Errors and Bootstrap Resampling

Statistics and Data Science

Spring 2025

Goals for today: you should be able to...

- Describe jackknife errors
- **lecture 13/14 notebook**:
  - Obtain bootstrap samples in Python
- Propagate errors in one quantity to errors in another quantity

Review: Bootstraps in pandas

- Suppose we have N data, $x_1 \ldots x_N$.
  - A bootstrap resampling of that data consists of generating another set of data, $y_1 \ldots y_M$, randomly choosing one of the original N data points for each of the y_i , with the possibility of a given x value occurring either 0 or multiple times in the y (when we resample with replacement)
- We used the `numpy.random.choice()` function, which generates bootstrap samples from an array or dataframe:

```
# this works in pandas:
hboot=np.random.choice(hdata,(nbootstraps,ndata) )
```

The Jackknife

- If the data are not all independent of each other the assumptions underlying bootstraps don't work
- An older resampling technique, known as the jackknife, is commonly used in that case.
  - Suppose the data can be broken up into N blocks, each of which is (mostly) independent of each other (e.g., corresponding to different regions of the sky).
  - In that case, we can leave out 1 block at a time, and calculate the mean (say) of all the remaining N-1 blocks of data. We can then estimate the standard deviation of the mean as sqrt((N-1)/N) times the standard deviation of the mean measurements from those N subsets.

An Introduction to Machine Learning

Introduction to scikit-learn

- The `scikit-learn` package provides implementations of many modern algorithms developed for "machine learning": developing methods of predicting quantities (classifications, values, etc.) from a set of numbers, with the algorithm optimized based upon a set of training data
- Most other machine learning packages have very similar APIs to scikit-learn
- 2 basic classes of algorithms:
  - 1) ***supervised learning***: dependent upon labels/values provided by the user (as in regression, where we are trying to match a set of y values at given x values)
  - 2) ***unsupervised learning***: algorithms that require no labels / have no dependent variable; e.g., search for groupings in data values in N-dimensional space without any input from user on what those groups should be, or find remappings of data onto simpler coordinate systems (e.g. PCA)

Introduction to scikit-learn

- Key advantage: `Scikit-learn` makes many powerful algorithms straightforward to apply using a standardized interface
- Challenges: the documentation can be obscure and provides little guidance on what the right algorithm for a particular problem may be, or what values of free parameters to use
  - May need to explore multiple algorithms, but **`scikit-learn`** does make it easy to swap one algorithm for another

Table from Ivezic et al., *Statistics, data mining, and machine learning in astronomy*

TABLE 8.1.
A summary of the practical properties of different regression methods.

Method	Accuracy	Interpretability	Simplicity	Speed
Linear regression	L	H	H	H
Linear basis function regression	M	M	M	M
Ridge regression	L	H	M	H
LASSO regression	L	H	M	L
PCA regression	M	M	M	M
Nadaraya–Watson regression	M/H	M	H	L/M
Local linear/polynomial regression	H	M	M	L/M
Nonlinear regression	M	H	L	L/M

Introduction to scikit-learn

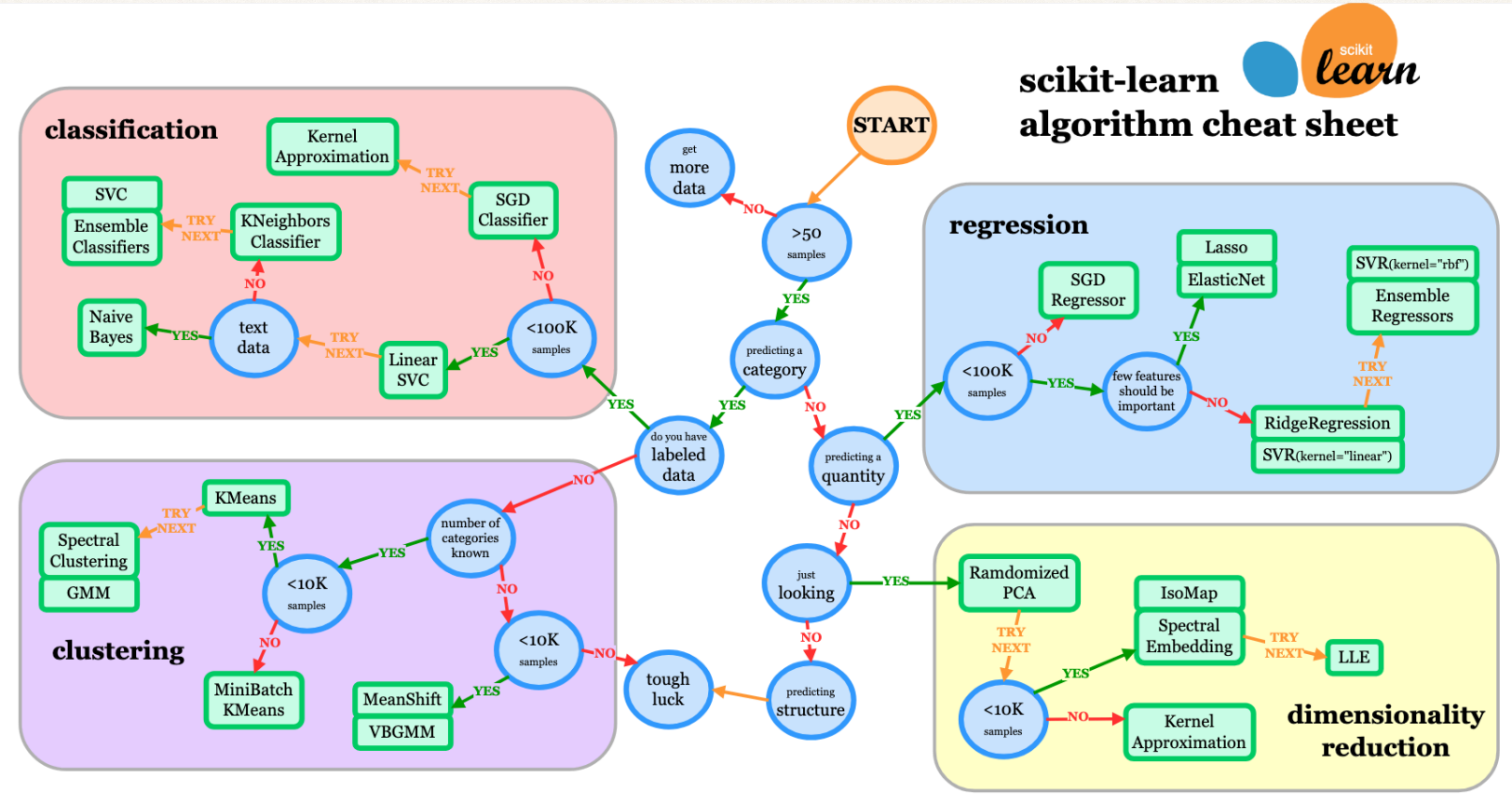
- Key advantage: `Scikit-learn` makes many powerful algorithms straightforward to apply using a standardized interface
- Challenges: the documentation can be obscure and provides little guidance on what the right algorithm for a particular problem may be, or what values of free parameters to use
  - May need to explore multiple algorithms, but **`scikit-learn`** does make it easy to swap one algorithm for another

Table from Ivezic et al., *Statistics, data mining, and machine learning in astronomy*

TABLE 9.1.
Summary of the practical properties of different classifiers.

Method	Accuracy	Interpretability	Simplicity	Speed
Naive Bayes classifier	L	H	H	H
Mixture Bayes classifier	M	H	H	M
Kernel discriminant analysis	H	H	H	M
Neural networks	H	L	L	M
Logistic regression	L	M	H	M
Support vector machines: linear	L	M	M	M
Support vector machines: kernelized	H	L	L	L
K-nearest-neighbor	H	H	H	M
Decision trees	M	H	H	M
Random forests	H	M	M	M
Boosting	H	L	L	L

Flowchart of scikit-learn methods: https://scikit-learn.org/stable/machine_learning_map.html



Two different families of algorithms tend to be most useful for physics/astrophysics problems

- Deep learning
  - Massive, many-layered neural networks with millions of free parameters.
  - Tend to dominate Kaggle competitions for unstructured data (images, text, sound).
  - Deep learning is preferred when you have massive amounts of accurate and representative training data, large-scale computing available, no need to worry about overfitting, and no concerns about interpretability.
  - Most of these methods rely on either the `Tensorflow` or `pytorch` package
- Boosted trees algorithms: e.g., Random Forest, Gradient Boosted Machines, Boosted Decision Trees
  - Most successful at winning Kaggle competitions for structured data problems; identification of key quantities in data is important
  - Provide good information about what is going on "under the hood", can be trained quickly and effectively from modest-size training sets, and good control against overfitting (extrapolate from training sets better).
  - Example algorithms: `xgboost`, `catboost`, `LightGBM`
- Plus a variety of other techniques: e.g., Mixture Models, Gaussian Processes, Bayesian methods... check out (e.g.) the CMU ML department seminar list

Deep learning methods are preferred when you have large amounts of accurate training data, large amounts of computing available, no need to worry about extrapolation/overfitting, and you don't care about interpretability. This is rare in astro, more common in particle physics with massive Monte Carlo simulations

Algo	Type	Tolerance number features	Parametrization	Memory size	Minimal required quantity	Comm	Overfitting Tendency	Difficulty	Time for Learning	Time for predicting
Linear Regression	R	Weak	Weak	Small	Small	++	Low	Weak	Weak	Weak
Logistic Regression	C	Weak	Simple	Small	Small	++	Low	Weak	Weak	Weak
Decision Tree	R & C	Strong	Simple / intuitive	Large	Small	+++	Very high	Weak	Weak	Weak
Random Forest	R & C	Strong	Simple / intuitive	Very Large	Large	++	Average	Average	Costly	Costly
Boosting	R & C	Strong	Simple / intuitive	Very Large	Large	+	Average	Average	Costly	Weak
Naive Bayes	C	Weak	No params.	Small	Small	++	Low	Weak	Weak	Weak
SVM	C	Very strong	Not intuitive	Small	Large	--	Average	High	Costly	Weak
Neural Network (NN)	C	Very strong **	Not intuitive	Inter	Large	---	Average	Very high	Costly	Weak
Deep Neural Network	C	Very strong **	Not intuitive	Very Large	Very Large	---	High	Very high	Very costly	Weak
K-Means	CL*	Strong	Simple / Intuitive		Small	+		High	Weak	
One class SVM	A	Very strong	Not intuitive	Weak	Large	--	Average	High	Costly	Weak

from https://cai.tools.sap/blog/machine-learning-algorithms/

Some resources that may be helpful in projects

Statistics, data mining, and machine learning in astronomy by Ivezic, Connolly, VanderPlas, and Gray (also available as e-book from the library)

Hands-On Machine Learning with Scikit-Learn and TensorFlow by Aurélien Géron (notebooks are at https://github.com/ageron/handson-ml3), available electronically from the library

The Python Data Science Handbook, by Jake VanderPlas: https://github.com/jakevdp/PythonDataScienceHandbook (also available as e-book from the library); see also introductory *Whirlwind Tour of Python* at https://github.com/jakevdp/WhirlwindTourOfPython

Bayesian Data Analysis by Andrew Gelman, available at *http://www.stat.columbia.edu/~gelman/book/BDA3.pdf*

Scikit-learn tutorials from Jake VanderPlas: https://github.com/jakevdp/sklearn_tutorial

Frequentism and Bayesianism: A Python-driven Primer by Jake Vanderplas (includes tutorials on MCMC), *https://arxiv.org/abs/1411.5018*

The problem of regression

- Suppose we have a set of measurements y_i that should be expressible as a function of some other set of variables, $\boldsymbol{x}_i$, having parameters α,β,γ etc. (so the ith y will depend on the *ith* vector of variables $\boldsymbol{x}_i$), plus some random scatter:

$$y_i = f(\boldsymbol{x}_i; \alpha,\beta,\gamma...) + \varepsilon_i$$

  where ε_i is a random variable of mean 0, independent of all the parameters (and possibly independent of $\boldsymbol{x}_i$, in which case the errors are called ***homoskedastic***).

- In machine learning parlance, determining a mapping from values of $\boldsymbol{x}$ to a prediction for y is considered 'regression', even if we aren't (for instance) applying least-squares methods
  - Many of the class projects will focus on problems of this nature

Working with scikit-learn

- Some jargon:
  - ***training*** data is the data used to develop an instance of a machine-learning algorithm: e.g., the x/y data we fit in a least-squares regression
  - that algorithm can then provide ***predictions*** for other data
  - the ***features*** of the training data are the quantities that will be used to make predictions (e.g., the x vector in regression as we described it before)
- We'll find that many algorithms can be run with a similar interface.

Classification and Regression Algorithms

- `Scikit-learn` provides a wide variety of algorithms that can be used either for classification (separating data into two) or for regression (predicting a floating point value).
- Two examples that tend to be effective for a wide variety of problems are ***k-Nearest-Neighbors*** and ***Random Forest.***

k-Nearest Neighbors

- Fairly simple algorithm:
  1) find the objects in the training data set closest in the N-dimensional feature space to the object you want predictions for

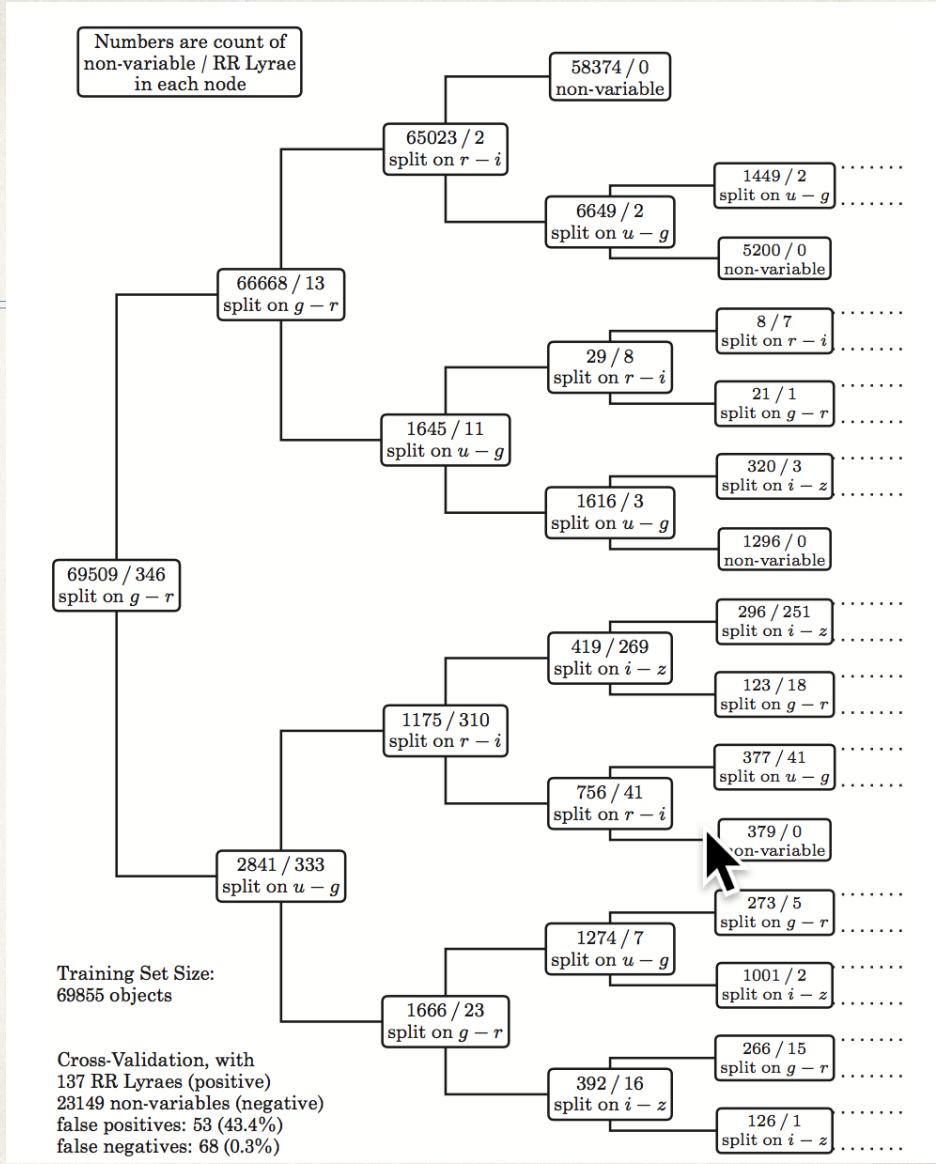
  2) Average the classifications / dependent variable values for the nearest *k* objects. That is your prediction.

  - Can optimize *k* with cross-validation methods, which we will talk about later
  - Can optionally weight neighbors inversely with distance (for many possible measures of distance)
  - Generally desirable to rescale datasets so 'distance' means something comparable for every feature, or use a distance measure that takes this into account.

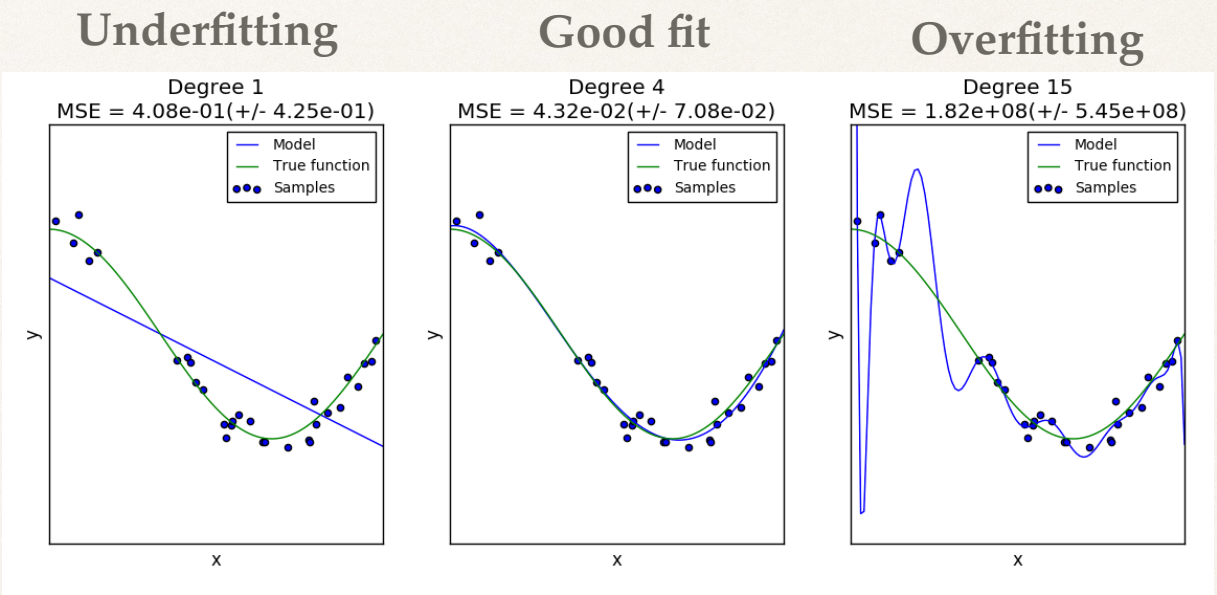
Random Forest

- Start with Decision Trees: a set of nodes where we divide up the dataset based on some feature, then split based on other features, and other features, till on the terminal 'leaves' we separate out the desired classes.
- Decision trees can have problems with overfitting

Figure from Ivezic et al., *Statistics, data mining, and machine learning in astronomy*



The problem of over-fitting



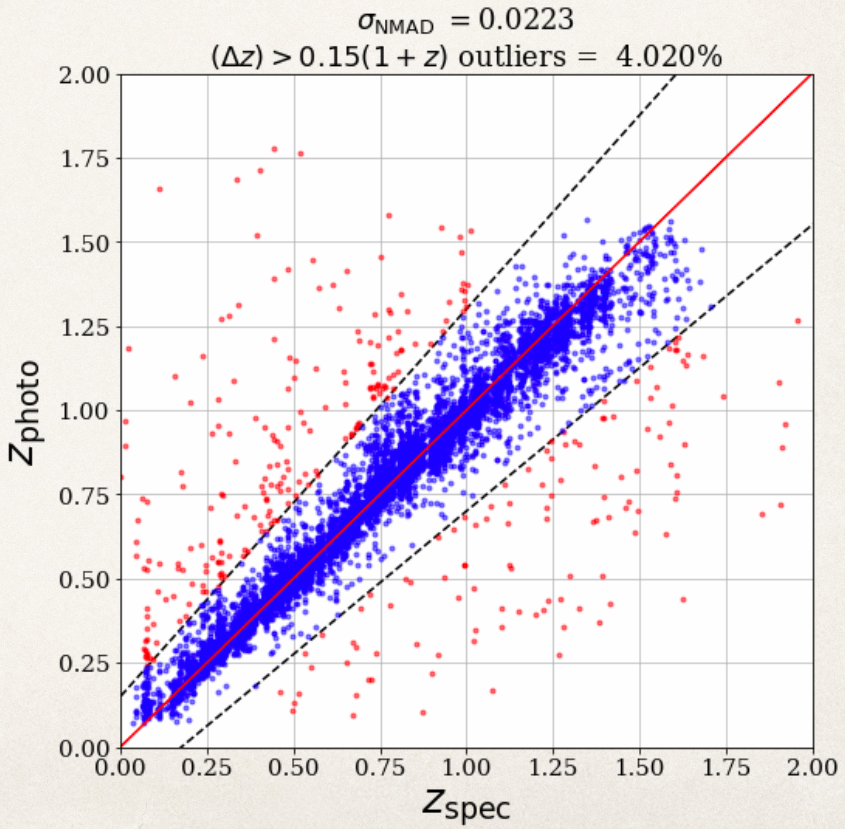
- Plot from http://scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting.html
- In the case of overfitting, the algorithm is fitting to fluctuations in a particular dataset rather than reflecting the underlying structure of the problem
-

Random Forest

- In Random Forest, many decision trees get trained from a training set of data, randomly bootstrapping amongst that data for optimizing each tree
- A random subset of possible data features are used in each node of the tree
- In the end, results from all trees get combined to give a prediction (in our case, this will be a prediction for redshift)
- For regression, each terminal leaf yields a fixed value of the y variable we are regressing for; each tree can yield only a discrete value of y for given input data, but once we average many trees the results can become pretty continuous.
- There are a few parameters to the algorithm that can be tuned; see, e.g., https://stackoverflow.com/questions/36107820/how-to-tune-parameters-in-random-forest-using-scikit-learn

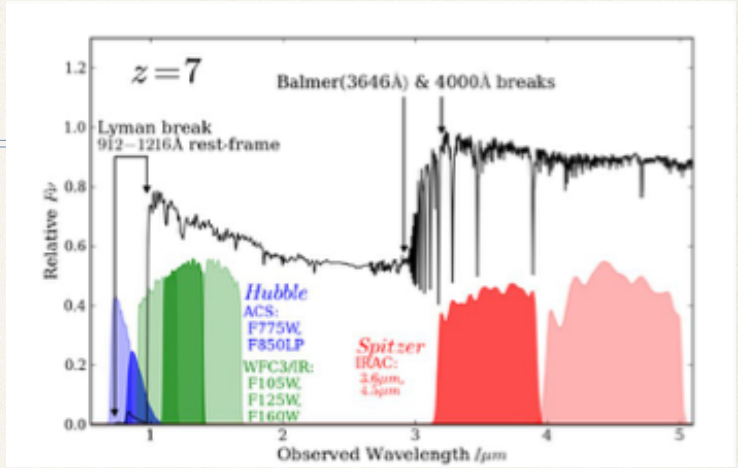
Applying these methods

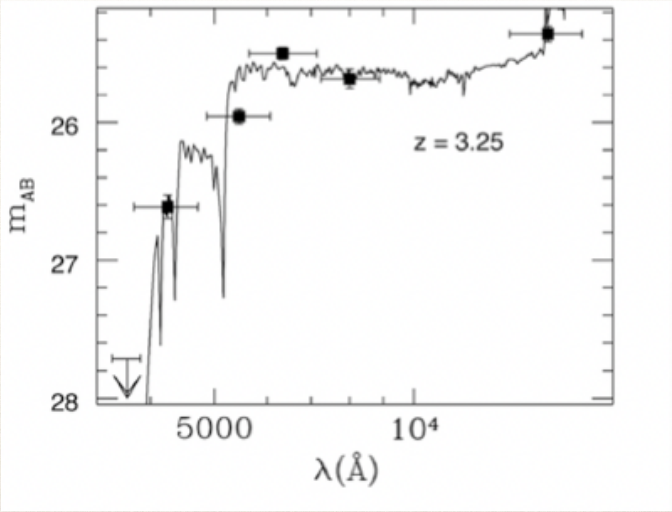
- We will apply both methods to the problem of ***photometric redshifts*** (or "photo-z's"): determining galaxy redshifts (z's) from their colors / magnitudes
- We will work with a compilation of Rubin Observatory LSST-like imaging + DEEP2/3 redshifts from Rongpu Zhou



A quick introduction to photometric redshifts

- Redshift (z) measurements allow us to determine how far back in Universe's history we are looking
- Define z so wavelengths of light are enlarged by a factor of $(1+z)$
  - Study galaxy evolution, cosmology, etc. by measuring properties as a function of redshift
  - To determine well: measure detailed spectrum of light from object; compare observed wavelengths of spectral features to rest-frame values to get z
  - For faint objects, this is infeasible. We instead must rely on measurements of brightness through filters to infer z: "photometric redshifts"





A quick introduction to photometric redshifts

- ***Photometry*** = measuring how bright something is
- To an astronomer, ***magnitude*** is a measure of flux (= energy received per square meter per second per steradian):
  - e.g.: r = r-band magnitude = -2.5 log10 (r flux / r_0) , where r_0 is some reference flux value (the 'zero point' as it corresponds to magnitude 0)
- Bigger flux corresponds to smaller magnitude:
  - In classical Greek astronomy, brightest stars had magnitude 1, next brightest magnitude 2, etc.

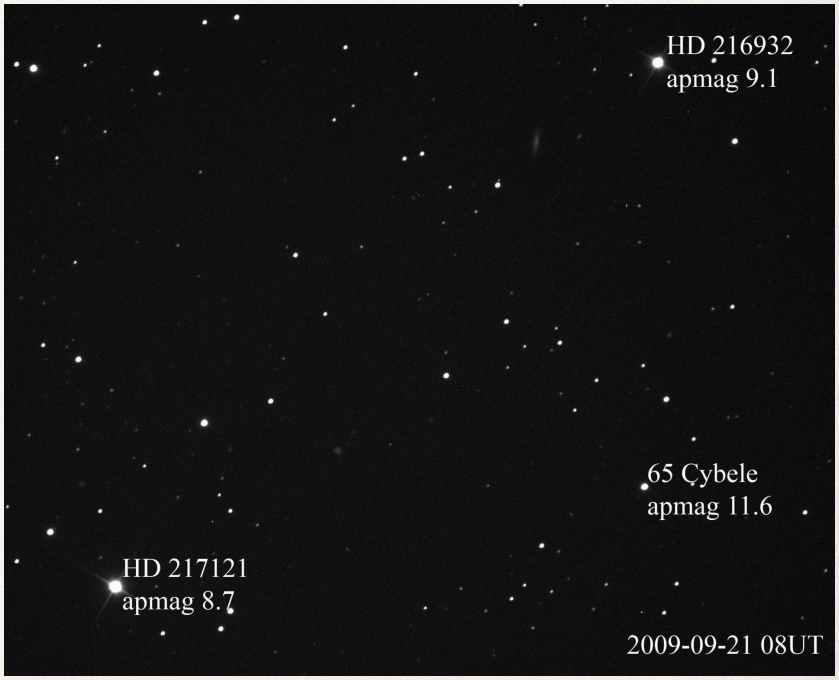
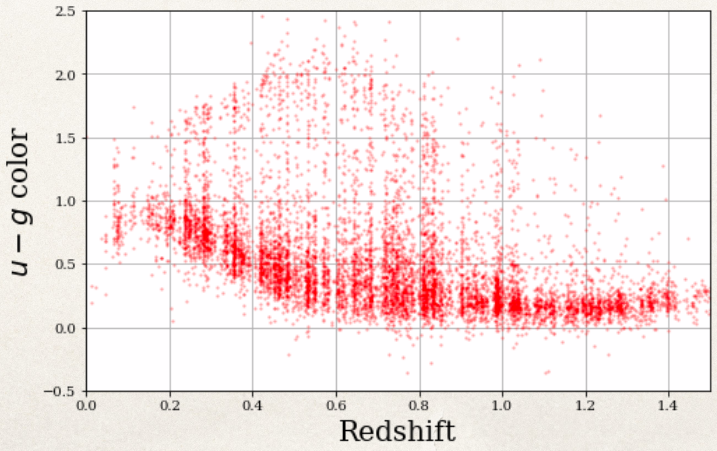


Image from Kevin Heider, wikimedia.org

A quick introduction to photometric redshifts

- In astronomy, ***color*** is the difference between magnitudes in 2 filters (with bluest specified first)
  - e.g.: g - r = g-band magnitude - r-band magnitude = - 2.5 log10 (g flux / r flux)
- We always subtract magnitude in the redder band from magnitude in the bluer filter
- Larger color corresponds to redder color, as bigger magnitudes are fainter
- Colors correlate with redshifts: could predict relationship of redshift to color given models of galaxy spectra and the filters used -- or just train an algorithm to map those relations (machine learning!)



Quantities we will work with

```
Full catalog:  8508  objects
['U', 'G', 'R', 'I', 'Z', 'Y', 'UERR', 'GERR', 'RERR', 'IERR', 'ZERR',
 'YERR', 'RADIUS_ARCSEC', 'ZHELIO', 'MAGB', 'UB_0']
```

- U, G, R, I, Z, and Y are measurements of magnitudes in filters (UV/ green/red/IR/ more IR/ even more IR)
- ZHELIO is a spectroscopic measurement of the redshift $z=(\lambda_{observed}/\lambda_{restframe}) -1$, translated to the rest frame of the Sun (so that it does not vary annually)
  - We can ignore the other quantities today. We want to take the magnitudes, and predict ZHELIO.
- It turns out that colors are more informative about redshift than raw magnitude. We will construct *u-g, g-r, r-i, i-z,* and *z-y* colors

Walking through the code: imports

```
from astropy.table import Table   #astropy routine for reading tables
import pandas   # we'll mostly work with things in Pandas

# Random forest routine from scikit-learn:
from sklearn.ensemble import RandomForestRegressor

# kNN routine from scikit-learn:
from sklearn.neighbors import KNeighborsRegressor

# Cross-Validation routines:
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_predict
```

Walking through the code: reading in data

```
#read in catalog of magnitudes and redshifts
# CHANGE PATH VARIABLE TO POINT TO DIRECTORY WITH THE FILE
path='./'

# easiest way to read in a FITS BINTABLE file containing a database is
# using astropy
cat = Table.read(path+'data_trim.fits.gz')

# we then convert the catalog to Pandas form
cat=cat.to_pandas()


# How big is the catalog?
print('Full catalog: ',len(cat),' objects')

# What information does it contain for each object?
print(cat.columns)
```

Walking through the code: setup for scikit-learn

```
# create convenience arrays for all magnitudes
u_mag = cat.U
g_mag = cat.G
r_mag = cat.R
i_mag = cat.I
z_mag = cat.Z
y_mag = cat.Y

#Redshift array
z = cat.ZHELIO

#vector of redshifts
data_z = z

# Now, set up input data array for scikit-learn regression algorithms
# We will include galaxy colors (expressed as differences between magnitudes
# in adjoining bands) and one magnitude.
# np.column_stack makes a 2D array out of a set of 1d arrays :
# with 6 variables we get an N x 6 numpy array out
data_colmag = np.column_stack((u_mag-g_mag, g_mag-r_mag, r_mag-i_mag,
                               i_mag-z_mag, z_mag-y_mag, i_mag))
data_colmag.shape
```

Walking through the code: rescaling features

```
# For k-nearest neighbor we want to rescale all variables
#    to have mean 0 and variance 1
# This is often, but not always, important for machine
#    learning routines
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# the rescaling requires a fit step...
scaler.fit(data_colmag)

# then we apply the scaler to create a new array of features,
#   all normalized
scaled_colmag = scaler.transform(data_colmag)
```

- Defining a combined distance between two objects in k-nearest neighbor makes little sense if the different features have very different ranges.
- The easy way to handle this is to rescale everything to have similar span. This is often helpful or assumed to have been done when applying machine learning algorithms.
- In **`scikit-learn`**, most routines follow a standard workflow:
  - We set up an instance of a particular algorithm (object); use its **`.fit()`** method to train the algorithm; and use the **`.transform()`** or (more often) **`.predict()`** to apply the algorithm to data.

Walking through the code: Creating instances of each algorithm

```
# We will set up an implementation of the scikit-learn
# RandomForestRegressor in an object called 'regrf'.
regrf = RandomForestRegressor(n_estimators = 50,
                max_depth = 30, max_features = 'auto')


# We need to set up an implementation of the scikit-learn
# KNeighborsRegressor in an object called 'regrn'.
# uniformly weighting 10 nearest neighbors:
regknn = KNeighborsRegressor(n_neighbors = 10, weights='uniform')

# or weighting neighbors inversely proportional to their distance:
regknn_d = KNeighborsRegressor(n_neighbors = 10, weights='distance')
```

- We can set parameters for how an algorithm is constructed when the instance of an object from that method's class is created

Assessing performance: training vs. testing

```
if 0:
    # To better assess the quality of the Random Forest fitting,
    # we split the data into Training (50%) and Test (50%) sets.
    # The code below performs this task on the data_mags and data_z arrays:

    # 1) randomly divide the sample into 50% training and 50% testing sets
    # (e.g., data_train, z_train, and scaled_train are the training
    # portions of data_colmag, data_z, and scaled_colmag

    data_train, data_test, z_train, z_test, scaled_train, scaled_test \
           =train_test_split(data_colmag, data_z, scaled_colmag, \
                        test_size = 0.50, train_size = 0.50)
```

- Any machine learning algorithm will try to optimize performance on the particular training set it is given. This may result in overfitting.
- As a result, an algorithm may perform much better on the data it is provided as input than on new data
- We can assess the performance of an algorithm more realistically by fitting our model with one subset of the data, and testing with another.
- `sklearn.model_selection.train_test_split()` will randomly split up our data arrays into separate, matched training and test sets.

Walking through the code: Doing the learning!

```
    #Train the regressor using the training data
    regrf.fit(data_train,z_train)

    #Apply the regressor to predict values for the test data
    z_phot = regrf.predict(data_test)
    z_spec = z_test

#Make a photo-z/spec-z plot and output summary statistics for the test set.
    plot_and_stats(z_spec,z_phot)
```

- **Evaluate the performance of random forest and kNN from the below code boxes.**
  - Activate the calculations for Random Forest and k-Nearest Neighbors in the following code boxes by changing if 0 to if 1.
- **Which algorithm does better at minimizing scatter? Reducing the fraction of outliers (objects with large deviations)?**

Warning: do not train and test with the same data!

```
if 0:

    # use the RF regressor trained on the training set, but
    #   apply it to the training set instead of the test set
    z_phot = regrf.predict(data_train)
    z_spec = z_train

    plot_and_stats(z_spec,z_phot)
```

- **How do the statistics (NMAD + outlier rate) differ when evaluated on the training set, as compared to when using an independent test set?**

Cross-validation

- With a 50-50 training/testing split, we are always training significantly more poorly than we would with the full dataset, and can only use half the data to evaluate how well we are doing. ***K-fold cross-validation*** provides a way around this.
- In k-fold cross-validation, we split the data into k subsets. We loop over the subsets, training with all but one and testing with the other; in the end, we get the performance of training with a fraction *(k-1)/k* of the data, but are able to get test statistics based on the entire dataset.
- This is easy to do in `scikit-learn`, but does require running the training ***k*** times ...
  - Note: we can search multi-dimensional grids of ML algorithm parameters in an automated way and optimize parameters with cross- validation with `sklearn.model_selection.GridSearchCV`; see https://scikit-learn.org/stable/auto_examples/model_selection/plot_grid_search_digits.html for an example.

Walking through the code: cross-validation

```
if 0:

    # use the RF regressor trained on the training set, but
    #   apply it to the training set instead of the test set
    z_phot = regrf.predict(data_train)
    z_spec = z_train

    plot_and_stats(z_spec,z_phot)
```

- **Compare the performance (NMAD, outlier rate...) from random forest regression with 5-fold cross-validation vs. training with only 50% of the sample.**

What does the plotting code do?

```
#This is a function that makes a plot of photometric redshift
#    as a function of spectroscopic redshift
#    and calculates key statistics.  It will save us a lot of work.
def plot_and_stats(zspec,zphot):

    x = np.arange(0,5.4,0.05)

# define differences of >0.15*(1+z) as non-Gaussian 'outliers'
    outlier_upper = x + 0.15*(1+x)
    outlier_lower = x - 0.15*(1+x)

    mask = np.abs((z_phot - z_spec)/(1 + z_spec)) > 0.15
    notmask = ~mask

#Standard Deviation of the predicted redshifts compared to the data:
    std_result = np.std((z_phot - z_spec)/(1 + z_spec), ddof=1)

#Normalized MAD (Median Absolute Deviation):
    nmad = 1.48 * np.median(np.abs((z_phot - z_spec)/(1 + z_spec) -np.median((z_phot - z_spec)/(1 + z_spec))))

#Percentage of delta-z > 0.15(1+z) outliers:
    eta = np.sum(np.abs((z_phot - z_spec)/(1 + z_spec)) > 0.15)/len(z_spec)

    #Median offset (normalized by (1+z); i.e., bias:
    bias = np.median(((z_phot - z_spec)/(1 + z_spec)))
    sigbias=std_result/np.sqrt(0.64*len(z_phot))
```

What does the plotting code do?

```
# make photo-z/spec-z plot
plt.figure(figsize=(8, 8))

#add lines to indicate outliers
plt.plot(x, outlier_upper, 'k--')
plt.plot(x, outlier_lower, 'k--')
plt.plot(z_spec[mask], z_phot[mask], 'r.', markersize=6,  alpha=0.5)
plt.plot(z_spec[notmask], z_phot[notmask], 'b.',  markersize=6, alpha=0.5)
plt.plot(x, x, linewidth=1.5, color = 'red')
plt.title(f'NMAD:  {nmad:6.4f} Delta z >0.15(1+z) outlier rate:{eta*100:6.3f} %', fontsize=18)
plt.xlim([0.0, 2])
plt.ylim([0.0, 2])
plt.xlabel(r'$z_{\mathrm{spec}}$', fontsize = 27)
plt.ylabel(r'$z_{\mathrm{photo}}$', fontsize = 27)
plt.grid(alpha = 0.8)
plt.tick_params(labelsize=15)
plt.show()
```

Warning: ML methods extrapolate poorly!

```
if 0:
    # split the sample at the median magnitude; note that
    #      smaller magnitude means brighter!
    is_bright = r_mag < 23.15

    # we can use a logical statement as a mask to select only
    #       those array elements where it is true
    data_bright = data_colmag[is_bright]
    z_bright = data_z[is_bright]

    # or negate it to select where it is false
    data_faint=data_colmag[~is_bright]
    z_faint = data_z[~is_bright]

    #train the regressor with the bright data
    regrf.fit(data_bright,z_bright)

    # run on the faint test set
    z_phot = regrf.predict(data_faint)
    z_spec = z_faint

    plot_and_stats(z_spec,z_phot)
```

- In the notebook we present a variety of scenarios in which the training & test data differ in brightness, color, and redshift.
- **Assess: which of these causes the worst problems / worst predictions?**

Random Forest for classification

```
if 0:
    from sklearn.ensemble import RandomForestClassifier

    # set up the classifier object
    classrf = RandomForestClassifier(n_estimators=50)

    # fit a classifier intended to separate objects at
    #       redshift > 0.75 from those at < 0.75
    classrf.fit(data_train,z_train > 0.75)

    # predict the classifications for the test set
    class_predict = classrf.predict(data_test)

    # test which objects are selected as being at high redshift
    ishiz = class_predict == True

    #plot redshift histograms for each sample
    bins = np.linspace(0,1.5,150)
    a,b,c=plt.hist(z_test[ishiz],bins=bins,histtype='step',label = 'High z')
    a,b,c=plt.hist(z_test[~ishiz],bins=bins,histtype='step',label = 'Low z')
    plt.xlabel('Redshift')
    plt.legend()
```

- Classification with machine learning techniques in **`scikit-learn`** works much like regression; but the target array will consist of True and False, instead of a continuous variable.

Optimizing parameters of a scikit-learn algorithm

```
from sklearn.metrics import mean_squared_error,median_absolute_error
if 1:
    ntree_test = 10,25, 50,100
    mse=np.copy(ntree_test)*0.
    mad=np.copy(ntree_test)*0.

    for i,n in enumerate(ntree_test):
        # define the RF object with our choice of number of trees
        regrf = RandomForestRegressor(n_estimators = n, \
                                      max_depth = 30, max_features = 'auto')
        # do training AND prediction on the whole sample
        #     using cross-validation
        predicted = cross_val_predict(regrf,data_colmag,data_z, cv=5)
        #calculate mean squared error
        mse[i]=mean_squared_error(data_z,predicted)
        #calculate MAD
        mad[i]=median_absolute_error(data_z,predicted)

    plt.plot(ntree_test, mse,label='MSE')
    plt.plot(ntree_test, mad,label='NMAD')
    plt.legend()
    plt.xlabel('Number of trees')
```

- In general, it is best to test whether your results could be improved by changing the basic free parameters of the algorithm. You do need to decide what kind of loss you want to optimize though...

Things to keep in mind when applying machine learning methods

- Machine Learning algorithms can be very good at handling data like that which they were trained on, but can extrapolate very poorly
  - In the notebook, you can see what happens when you test with data that is identical to the training set, and also for data that is systematically different in at least some way
  - Always be sure to think about whether the data you will be applying the algorithm to will really match the training data!
- Imbalanced training sets can also be an issue: if 99.9% of the training data is in one class, always outputting that class as the solution could yield a small loss and get favored...