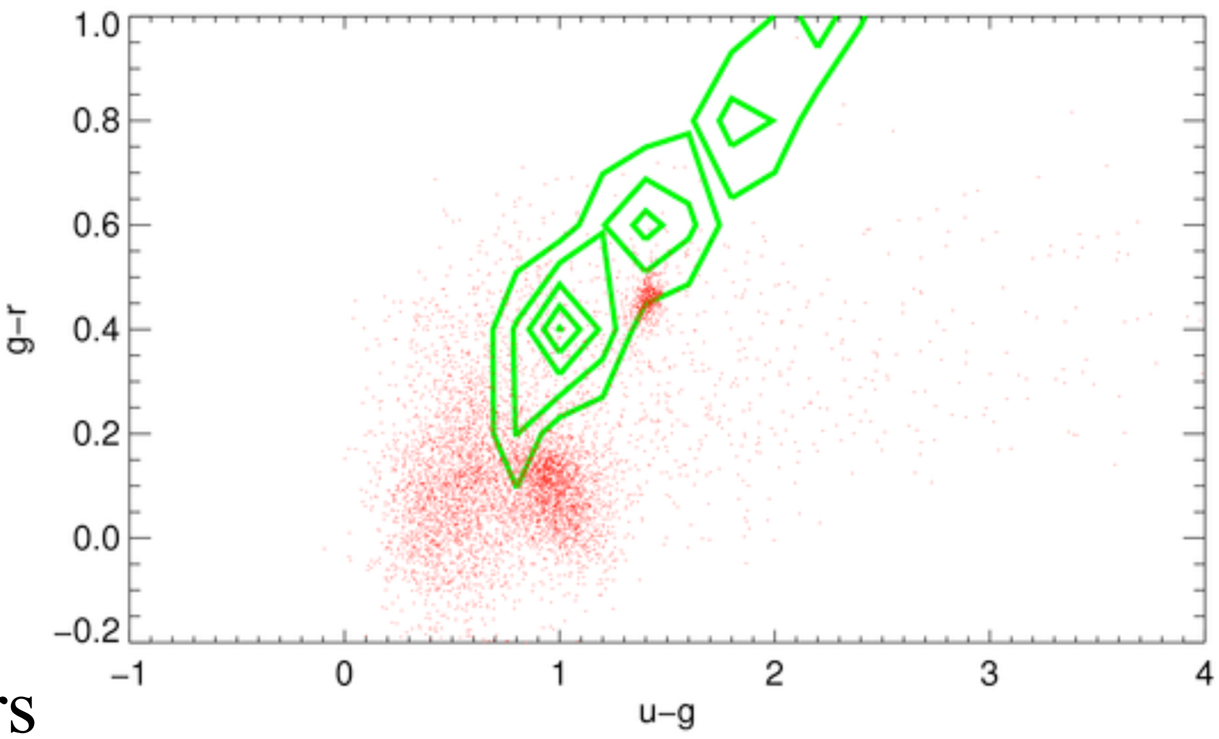
Deconvolution

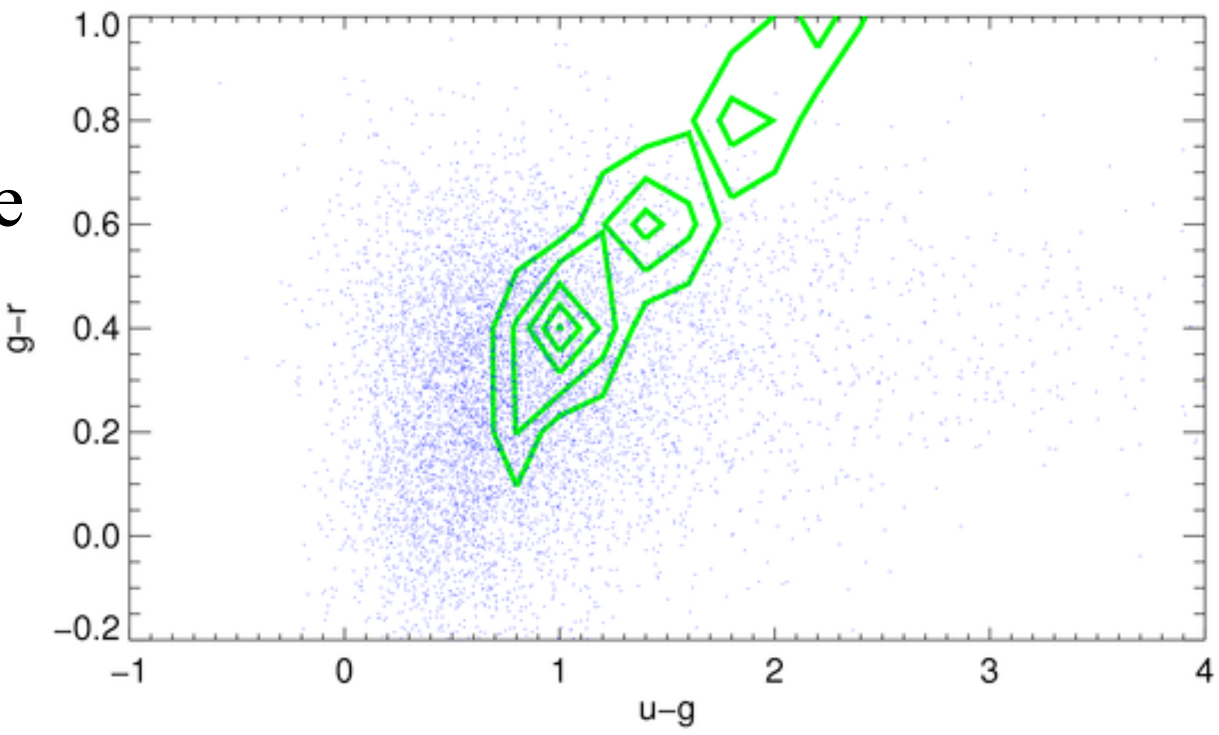
Errors and Photometric Classification of Objects



- It becomes more difficult to classify objects as imaging data degrades
- In particular, surveys only achieve a certain depth before errors on fluxes become large. Near the limits of an imaging survey errors on colors therefore increase *significantly*
- Consider the figure, which plots the colors of a set of quasars in the redshift range $2.5 < z < 3.5$ in red and the positions of typical stars in color space in green...

Errors and Photometric Classification of Objects

- ...this figure is similar *but* the quasar colors have been scattered according to their errors near the imaging limits of a survey



- It is now much harder to distinguish the quasars and stars using colors (particularly as the colors of the stars will also degrade)
- In the prior figure, some quasars were in distinct parts of color space, which would allow photometric redshifts to be derived...but now this information is washed out

Deconvolution to the Rescue

- To solve this sort of problem, the correct approach is to model the noisy distribution of objects in flux space by *correctly incorporating the measurement errors*
  - Errors on PSFFLUX in the sweeps files are stored as inverse variances and named PSFFLUX_IVAR
  - flux_error = 1./np.sqrt(PSFFLUX_IVAR)
  - mag_error = (2.5/np.log(10)) * flux_error/flux
- A common model of any measurement and its errors is a mixture (of possibly N-dimensional) Gaussians (with means, μ and standard deviations σ)
  - by *deconvolving* measurement space into (μ,σ) the *true* error-free distribution of the measurements can be estimated from the set of means μ

Deconvolution to the Rescue

- Deconvolution is described in a great deal more detail in Bovy et al. (2009, 2011, 2012; linked from the syllabus)
- Our Python build, includes a version of astroML (linked from the syllabus) that includes code for breaking a measurement space into a mixture of Gaussians
- Look at the *AstroML Extreme Deconvolution Example* linked from the syllabus. The basic setup is:
  - Stack measurements that you would plot as (*x*, *y*)
    - *np.vstack([x, y]).T*
  - Construct the covariance matrix for your data sample using the errors on the measurements
    - *Xerr[:, diag, diag] = np.vstack([dx**2, dy**2]).T*

The covariance matrix

- If variance (for *n* samples of normally distributed data) is
  - $\sigma_i^2 = 1/(n-1)\ \Sigma_i\ (x_i - \mu_i)^2$
- Then the *covariance* matrix is
  - $C_{ij} = 1/(n-1)\ \Sigma_{i,j}\ (x_i - \mu_i)\ (x_j - \mu_j)$
  - where i, j are corresponding samples in different bins
- Which looks something like the following matrix (for *n*=5):

$$\begin{matrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} & \sigma_{14} & \sigma_{15} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} & \sigma_{24} & \sigma_{25} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 & \sigma_{34} & \sigma_{35} \\ \sigma_{41} & \sigma_{42} & \sigma_{43} & \sigma_4^2 & \sigma_{45} \\ \sigma_{51} & \sigma_{52} & \sigma_{53} & \sigma_{54} & \sigma_5^2 \end{matrix}$$

Python tasks

1. Follow this process (which is very similar to that from the *Flagging Bad Data* class in Week 10) to obtain a set of PRIMARY, i < 20 objects that are confirmed stars:
   - retrieve every point source ("*objtype=star*") in the SDSS sweeps files imaging that lie within a 3^o radius of the coordinate $(\alpha,\delta) = (180^o,30^o)$. We'll call these *objs*
   - For a circular area, you can use *sdss_sweep_circle* to retrieve the *objs*…remember to send *all=False* to only retrieve only objects that are *SURVEY_PRIMARY*
   - restrict the *objs* in magnitude to $i < 20$
   - coordinate-match (at ~1") the *objs* to the *stars-ra180-dec30-rad3.fits* file in my week 10 SVN directory, to find which $i < 20$ objects in SDSS imaging are stars

Python tasks

2. Plot *u-g* (y-axis) vs. *g-r* (x-axis) *magnitudes* for your stars to check they resemble a reasonably tight stellar locus. Note that we'll call *u-g* and *g-r* "*y*" and "*x*" on the next slide, and we'll call their errors "*dy*" and "*dx*"
   - Remember to de-extinct the magnitudes, first
   - Include the magnitude *errors* in your plot
   - For the error on, e.g., *u-g* (what we'll call "*dy*"), use:
     - *np.sqrt(u_mag_error**2 + g_mag_error**2)*
   - You can use *matplotlib* to include error bars on plots with something like the following syntax
     - *plt.figure()*
     - *plt.errorbar(gmr, umg, gmrerr, umgerr, fmt='x')*

Python tasks

3. Use AstroML's *Extreme Deconvolution* (*XD*) routine to model the distribution of your data (see the *syllabus link*)
   - Import the astroML packages
     - *from astroML.density_estimation import XDGMM*
     - *from astroML.plotting.tools import draw_ellipse*
   - Set up the measurement and covariance arrays
     - *X = np.vstack([x, y]).T*
     - *Xerr = np.zeros(X.shape + X.shape[-1:])*
     - *diag = np.arange(X.shape[-1])*
     - *Xerr[:, diag, diag] = np.vstack([dx**2, dy**2]).T*
   - Execute the *XD Gaussian Mixture Model*
     - *clf=XDGMM(5,10) #ADM 5 components, 10 iterations*
     - *clf.fit(X, Xerr) #ADM this will take about a minute*

Python tasks

4. Plot the *XD* model ellipses (which are 2-D slices through 3-D Gaussians) together with your original data points
   - *plt.figure()*
   - *plt.plot(x,y,'b,')*
   - *for i in range(clf.n_components):*
     draw_ellipse(clf.mu[i],clf.V[i],fc='gray',alpha=0.2)
   - *plt.show()*
   - You requested a 5-component fit on the previous slide, so there are 5 ellipses
   - The ellipses are *density contours* (where there are more data, the ellipses are more densely concentrated)
   - To fit more ellipses, or to improve the fits, you would increase the values in *clf = XDGMM(5, 10)*

Python tasks

5. Modeling your parameter space with N-dimensional mixtures of Gaussians is superior to putting hard cuts on data (such as the color cuts we discussed last week). Data have errors, so hard cuts are almost never justified
   - This is because data are *noisy*…they have errors expressed as standard deviations (σ) or variances (σ^2)
   - Note that XD is returning a true or error-deconvolved model of the distribution of the data. *In other words, XD is modeling what your data distribution would look like in the absence of noise!*
   - In reality, data is *probabilistic*, so comparing data to a model can be thought of as asking the question of what fraction of a density contour that represents a data point (μ,σ) overlaps a density contour that represents a model

Python tasks

6. *XD* can calculate the likelihood that data lie in the model density contours, replacing the need for hard cuts. Let's set up some test data *(x,y)* = *(0.4,1)* and *(1,1.5)* to do this. We'll adopt errors on these test data of *dx, dy=(0.01,0.01)*
   - *Xtest = np.vstack([[0.4,1.], [1.,1.5]]).T*
   - *Xtesterr = np.zeros(Xtest.shape + Xtest.shape[-1:])*
   - *diag = np.arange(Xtest.shape[-1])*
   - *Xtesterr[:,diag,diag]=np.vstack([0.01**2, 0.01**2]).T*
   - *L = np.sum(np.exp(clf.logprob_a(Xtest,Xtesterr)),axis=1)*
   - The *L* are *relative* likelihoods. Higher values mean the data better overlap the model, but note that the *L are not* normalized to be probabilities in the range [0,1]
   - Plot the test data and the model ellipses. Do the *L* make sense given the locations of the test data and the model?