: GAB, 2: MF,

        Args:
            info_id (int): index of the info wanted from info_dict['label']

        Returns:
            subjects_info (NDArray): Info for all subject ids
        """

        info_dict = self.get_info_dict()
        n = info_dict['new'].shape[0]
        subject_ids = np.zeros(n)
        subject_infos = np.zeros(n)

        for idx in range(n):
            subject_ids[idx] = int(info_dict['new'][idx, 0])
            subject_infos[idx] = info_dict['new'][idx,info_id]


        # remove the ids that error when loading, or don't have all 10x TEs
        for subject_id_del in SUBJECT_IDS_REMOVE:
            subject_id_del_index = np.squeeze(np.argwhere(subject_ids == subject_i
            subject_infos = np.delete(subject_infos, subject_id_del_index)
            subject_ids = np.delete(subject_ids, subject_id_del_index)

        return subject_infos


    def normalise_img(self, img):
        """changes image to be between [0,1] - don't do this if fitting, need non-n
        only useful for visualising images

        Args:
            img (numpy array): any numpy array of any size

        Returns:
            normalised_img (numpy array): same size as given img array but all valu
        """
        max = img.max()
        return img / max


    def get_preterm_ids(self):
        """return all the subject ids that are preterm

        Returns:
            NDArray: preterm ids, fullterm_ids
```

```
        """
        is_preterm = self.get_info(1) < 26
        is_fullterm = ~is_preterm
        preterm_ids = np.arange(self.subject_ids.shape[0])[is_preterm]
        fullterm_ids = np.arange(self.subject_ids.shape[0])[is_fullterm]
        return preterm_ids, fullterm_ids


    def load_nb_roi_thres(self, thresh: float) -> NDArray:
        """calculates the number of voxels in each subject in each roi over thresho

        Args:
            thresh (float): threshold value

        Returns:
            NDArray: [n,roi_id]
        """
        file_path = f'data/arrays/nb_roi_subject_t{thresh}.npy'
        try:
            nb_roi_subject = np.load(file_path)
        except:
            print('No File exists - Creating Data...')
            nb_roi_subject = []
            for subject_id in tqdm(self.subject_ids, ascii=True):
                seg_data = self.get_img(subject_id, 'seg')
                seg_data_flatten = seg_data.reshape(-1, seg_data.shape[-1])   # [n
                nb_roi = (seg_data_flatten > thresh).sum(axis=0)
                nb_roi_subject.append(nb_roi)
            nb_roi_subject = np.array(nb_roi_subject)
            np.save(file_path, nb_roi_subject)
        return nb_roi_subject


    def load_all_roi_thresh_mono_data(self, thresh: float, subject_names='all'):
        """Load all mri and seg data for all subjects according to parameters

        Args:
            thresh (float): threshhold given to classify each roi
            subject_names (str): ['all', 'preterm', 'fullterm'] - what subject ids

        Returns:
            NDArray: Returns 2x NDArrays of mri data and seg data
        """
        preterm_ids, fullterm_ids = self.get_preterm_ids()

        if subject_names == 'all':
            subject_ids = self.subject_ids
            preterm_string = 'all'
        elif subject_names == 'preterm':
            subject_ids = preterm_ids
            preterm_string = 'preterm'
        elif subject_names == 'fullterm':
            subject_ids = fullterm_ids
            preterm_string = 'fullterm'
        else:
            print('ERROR: subject_names incorrect')
            return 0


        root_path = 'data/arrays/'
        file_end_path = f'_{preterm_string}_t{str(thresh)}_mono.npy'

        try:
            roi_data = np.load(root_path + 'roi_data' + file_end_path, allow_pickl
```

```
            roi_seg =  np.load(root_path + 'roi_seg'  + file_end_path, allow_pickl
        except:
            print('No File exists - Creating Data...')
            roi_data_subject = [[],[],[],[],[],[]]
            roi_seg_subject = [[],[],[],[],[],[]]

            for subject_id in tqdm(subject_ids, ascii=True):
                mri_data = self.get_img(subject_id, 'signal')
                seg_data = self.get_img(subject_id, 'seg')

                mri_data_flatten = mri_data.reshape(-1, mri_data.shape[-1])
                seg_data_flatten = seg_data.reshape(-1, seg_data.shape[-1])

                is_mono = is_monotonic_index(mri_data_flatten).astype(bool)

                mri_data_flatten_mono = mri_data_flatten[is_mono]
                seg_data_flatten_mono = seg_data_flatten[is_mono]
                is_roi = seg_data_flatten_mono > thresh

                for roi_id in self.roi_id_dict:
                    roi_data_subject[roi_id].append(mri_data_flatten_mono[is_roi[.
                    roi_seg_subject[roi_id].append(seg_data_flatten_mono[is_roi[..

            roi_data = []
            roi_seg = []
            for roi_id in self.roi_id_dict:
                roi_data.append(np.concatenate(roi_data_subject[roi_id], axis=0))
                roi_seg.append(np.concatenate(roi_seg_subject[roi_id], axis=0))
            roi_data = np.asarray(roi_data, dtype=object)
            roi_seg = np.asarray(roi_seg, dtype=object)

            np.save(root_path + 'roi_data' + file_end_path, roi_data)
            np.save(root_path + 'roi_seg' + file_end_path, roi_seg)

        return roi_data, roi_seg


    def load_all_roi_WM_GM_thresh_mono_data(self, thresh: float, subject_names='al
        """Load all mri and seg data for all subjects according to parameters

        Args:
            thresh (float): threshhold given to classify each roi
            subject_names (str): ['all', 'preterm', 'fullterm'] - what subject ids

        Returns:
            NDArray: Returns 2x NDArrays of mri data and seg data
        """
        preterm_ids, fullterm_ids = self.get_preterm_ids()

        if subject_names == 'all':
            subject_ids = self.subject_ids
            preterm_string = 'all'
        elif subject_names == 'preterm':
            subject_ids = preterm_ids
            preterm_string = 'preterm'
        elif subject_names == 'fullterm':
            subject_ids = fullterm_ids
            preterm_string = 'fullterm'
        else:
            print('ERROR: subject_names incorrect')
            return 0


        root_path = 'data/arrays/'
```

```
        file_end_path = f'_{preterm_string}_GM_WM_t{str(thresh)}_mono.npy'

        WM_id, GM_id = self.roi_dict['WM'], self.roi_dict['GM']

        try:
            data_final = np.load(root_path + 'roi_data' + file_end_path, allow_pickl
            seg_final =  np.load(root_path + 'roi_seg'  + file_end_path, allow_pickl
        except:
            print('No File exists - Creating Data...')
            data_subject = []
            seg_subject = []

            for subject_id in tqdm(subject_ids, ascii=True):
                mri_data = self.get_img(subject_id, 'signal')
                seg_data = self.get_img(subject_id, 'seg')

                mri_data_flatten = mri_data.reshape(-1, mri_data.shape[-1])
                seg_data_flatten = seg_data.reshape(-1, seg_data.shape[-1])

                is_mono = is_monotonic_index(mri_data_flatten).astype(bool)

                mri_data_flatten_mono = mri_data_flatten[is_mono]
                seg_data_flatten_mono = seg_data_flatten[is_mono]
                is_thresh = (seg_data_flatten_mono[:,WM_id] + seg_data_flatten_mono

                data_subject.append(mri_data_flatten_mono[is_thresh,:])
                seg_subject.append(seg_data_flatten_mono[is_thresh,:])

            data_final = np.concatenate(data_subject, axis=0)
            seg_final = np.concatenate(seg_subject, axis=0)

            np.save(root_path + 'roi_data' + file_end_path, data_final)
            np.save(root_path + 'roi_seg' + file_end_path, seg_final)

        return data_final, seg_final
```