pipeline-description

August 15, 2014

1 Getting to know the image-photo-z pipeline

---

image-photo-z is a pixel-level method for estimating photometric redshifts. The process from beginning to end involves a number of transformations and procedures on the images that are documented here. Wherever and whenever possible, the logic, implementation, validation and expressions for the stages in the pipeline are provided. This file does **NOT** describe the API. See the API reference for more details regarding the API.

1.1 Getting the object catalog ready

For training the machine learning algorithm to be used, we need the pixel data and known redshifts from known sources. For this, a catalog of objects for which redshifts are known is necessary. This step gets this catalog from SDSS, which is our primary source of data. We have followed a two-step procedure to achieve this. First, we query the SDSS server for all 'valid' objects. The meaning of 'valid' will be clear from the following SQL code we use (provided as image-photo-z/catalog_gen/sql/QueryDR10.sql, run on CasJobs):

```
In []: SELECT TOP 10
         spec.specObjID AS id,
         spec.ra,
         spec.dec,
         spec.z AS redshift,
         spec.zErr AS redshiftError,
         specAll.class,
         specAll.run,
         specAll.camcol,
         specAll.field,
         specAll.dered_u AS u,
         specAll.dered_g AS g,
         specAll.dered_r AS r,
         specAll.dered_i AS i,
         specAll.dered_z AS z
       INTO objects_with_redshifts
       FROM SpecObj AS spec
       JOIN SpecPhotoAll AS specAll
       ON spec.specObjID = specAll.specObjID,
         PhotoObj AS phot
       WHERE
         specAll.ObjID = phot.ObjID
         AND phot.CLEAN = 1
         AND spec.zWarning = 0
```

We have selected from the objects with known redshifts some parameters (which are evident from their names) of these objects. We have taken data from only those objects which have clean photometry and no warnings in redshift measurement.

Once we have data for all objects matching our criterion, we query this small database for a region of the sky we want to use as follows (provided as image-photo-z/catalog_gen/sql/GetArea.sql):

```
In []: SELECT ALL
         objs.id,
         objs.ra,
         objs.dec,
         objs.redshift,
         objs.redshiftError,
         objs.class,
         objs.run,
         objs.camcol,
         objs.field
       INTO mydb.one_square_degree
       FROM mydb.objects_with_redshifts AS objs
       WHERE
         objs.ra>180 AND objs.ra<181
         AND objs.dec>45 AND objs.dec<46
```

Insert in the above code appropriate values of ra and dec to get your desired region of the sky.

We save the output of the above query as a CSV file and use this file as the input catalog for the pipeline.

1.2 Downloading images

The catalog files created above contain information on the run-camcol-field of the frame in which the above object was captured. This is the identity of the frame. We first enumerate all the distinct run-camcol-field combinations in a logfile so as to prevent multiple downloads of the same file. Now the URL of an image frame in the SDSS catalog is of the form http://data.sdss3.org/sas/dr10/boss/photoObj/frames/[rerun]/[run]/[camcol]/frame-[band]-[run]-[camcol]-[field].fits.bz2. We run this over the logfile, get images into the images folder and extract them. We name the images as [run]-[camcol]-[field]-[band].fits (3813-5-23-r.fits, for instance) for easy future reference. Note that if we choose to use local images, they need to be named similarly as well. The relevant code block is presented here for completeness.

```
In []: pref="http://data.sdss3.org/sas/dr10/boss/photoObj/frames/"
       downloadURL=pref+rerun+"/"+run+"/"+camcol+"/frame-"+band+"-"+run.zfill(6)+"-"+\
       camcol+"-"+field.zfill(4)+".fits.bz2"
       if not os.path.isfile(downloadFolder+"/"+run+"-"+camcol+"-"+field+"-"+band+".fits"):
               if band==bands[0]:
                       logfileFile.write(run+"-"+camcol+"-"+field+"\n")
               urllib.urlretrieve(downloadURL, downloadFolder+"/"+run+"-"+camcol+"-"+field+\
                   "-"+band+".fits.bz2")
               os.system("bunzip2 "+downloadFolder+"/"+run+"-"+camcol+"-"+field+"-"+band+\
                   ".fits.bz2")
```

As an additional feature, if an image is already present it is not downloaded again.

1.3 Getting image errors

Once we have the images, we have complete information about all pixels compressed in a FITS file. The primary HDU in the file is the actual corrected image of the sky (which corresponds to the pixel values we want). We do not need to put any effort at all to get the corrected frame values from the FITS file. However, some machine learning algorithms also require the errors in these pixel values. The ‘error’ values are encoded in a ‘sky image matrix’ as described in http://data.sdss3.org/datamodel/files/BOSS_PHOTOOBJ/frames/RERUN/RUN/CAMCOL/frame.html.

This ‘sky image’ is of size lesser than the actual image (256x192 vs 2048x1489) and we need to interpolate this image to “blow” it up at its right size. For this interpolation we use the scipy interp2d function. The relevant code block for getting the full error matrix is shown here:

```
In []: skyImageInit=fitsFile[2].data[0][0]
       # Since we're interpolating an image whose values are known at a grid, we use
       # interp2d. It requires an array of the grid x and grid y positions, which
       # is what we're generating in first two lines. Then we fit the interpolator.
       xs=numpy.fromfunction(lambda k: k, (skyImageInit.shape[0],), dtype=int)
       ys=numpy.fromfunction(lambda k: k, (skyImageInit.shape[1],), dtype=int)
       interpolator=interp2d(xs, ys, skyImageInit, kind='cubic')
       for j in range(errorImage.shape[1]):
               for k in range(errorImage.shape[0]):
                       skyImageValue=interpolator.__call__(j*skyImageInit.shape[0]\
                       /errorImage.shape[0],k*skyImageInit.shape[1]/errorImage.shape[1])
                       errorImage[k][j]=return_sdss_pixelError(band, camcol, run,\
                       fitsImage[k][j], skyImageValue, errorImg[k][j])
```

Here, we have fitted a function to the smaller 256x192 array using a cubic interpolation and to ‘expand’ it, have scaled the axes by (1489/256) and (2048/192) respectively.

The errors are calculated as shown in the link above. The relevant functions are:

```
In []: def return_sdss_gain(band, camcol, run):
               lut=[{},{'u':1.62,'g':3.32,'r':4.71,'i':5.165,'z':4.745},\
                {'u':1.595,'g':3.855,'r':4.6,'i':6.565,'z':5.155},\
                {'u':1.59,'g':3.845,'r':4.72,'i':4.86,'z':4.885},\
                {'u':1.6,'g':3.995,'r':4.76,'i':4.885,'z':4.775},\
                {'u':1.47,'g':4.05,'r':4.725,'i':4.64,'z':3.48},\
                {'u':2.17,'g':4.035,'r':4.895,'i':4.76,'z':4.69}]
               if run>1100:
                       lut[2]['u']=1.825
               return lut[camcol][band]


       def return_sdss_darkVariance(band, camcol, run):
               lut=[{},{'u':9.61,'g':15.6025,'r':1.8225,'i':7.84,'z':0.81},\
                {'u':12.6025,'g':1.44,'r':1.0,'i':5.76,'z':1.0},\
                {'u':8.7025,'g':1.3225,'r':1.3225,'i':4.6225,'z':1.0},\
                {'u':12.6025,'g':1.96,'r':1.3225,'i':6.25,'z':9.61},\
                {'u':9.3025,'g':1.1025,'r':0.81,'i':7.84,'z':1.8225},\
                {'u':7.0225,'g':1.8225,'r':0.9025,'i':5.0625,'z':1.21}]
               if run>1500:
                       lut[2]['i']=6.25
                       lut[4]['i']=7.5625
                       lut[4]['z']=12.6025
                       lut[5]['z']=2.1025
               return lut[camcol][band]


       def return_sdss_pixelError(band, camcol, run, img, simg, cimg):
               dn=img/cimg+simg
               dn_err=math.sqrt(abs((dn/return_sdss_gain(band, camcol, run))+\
                   return_sdss_darkVariance(band, camcol, run)))
               img_err=dn_err*cimg
               return img_err
```

This was validated in the following way:

1) Checked registration between original error and original image.

2) Checked registration between registered error and registered image.

These images match well in registration. Now the error predicted by the SDSS model increases with increasing image intensities. Since bright regions seem to match between images, astrometry seems to be fine between these images.

3) Used ds9 to check error magnitudes with respect to signal value. The error seemed to be around the signal value in the background and significantly less than the signal at object centers. At object boundaries though, sometimes errors become greater than signal values (as expected).

1.4 Image Registration

Now we have images and image errors in all filters. We need to align images in different filters to make sure corresponding pixels line up. This process is Image Registration. We implement this using the Montage Image Mosaic Engine and montage_wrapper for Python. The relevant code lines are:

```
In []: montage_wrapper.commands.mGetHdr(refImage,headerName)
       montage_wrapper.reproject(inputImages,outputImages,header=headerName,\
                   north_aligned=True,system='EQUJ',exact_size=True,common=True,...)
       montage_wrapper.reproject(intErrorImages,outputErrorImages,header=headerName,\
                   north_aligned=True,system='EQUJ',exact_size=True,common=True,...)
```

The reproject function reprojects the given image list to a given FITS header. If some images have the same header, they naturally have the same astrometry and hence are aligned. So we reproject all images to the same (reference image) header and get the aligned output images. The 'exact_size' argument must be set to true for images to align exactly.

We validated this by using the 'blink' feature in ds9 FITS viewer. One can visually observe that the images are well-aligned after following this procedure by flipping through them using this feature on ds9.

1.5 Generating pixel data

We are now ready to generate pixel data. The final output from this stage needs to be vectors of the form ([list of magnitudes in all bands], [redshift, redshiftError], [other optional parameters]) corresponding to each pixel. Redshifts are assigned to pixels from the catalogs by recognizing which object the pixel belongs to. This is implemented using ASSOC in SExtractor. SExtractor is run on images in all bands for a frame with the standard (for now) set of parameters.

The ASSOC requires that we supply to it a list of expected pixel coordinates for objects. It returns to us a list of parameters we specify in the config file. The conversion from absolute object coordinates to pixel coordinates for the given frame, given the image header is done using astropy.wcs as follows:

```
In []: # Get the FITS hdulist
       hdulist = fits.open(filename)
       # Create the coordinate system object using the header
       w = wcs.WCS(hdulist[0].header)
       # Calculate expected pixel coordinates
       px, py = w.wcs_world2pix([ra], [dec], 1)
```

The part of the config file related to the ASSOC is (the significance of this can be found in the SExtractor manual):

```
In []: ASSOC_NAME       sky.list       # name of the ASCII file to ASSOCiate, the expected pixel
                                       # coordinates list given as [id, xpos, ypos]
       ASSOC_DATA       1              # columms of the data to replicate (0=all), replicate id
                                       # of the object in the SExtractor output file
       ASSOC_PARAMS     2,3            # columns of xpos,ypos[,mag] in the expected pixel
                                       # coordinates list
       ASSOC_RADIUS     5.0            # cross-matching radius (pixels)
       ASSOC_TYPE       NEAREST        # ASSOCiation method: FIRST, NEAREST, MEAN,
                                       # MAG_MEAN, SUM, MAG_SUM, MIN or MAX
       ASSOCSELEC_TYPE  MATCHED        # ASSOC selection type: ALL, MATCHED or -MATCHED
```

We also ask SExtractor for the peak (in intensity), maximum and minimum pixel coordinates from each identified object and a segmentation image (which is non-zero only wherever an object is found, and is a particular "object mask" value wherever an object is found. This object mask is unique to a given object).

Now given all this, for a given object, we note the object mask for its peak pixel. All pixels having this mask belong to this object. We scan the array from minimum to maximum coordinates and add the pixel magnitudes if the object masks match. We also remove those pixels whose intensities are nan in the way. The relevant part of the code is:

```
In []: # Iterate over all pixels in the rectangle
       for j in range([minObjectX], [maxObjectX]):
               for k in range([minObjectY], [maxObjectY]):
                       isPixelValid=1
                       for l in fitsImages:
                           # Make pixel invalid if any of the values is a nan
                               if math.isnan(float(l[k][j])):
                                       isPixelValid=0
                               if isPixelValid==1:
                               # Check if the pixel belongs to our object
                               # The object flag has been noted earlier in the code
                                       if segImage[k][j]==thisObjFlag:
                                               trainingVector=[]
                                               for l in fitsImages:
                                                       trainingVector.append(float(l[k][j]))
                                               trainingVector.append(redshift)
                                               trainingVector.append(redshiftError)
                                               trainingVector.append(objClass)
                                               pixelRA, pixelDec = w.wcs_pix2world(float(j),\
                                                       float(k), 1)
                                               trainingVector.append(pixelRA)
                                               trainingVector.append(pixelDec)
                                               distance=math.sqrt(pow((k-thisObjX),2)+\
                                                       pow((j-thisObjY),2))/maxDist
                                               trainingVector.append(distance)
                                               for l in fitsErrors:
                                                       trainingVector.append(float(l[k][j]))
                               # Build the training array for this object
                                               trainingArray.append(trainingVector)
```

1.6 Training and testing

Now that we have pixel data, we train and test some machine learning algorithms. Currently, two algorithms are implemented in the code: kNN and MLZ. kNN is implemented with Scikit-learn. First we get the classified data ready into the file format that these software packages require the data to be in for training. Refer to the documentation for MLZ and Scikit-learn kNNClassifier and kNNRegressor for more details.

1.7 Visualising output

If using MLZ, consult the MLZ documentation for a description of how to generate plots and so on.

If using kNN, the kNN output file specified in the config can be plotted directly on GNUPlot to obtain a graph of predicted versus actual redshift or class.

1.8 Further improvements

This is a work under progress, and further improvements and fine-tuning must be made. At this stage, this is a rough to-do list:

1) Check objects that give isolated pixels. In the segmentation maps generated by SExtractor, some segments seem to give pixels that are isolated from the object itself. Check why this is happening and see if changing SExtractor input parameters improves this.

2) Some pixels seem to give negative flux values. This also happens in the images themselves, so this is not explicitly a problem with the code. Check if this is valid for the training.

3) Check why the image registration part takes a lot of time and remove the bottleneck. The "blowing up" of the sky noise image from the FITS HDU takes a lot of time. Check this.

4) Remove duplicate pixels between images. Some images in the data-set overlap and this is necessary.

5) Port the code to run on a GPU.

6) Automate generation of the catalog using sdssCL.

7) Add timing support.

8) Implement other machine learning algorithms.

9) Check the validity of the interpolation used to "blow up" the sky image. Use other interpolations if more valid.

10) Check how pixel redshifts are combined to form object redshifts. For now, we're doing direct averaging.

11) Create a setup file or an installer to install the package in some standard location like dist-utils.

12) Add more stuff to this file.