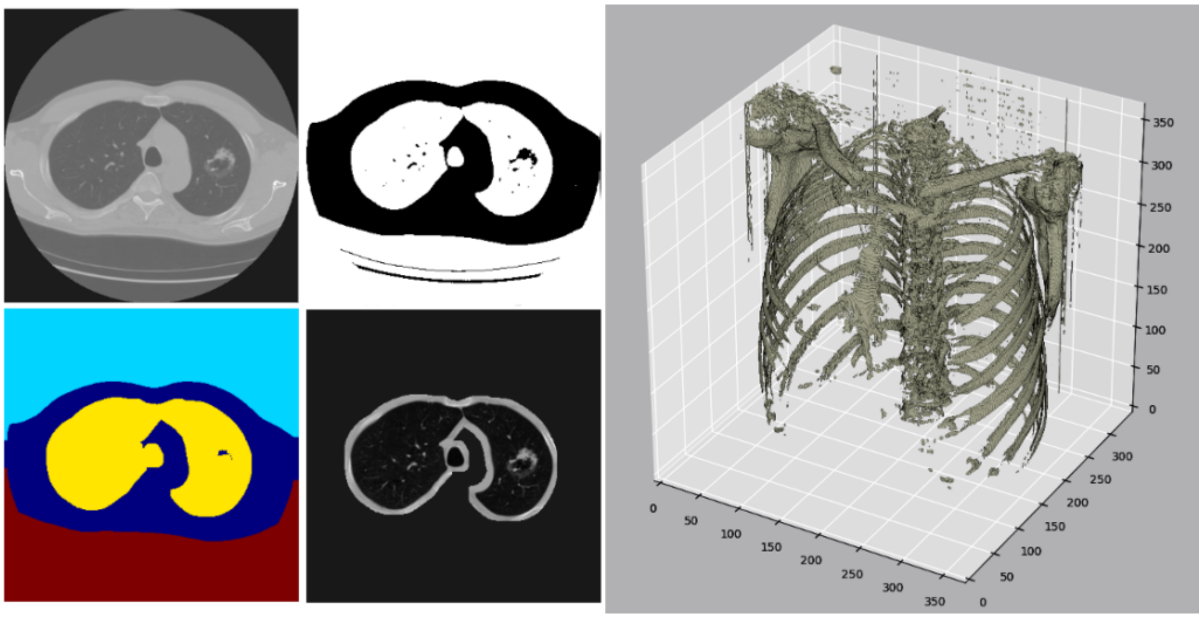
DICOM Processing and Segmentation in Python

January 28, 2017 Howard Chen 57 Comments Python, Quests



DICOM is a pain in the neck. It also happens to be very helpful. As clinical radiologists, we expect post-processing, even taking them for granted. However, the magic that occurs behind the scenes is no easy feat, so let's explore some of that magic.

In this quest, we will be starting from raw DICOM images. We will extract voxel data from DICOM into `numpy` arrays, and then perform some low-level operations to normalize and resample the data, made possible using information in the DICOM headers.

The remainder of the Quest is dedicated to visualizing the data in 1D (by histogram), 2D, and 3D. Finally, we will create segmentation masks that remove all voxel except for the lungs.

Processing raw DICOM with Python is a little like excavating a dinosaur – you'll want to have a jackhammer to dig, but also a pickaxe and even a toothbrush for the right situations. Python has all the tools, from pre-packaged imaging process packages handling gigabytes of data at once to byte-level operations on a single voxel.

Update 1/5/2019:

The Kaggle data science bowl 2017 dataset is no longer available. However, for learning and testing purposes you can use the National Lung Screening Trial chest CT dataset.

To follow along, set up your computer using the following Python tutorials:

- Setting Up the Python Data Science Environment
- Quick Tutorial for Jupyter

Alternatively, start a free Jupyter notebook from Azure Notebooks.

Getting Ready

If you're using Anaconda, you will have already have access to almost every necessary package necessary for this task. The notable exception is `dicom`, to do this, the easiest way is using `pip` from the command line:

```
pip install pydicom
```

To perform 3D plotting, we are using the free version of `plot.ly` in offline mode which uses WebGL to make visualization interactive. `plotly` and `scikit-image` can be installed using conda:

```
conda install plotly
```

```
conda install scikit-image
```

We will be using features from scikit-image 0.13 or above, which may require building from source. Instructions are here (http://scikit-image.org/docs/dev/install.html). Check your version with this command:

```
python -c "import skimage; print skimage.__version__"
```

If you're using Python v3.x, then you'd want to use the appropriate `print` syntax:

```
python -c "import skimage; print(skimage.__version__)"
```

Finally, you need a DICOM image stack. For this exercise, we are using Kaggle's Data Science Bowl 2017 dataset (https://www.kaggle.com/c/data-science-bowl-2017).

Some of the code used here are adapted from Kaggle contributors (such as Guido Zuidhorf (https://www.kaggle.com/gzuidhof/) and Booze Allen Hamilton's data team (https://www.kaggle.com/c/data-science-bowl-2017/details/tutorial)) who generously share their work. I have made modifications to clarify what happens at each step using more visuals and additional or simplified code.

Import Packages

```
%reload_ext signature
%matplotlib inline

import numpy as np
import dicom
import os
import matplotlib.pyplot as plt
from glob import glob
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
import scipy.ndimage
from skimage import morphology
from skimage import measure
from skimage.transform import resize
from sklearn.cluster import KMeans
from plotly import __version__
from plotly.offline import download_plotlyjs, init_notebook_mode, plo
t, iplot
from plotly.tools import FigureFactory as FF
from plotly.graph_objs import *
init_notebook_mode(connected=True)
```

Then, let's specify a specific DICOM study we can take a closer look. Let's take a look at a chest CT stack from Kaggle which contains a lung cancer.

The whole dataset is 140GB unzipped, but each examination is only 70MB or so.

Here we'll use the patient ID `5267ea7baf6332f29163064aecf6e443` from that dataset, which has been labeled as positive for lung cancer.

```
data_path = "/data/LungCancer-data/stage1/train/cancer/5267ea7baf6332
f29163064aecf6e443/"
output_path = working_path = "/home/howard/Documents/"
g = glob(data_path + '/*.dcm')

# Print out the first 5 file names to verify we're in the right folde
r.
print ("Total of %d DICOM images.\nFirst 5 filenames:" % len(g))
print '\n'.join(g[:5])
```

```
Total of 145 DICOM images.
First 5 filenames:
/data/LungCancer-data/stage1/train/cancer/5267ea7baf6332
f29163064aecf6e443/be386f61171cdae7f7ecbfe60dbac897.dcm
/data/LungCancer-data/stage1/train/cancer/5267ea7baf6332
f29163064aecf6e443/81a1e10bf9b8f45edc444ae8fe2601cc.dcm
/data/LungCancer-data/stage1/train/cancer/5267ea7baf6332
f29163064aecf6e443/c8a92b47e098b5372f247580518eecdc.dcm
/data/LungCancer-data/stage1/train/cancer/5267ea7baf6332
f29163064aecf6e443/4e1e82a9e728a78e08602b8af9b0ef94.dcm
/data/LungCancer-data/stage1/train/cancer/5267ea7baf6332
f29163064aecf6e443/d1a54381364ccb3626737a23f0bb7c00.dcm
```

Helper Functions

Here we make two helper functions.

- `load_scan` will load all DICOM images from a folder into a list for manipulation.
- The voxel values in the images are raw. `get_pixels_hu` converts raw values into Houndsfeld units (https://en.wikipedia.org/wiki/Hounsfield_scale)
  - The transformation is linear. Therefore, so long as you have a slope and an intercept, you can rescale a voxel value to HU.
  - Both the rescale intercept and rescale slope are stored in the DICOM header at the time of image acquisition (these values are scanner-dependent, so you will need external information).

```
#
# Loop over the image files and store everything into a list.
#

def load_scan(path):
    slices = [dicom.read_file(path + '/' + s) for s in os.listdir(pat
h)]
    slices.sort(key = lambda x: int(x.InstanceNumber))
    try:
        slice_thickness = np.abs(slices[0].ImagePositionPatient[2] -
slices[1].ImagePositionPatient[2])
```

```
slices[1].ImagePositionPatient[2])
    except:
        slice_thickness = np.abs(slices[0].SliceLocation - slices[1].
SliceLocation)

    for s in slices:
        s.SliceThickness = slice_thickness

    return slices

def get_pixels_hu(scans):
    image = np.stack([s.pixel_array for s in scans])
    # Convert to int16 (from sometimes int16),
    # should be possible as values should always be low enough (<32k)
    image = image.astype(np.int16)

    # Set outside-of-scan pixels to 1
    # The intercept is usually -1024, so air is approximately 0
    image[image == -2000] = 0

    # Convert to Hounsfield units (HU)
    intercept = scans[0].RescaleIntercept
    slope = scans[0].RescaleSlope

    if slope != 1:
        image = slope * image.astype(np.float64)
        image = image.astype(np.int16)

    image += np.int16(intercept)

    return np.array(image, dtype=np.int16)

id=0
patient = load_scan(data_path)
imgs = get_pixels_hu(patient)
```

This is a good time to save the new data set to disk so we don't have to reprocess the stack every time.

```
np.save(output_path + "fullimages_%d.npy" % (id), imgs)
```

Displaying Images

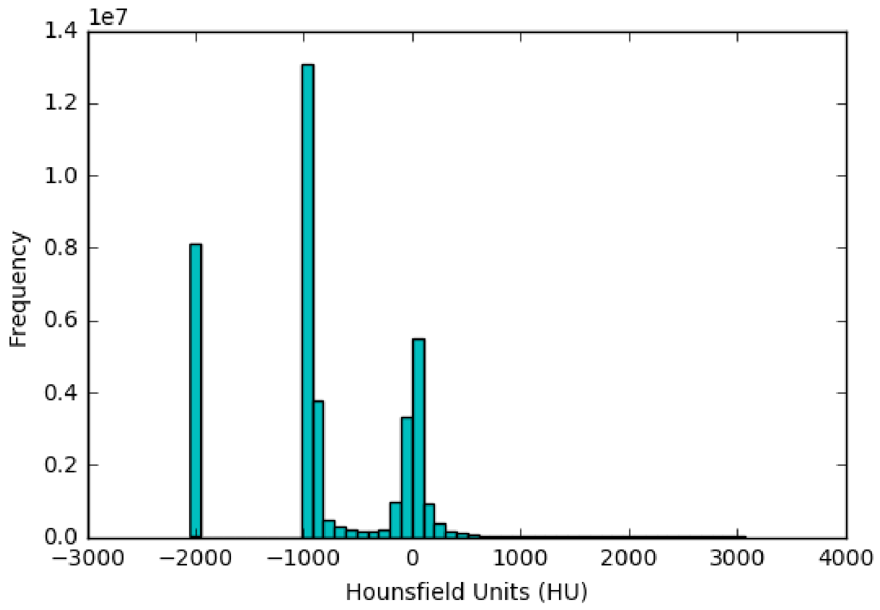
The first thing we should do is to check to see whether the Houndsfeld Units are properly scaled and represented.

HU's are useful because it is standardized across all CT scans regardless of the absolute number of photons the scanner detector captured. If you need a refresher, here's a quick list of a few useful ones, sourced from Wikipedia.

Substance	HU
Air	−1000
Lung	−500
Fat	−100 to −50
Water	0
Blood	+30 to +70
Muscle	+10 to +40
Liver	+40 to +60
Bone	+700 (cancellous bone) to +3000 (cortical bone)

Let's now create a histogram of all the voxel data in the study.

```
file_used=output_path+"fullimages_%d.npy" % id
imgs_to_process = np.load(file_used).astype(np.float64)

plt.hist(imgs_to_process.flatten(), bins=50, color='c')
plt.xlabel("Hounsfield Units (HU)")
plt.ylabel("Frequency")
plt.show()
```



Critiquing the Histogram

The histogram suggests the following:

- There is lots of air
- There is some lung
- There's an abundance of soft tissue, mostly muscle, liver, etc, but there's also some fat.
- There is only a small bit of bone (seen as a tiny sliver of height between 700-

3000)

This observation means that we will need to do significant preprocessing if we want to process lesions in the lung tissue because only a tiny bit of the voxels represent lung.

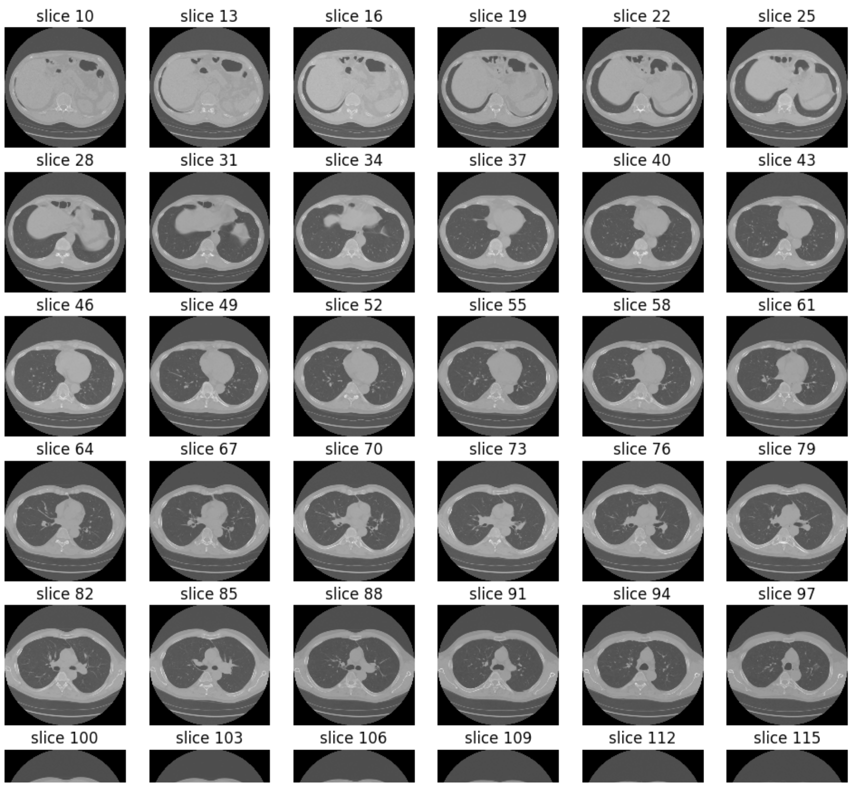
More interestingly, what's the deal with that bar at -2000? Air really only goes to -1000, so there must be some sort of artifact.

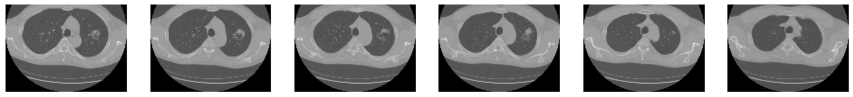
Let's take a look at the actual images.

Displaying an Image Stack

We don't have a lot of screen real estate, so we'll be skipping every 3 slices to get a representative look at the study.

```
id = 0
imgs_to_process = np.load(output_path+'fullimages_{}.npy'.format(id))

def sample_stack(stack, rows=6, cols=6, start_with=10, show_every=3):
    fig,ax = plt.subplots(rows,cols,figsize=[12,12])
    for i in range(rows*cols):
        ind = start_with + i*show_every
        ax[int(i/rows),int(i % rows)].set_title('slice %d' % ind)
        ax[int(i/rows),int(i % rows)].imshow(stack[ind],cmap='gray')
        ax[int(i/rows),int(i % rows)].axis('off')
    plt.show()

sample_stack(imgs_to_process)
```





So as it turns out, what we were seeing as HU=-2000 are the voxels outside of the bore of the CT. "Air," in comparison, appears gray because it has a much higher value. As a result, the lungs and soft tissue have somewhat reduced contrast resolution as well.

We will try to manage this problem when we normalize the data and create segmentation masks.

(By the way, did you see the cancer? It's on slices 97-112.)

Resampling

Although we have each individual slices, it is not immediately clear how thick each slice is.

Fortunately, this is in the DICOM header.

```
print "Slice Thickness: %f" % patient[0].SliceThickness
print "Pixel Spacing (row, col): (%f, %f) " % (patient[0].PixelSpacin
g[0], patient[0].PixelSpacing[1])
```

```
Slice Thickness: 2.500000
Pixel Spacing (row, col): (0.722656, 0.722656)
```

This means we have 2.5 mm slices, and each voxel represents 0.7 mm.

Because a CT slice is typically reconstructed at 512 x 512 voxels, each slice represents approximately 370 mm of data in length and width.

Using the metadata from the DICOM we can figure out the size of each voxel as the slice thickness. In order to display the CT in 3D isometric form (which we will do below), and also to compare between different scans, it would be useful to ensure that each slice is resampled in 1x1x1 mm pixels and slices.

```
id = 0
imgs_to_process = np.load(output_path+'fullimages_{}.npy'.format(id))
def resample(image, scan, new_spacing=[1,1,1]):
    # Determine current pixel spacing
    spacing = map(float, ([scan[0].SliceThickness] + scan[0].PixelSpa
cing))
    spacing = np.array(list(spacing))
    resize_factor = spacing / new_spacing
    new_real_shape = image.shape * resize_factor
    new_shape = np.round(new_real_shape)
    real_resize_factor = new_shape / image.shape
    new_spacing = spacing / real_resize_factor

    image = scipy.ndimage.interpolation.zoom(image, real_resize_facto
r)

    return image, new_spacing

print "Shape before resampling\t", imgs_to_process.shape
imgs_after_resamp, spacing = resample(imgs_to_process, patient, [1,1,
1])
print "Shape after resampling\t", imgs_after_resamp.shape
```

```
Shape before resampling (145, 512, 512)
Shape after resampling  (362, 370, 370)
```

3D Plotting

Having isotropic data is helpful because it gives us a sense of the Z-dimension. This means we now have enough information to plot the DICOM image in 3D space. For kicks we'll focus on rendering just the bones.

Visualization Toolkit (VTK) (http://vtk.org) is excellent for 3D visualization because it can utilize GPU for fast rendering. However, I can't get VTK to work in Jupyter, so we will take a slightly different approach:

- Create a high-quality static using 3D capability of `matplotlib`
- Create a lower-quality but interactive render using `plotly`, which has WebGL support via JavaScript.

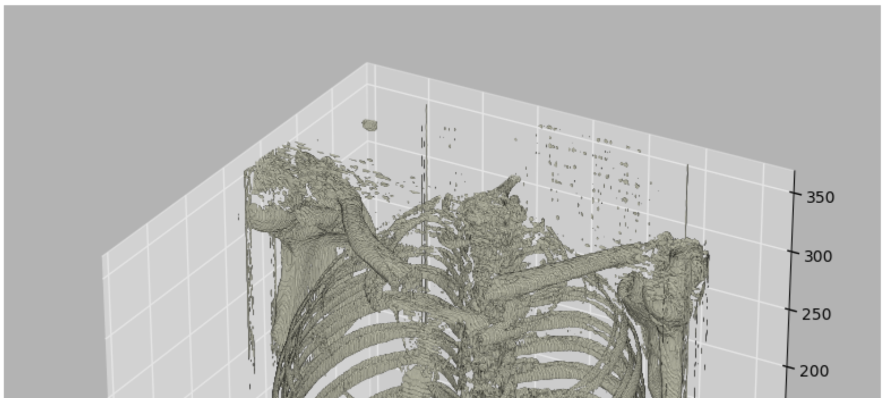
The marching cubes (https://en.wikipedia.org/wiki/Marching_cubes) algorithm is used to generate a 3D mesh from the dataset. The `plotly` model will utilize a higher `step_size` with lower voxel threshold to avoid overwhelming the web browser.

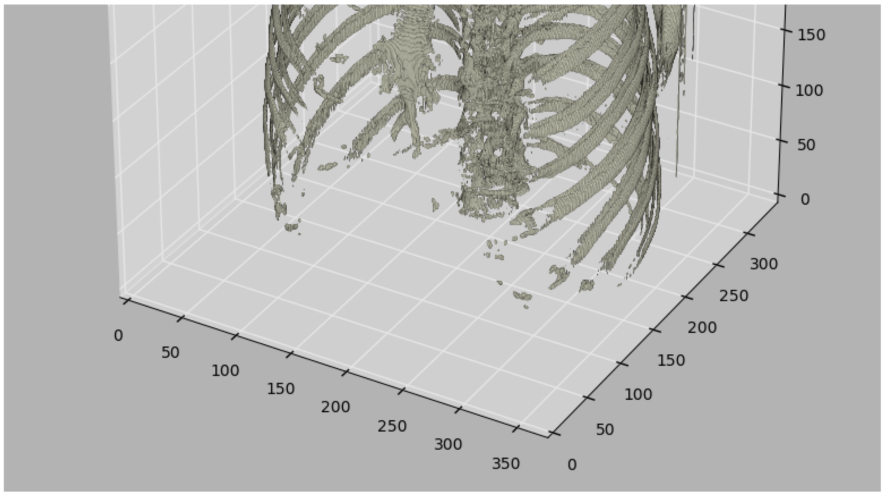
```
def make_mesh(image, threshold=-300, step_size=1):

    print "Transposing surface"
    p = image.transpose(2,1,0)

    print "Calculating surface"
    verts, faces, norm, val = measure.marching_cubes(p, threshold, st
ep_size=step_size, allow_degenerate=True)
    return verts, faces
```

```
def plotly_3d(verts, faces):
    x,y,z = zip(*verts)

    print "Drawing"

    # Make the colormap single color since the axes are positional no
t intensity.
#    colormap=['rgb(255,105,180)','rgb(255,255,51)','rgb(0,191,255)']
    colormap=['rgb(236, 236, 212)','rgb(236, 236, 212)']

    fig = FF.create_trisurf(x=x,
                        y=y,
                        z=z,
                        plot_edges=False,
                        colormap=colormap,
                        simplices=faces,
                        backgroundcolor='rgb(64, 64, 64)',
                        title="Interactive Visualization")
    iplot(fig)

def plt_3d(verts, faces):
    print "Drawing"
    x,y,z = zip(*verts)
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111, projection='3d')

    # Fancy indexing: `verts[faces]` to generate a collection of tria
ngles
    mesh = Poly3DCollection(verts[faces], linewidths=0.05, alpha=1)
    face_color = [1, 1, 0.9]
    mesh.set_facecolor(face_color)
    ax.add_collection3d(mesh)

    ax.set_xlim(0, max(x))
    ax.set_ylim(0, max(y))
    ax.set_zlim(0, max(z))
    ax.set_axis_bgcolor((0.7, 0.7, 0.7))
    plt.show()
```

```
v, f = make_mesh(imgs_after_resamp, 350)
plt_3d(v, f)
```

```
Transposing surface
Calculating surface
Drawing
```

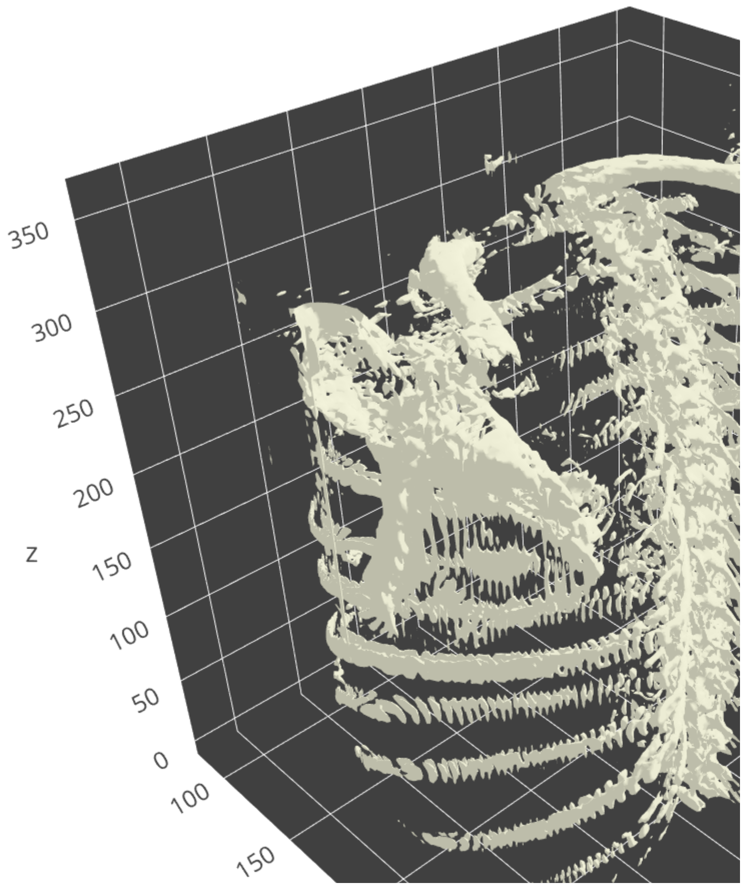


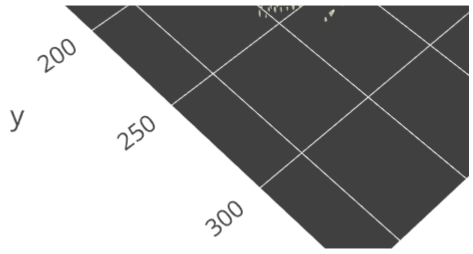


```
v, f = make_mesh(imgs_after_resamp, 350, 2)
plotly_3d(v, f)
```

```
Transposing surface
Calculating surface
Drawing
```

Interactive Visualiz





Segmentation

If you are interested in chest CTs because you're interested in picking up lung cancers, you're not alone.

Machine learning algorithms work a lot better when you can narrowly define what it is looking at. One way to do this is by creating different models for different parts of a chest CT. For instance, a convolutional network for lungs would perform better than a general-purpose network for the whole chest.

Therefore, it is often useful to pre-process the image data by auto-detecting the boundaries surrounding a volume of interest.

The below code will:

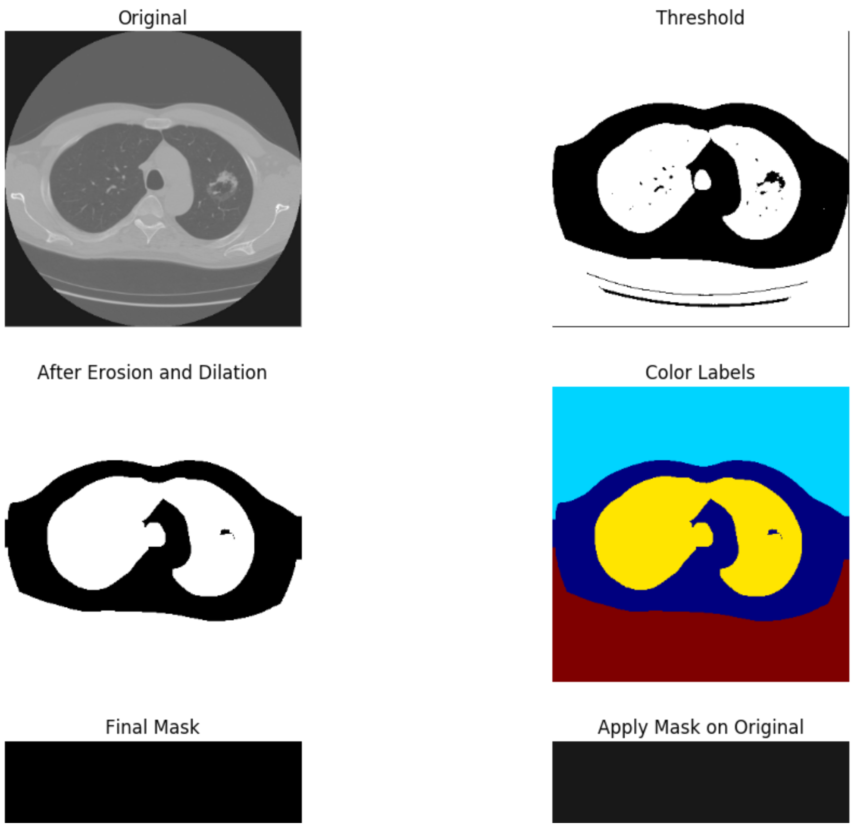
- Standardize the pixel value by subtracting the mean and dividing by the standard deviation
- Identify the proper threshold by creating 2 KMeans clusters comparing centered on soft tissue/bone vs lung/air.
- Using Erosion (https://en.wikipedia.org/wiki/Erosion_(morphology)) and Dilation (https://en.wikipedia.org/wiki/Dilation_(morphology)) which has the net effect of removing tiny features like pulmonary vessels or noise
- Identify each distinct region as separate image labels (think the magic wand in Photoshop)
- Using bounding boxes for each image label to identify which ones represent lung and which ones represent "every thing else"
- Create the masks for lung fields.
- Apply mask onto the original image to erase voxels outside of the lung fields.

```
#Standardize the pixel values
def make_lungmask(img, display=False):
    row_size= img.shape[0]
    col_size = img.shape[1]
    mean = np.mean(img)
    std = np.std(img)
    img = img-mean
    img = img/std
    # Find the average pixel value near the lungs
    # to renormalize washed out images
    middle = img[int(col_size/5):int(col_size/5*4),int(row_size/5):in
t(row_size/5*4)]
    mean = np.mean(middle)
    max = np.max(img)
    min = np.min(img)
    # To improve threshold finding, I'm moving the
    # underflow and overflow on the pixel spectrum
    img[img==max]=mean
    img[img==min]=mean
    #
    # Using Kmeans to separate foreground (soft tissue / bone) and ba
ckground (lung/air)
    #
    kmeans = KMeans(n_clusters=2).fit(np.reshape(middle,[np.prod(midd
le.shape),1]))
    centers = sorted(kmeans.cluster_centers_.flatten())
    threshold = np.mean(centers)
    thresh_img = np.where(img<threshold,1.0,0.0)  # threshold the ima
ge

    # First erode away the finer elements, then dilate to include som
e of the pixels surrounding the lung.
    # We don't want to accidentally clip the lung.

    eroded = morphology.erosion(thresh_img,np.ones([3,3]))
    dilation = morphology.dilation(eroded,np.ones([8,8]))

    labels = measure.label(dilation) # Different labels are displayed
 in different colors
    label_vals = np.unique(labels)
    regions = measure.regionprops(labels)
    good_labels = []
    for prop in regions:
        B = prop.bbox
        if B[2]-B[0]<row_size/10*9 and B[3]-B[1]<col_size/10*9 and B[
0]>row_size/5 and B[2]<col_size/5*4:
            good_labels.append(prop.label)
    mask = np.ndarray([row_size,col_size],dtype=np.int8)
    mask[:] = 0

    #
    #  After just the lungs are left, we do another large dilation
    #  in order to fill in and out the lung mask
    #
    for N in good_labels:
        mask = mask + np.where(labels==N,1,0)
    mask = morphology.dilation(mask,np.ones([10,10])) # one last dila
tion

    if (display):
        fig, ax = plt.subplots(3, 2, figsize=[12, 12])
        ax[0, 0].set_title("Original")
        ax[0, 0].imshow(img, cmap='gray')
        ax[0, 0].axis('off')
        ax[0, 1].set_title("Threshold")
        ax[0, 1].imshow(thresh_img, cmap='gray')
        ax[0, 1].axis('off')
        ax[1, 0].set_title("After Erosion and Dilation")
        ax[1, 0].imshow(dilation, cmap='gray')
        ax[1, 0].axis('off')
        ax[1, 1].set_title("Color Labels")
        ax[1, 1].imshow(labels)
        ax[1, 1].axis('off')
        ax[2, 0].set_title("Final Mask")
        ax[2, 0].imshow(mask, cmap='gray')
        ax[2, 0].axis('off')
        ax[2, 1].set_title("Apply Mask on Original")
        ax[2, 1].imshow(mask*img, cmap='gray')
        ax[2, 1].axis('off')

        plt.show()
    return mask*img
```

Single Slice Example At Each Step

We want to make sure the algorithm doesn't accidentally exclude cancer from the region of interest (due to its "soft tissue" nature). So let's test this out on a single slice.

```
img = imgs_after_resamp[260]
make_lungmask(img, display=True)
```



```
Out[13]:  array([[-0., -0., -0., ..., -0., -0.,  0.],
                 [-0., -0., -0., ..., -0., -0.,  0.],
                 [-0., -0., -0., ..., -0., -0.,  0.],
                 ...,
                 [-0., -0., -0., ..., -0., -0.,  0.],
                 [-0., -0., -0., ..., -0., -0.,  0.],
                 [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

A Few Observations

Compare the difference in contrast between the finished slice alongside the original. Not only is extrapulmonary data properly cleaned up, the contrast is also improved.

If we were to apply a machine learning algorithm to the image stack, the algorithm would have a much easier time to identify a primary lung lesion. The Kaggle lung cancer data contains labeled cancer and no-cancer datasets that can be used for this training (and a $1MM bounty).

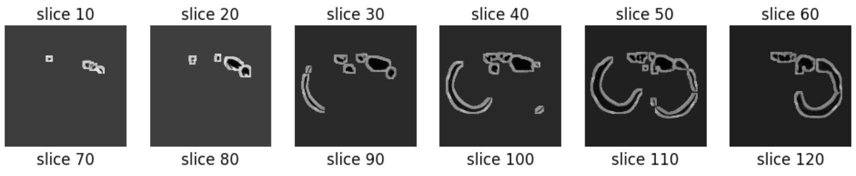
Downsides of using this mask appropach is you can miss hilar/perihilar disease fairly easily.

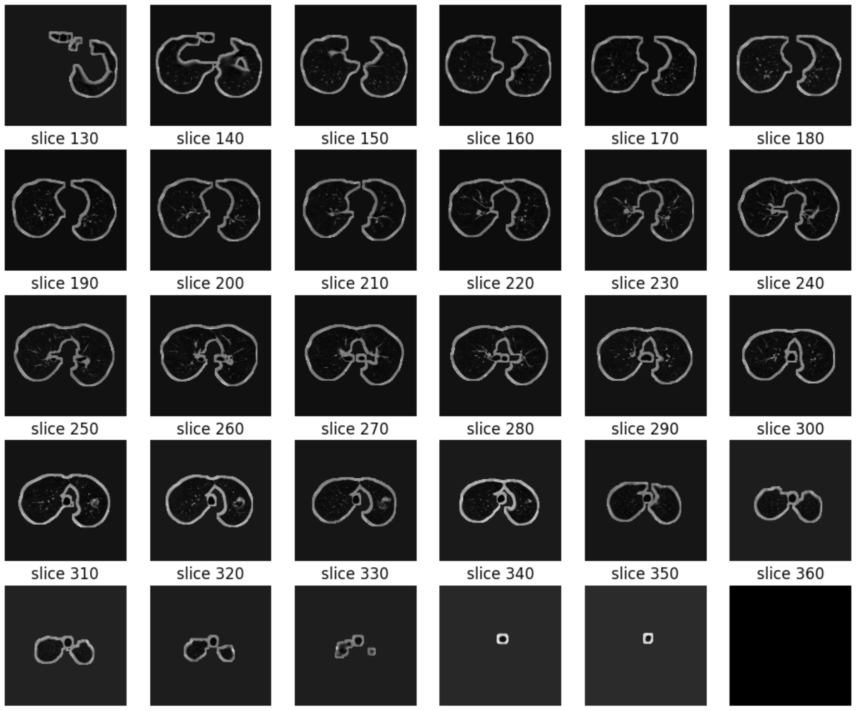
Apply Masks to All Slices

The single-slice example seemed to work pretty well.

Let's now apply the mask to all the slices in this CT and show a few examples.

```
masked_lung = []

for img in imgs_after_resamp:
    masked_lung.append(make_lungmask(img))

sample_stack(masked_lung, show_every=10)
```





Looks like things check out.

The lung lesion is properly preserved in the ROI, and it appears to work wel from lung bases all the way to the apices.

This would be a good time to save the processed data.

```
np.save(output_path + "maskedimages_%d.npy" % (id), imgs)
```

Conclusion

DICOM data can take a lot of getting used to, but Python provides a lot of useful tools to make things easier.

In this exercise, we have accomplished the following:

- Loaded DICOM data using `pydicom`
- Used 1D (histogram), 2D, and 3D plots to display DICOM images.
- Pre-processed data for future machine learning projects
  - Conversion of pixel value to Hundsfeld units
  - Resampling for isotropy
  - Segmentation
  - Masking

Where To Go From Here

There are many directions, such as these:

- Practice on DICOM data. There are freely available data sets, or you can export your own anonymized image set. For instance, try 3D plotting the bones in a MSK trauma case.
- Try your hands on the LUNA 2016 grand challenge (https://luna16.grand-challenge.org/) for pulmonary nodule analysis.The file format here are .mhd, but the segmentation/preprocessing concepts are the same.
- Try your hands on the Kaggle 2017 Data Science Bowl, which provides labeled cancer and normal chest CTs in DICOM format.

```
%signature
```

Out[44]: Author: Howard Chen (http://howardpchen.me/) Last edited: January 29, 2017

Linux 4.4.0-59-generic - CPython 2.7.13 - IPython 5.1.0 - matplotlib 1.5.3 - numpy 1.11.1 - pandas 0.18.1 - scikit-image 0.13dev

Share this:

23

Related

Automated IVC Filter Detection
April 26, 2017
In "Python"

Azure Notebooks in Preview Status - And It's Pretty Awesome
October 7, 2016
In "Commentary"

Its not Python OR R, its Python AND R
August 22, 2016
In "Commentary"

Howard Chen

Associate Informatics Officer at Cleveland Clinic Imaging Institute

(Howard) Po-Hao Chen, MD MBA is the Associate Informatics Officer at the Cleveland Clinic Imaging Institute and a musculoskeletal radiology subspecialist. He has an interest in data-driven radiology, quality improvement, and innovation. Howard has an MD and MBA from Harvard University, and he finished training with fellowships in musculoskeletal radiology, nuclear medicine, and clinical imaging informatics in June 2018 from University of Pennsylvania.

DICOM Image processing Segmentation Visualization