Machine Learning

In this activity, we're going to use decision trees to determine the redshifts of Galaxies from their photometric colours. We'll use galaxies where accurate spectroscopic redshifts have been calculated as our gold standard. We will learn how to assess the accuracy of the decision trees predictions and have a look at validation of our model.

We will also have a quick look at how this problem might be approached without using machine learning. This will highlight some of the limitations of the classical approach and demonstrate why a machine learning approach is ideal for this type of problem.

If you want to run your code offline, you can download the full NumPy dataset for this activity here.

This activity is based on the scikit-learn example on Photometric Redshifts of Galaxies.

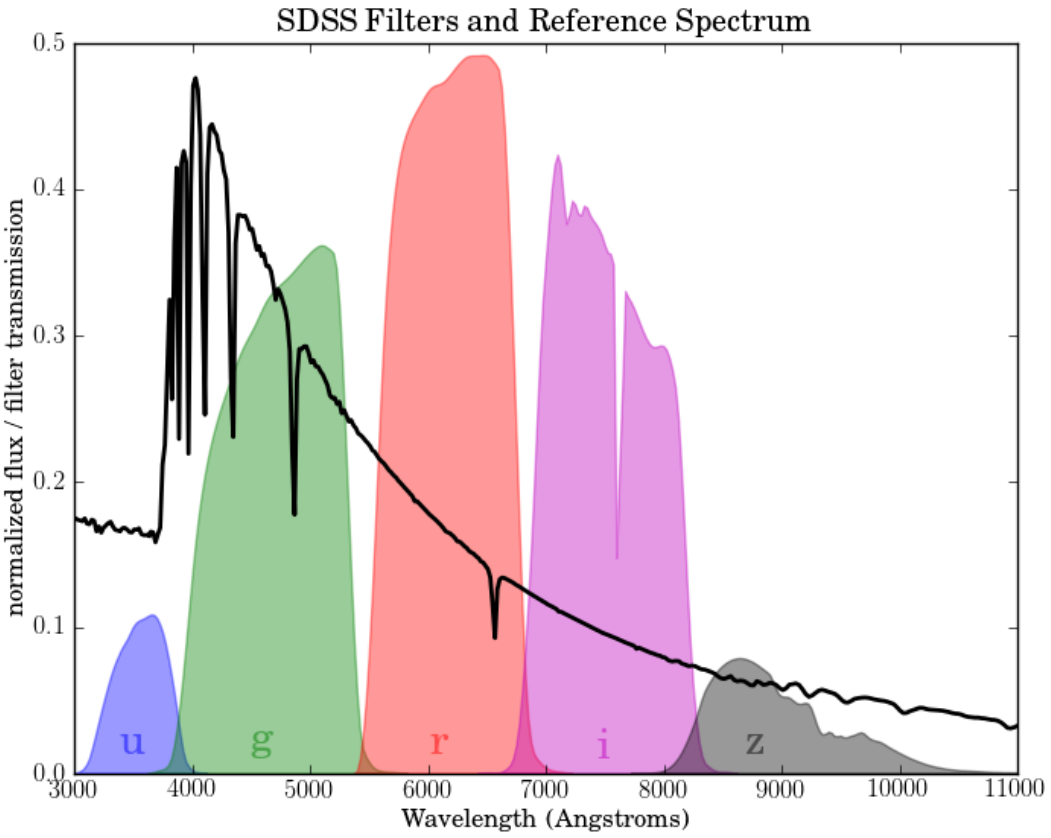
.

We will be using flux magnitudes from the Sloan Digital Sky Survey (SDSS) catalogue to create colour indices. Flux magnitudes are the total flux (or light) received in five frequency bands (*u*, *g*, *r*, *i* and *z*).

PHYSICS SOCIETY

The *astronomical colour* (or *colour index*) is the difference between the magnitudes of two filters, i.e. *u* - *g* or *i* - *z*.

This index is one way to characterise the colours of galaxies. For example, if the u-g index is high then the object is brighter in ultraviolet frequencies than it is in visible green frequencies.



Colour indices act as an approximation for the spectrum of the object and are useful for classifying stars into different types.

S2Z.

To calculate the redshift of a distant galaxy, the most accurate method is to observe the optical emission lines and measure the shift in wavelength. However, this process can be time consuming and is thus infeasible for large samples.

For many galaxies we simply don't have spectroscopic observations.

Instead, we can calculate the redshift by measuring the flux using a number of different filters and comparing this to models of what we expect galaxies to look like at different redshifts.

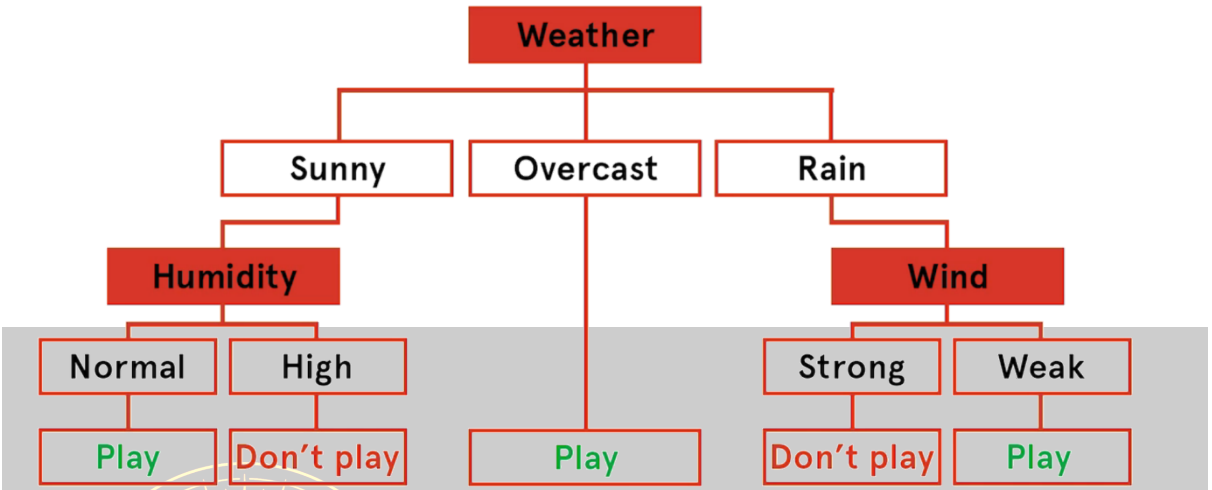
In this activity, we will use machine learning to obtain photometric redshifts for a large sample of galaxies. We will use the colour indices (u-g, g-i, r-i and i-z) as our input and a subset of sources with spectroscopic redshifts as the training dataset.

.

Decision trees are a tool that can be used for both classification and regression. In this module we will look at regression, however, in the next module we will see how they can be used as classifiers.

Decision trees map a set of input features to their corresponding output targets. This is done through a series of individual decisions where each decision represents a node (or branching) of the tree.

The following figure shows the decision tree our proverbial robot tennis player *Robi* used in the lectures to try and decide whether to play tennis on a particular day.



Should we play tennis?

Each node represents a decision that the robot needs to make (or assess) to reach a final decision. In this example, the decision tree will be passed a set of input features (Outlook, Humidity and Wind) and will return an output of whether to play or not.

.

In decision trees for real-world tasks, each decision is typically more complex, involving measured values, not just categories.

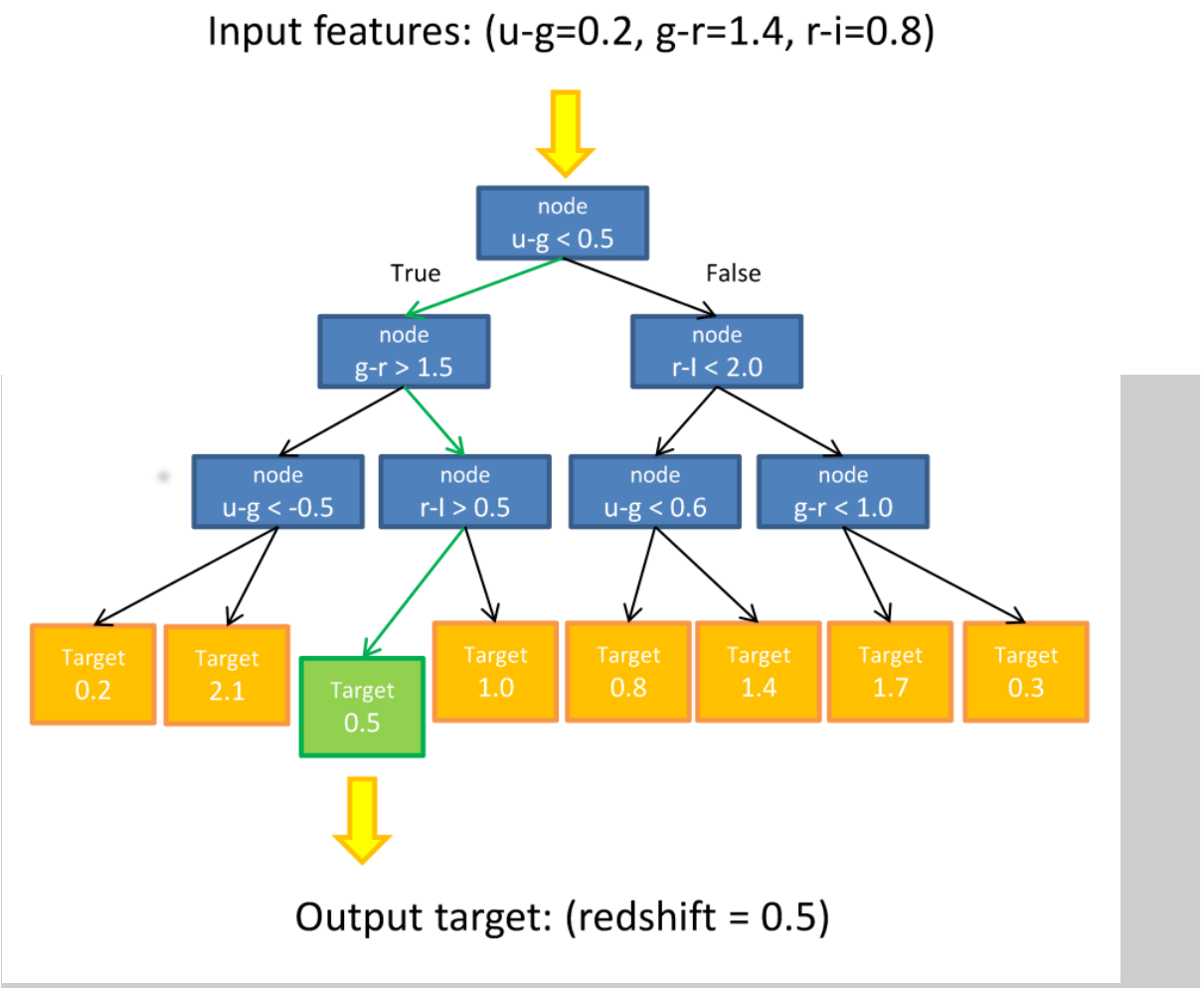
Instead of the input values for **`humidity`** being **`Normal`** or **`High`** and **`wind`** being **`Strong`** or **`Weak`** we might see a percentage between 0 and 100 for humidity and a wind speed in km/hr for wind. Our decisions might then be **`humidity < 40%`** or **`wind < 5 km/hr`**. The output of regression is a real number. So, instead of the two outputs of **`Play`** and **`Don't Play`** we have a probability of whether we will play that day. The decision at each branch is determined from the training data by the decision tree learning algorithm. Each algorithm employs a different metric (e.g. Gini impurity or information gain) to find the decision that splits the data most effectively.

For now, just need to know that a decision tree is a series of decisions, each made on a single feature of the data. The end point of all the branches is a set of desired target values.

.

The inputs to our decision tree are the colour indices from photometric imaging and our output is a photometric redshift. Our training data uses accurate spectroscopic measurements.

The decision tree will look something like the following.



We can see how our calculated colour indices are input as features at the top and through a series of decision nodes a target redshift value is reached and output.

We will be using the Python machine learning library scikit-learn which provides several machine learning algorithms.

The scikit-learn decision tree regression takes a set of input features and the corresponding target values, and constructs a decision tree model that can be applied to new data.

.

We have provided the Sloan data in NumPy binary format (`.npy`) in a file called `sdss_galaxy_colors.npy`. The Sloan data is stored in a NumPy structured array and looks like this:

u	g	r	i	z	...	redshift
19.84	19.53	19.47	19.18	19.11	...	0.54
19.86	18.66	17.84	17.39	17.14	...	0.16
...	...	...	...	...	...	...
18.00	17.81	17.77	17.73	17.73	...	0.47

It also include `spec_class` and redshift_err columns we don't need in this activity.
The data can be loaded using:

```
import numpy as np
data = np.load('sdss_galaxy_colors.npy')
print(data[0])
```

The `data[0]` corresponds to the first row of the table above. Individual named columns can be accessed like this:

```
import numpy as np
data = np.load('sdss_galaxy_colors.npy')
print(data['u'])
```

Each flux magnitude column can be accessed in the `data` array as `data['u']`, `data['g']` etc. The redshifts can accessed with `data['redshift']`.

.

PHYSICS SOCIETY
ARSD COLLEGE, UNIVERSITY OF DELHI

Write a `get_features_targets` function that splits the training data into input `features` and their corresponding `targets`. In our case, the inputs are the 4 colour indices and our targets are the corresponding redshifts.
Your function should return a tuple of:

- `features`: a NumPy array of dimensions $m \times 4$, where m is the number of galaxies;
- `targets`: a 1D NumPy array of length m, containing the redshift for each galaxy.

The data argument will be the structured array described on the previous slide.
The `u` flux magnitudes and redshifts can be accessed as a column with `data['u']` and `data['redshift']`.
The four features are the colours `u - g`, `g - r`, `r - i` and `i - z`. To calculate the first column of `features`, subtract the `u` and `g` columns, like this:

```
import numpy as np
data = np.load('sdss_galaxy_colors.npy')
print(data['u'] - data['g'])
```

The features for the first 2 galaxies in the example data should be:

```
[[ 0.31476  0.0571   0.28991  0.07192]
 [ 1.2002   0.82026  0.45294  0.24665]]
```

And the first 2 targets should be:

```
[ 0.539301   0.1645703]
```

.

Sample solution

decision_tree_regression.py

```
import numpy as np

def get_features_targets(data):
  features = np.zeros(shape=(len(data), 4))
  features[:, 0] = data['u'] - data['g']
  features[:, 1] = data['g'] - data['r']
  features[:, 2] = data['r'] - data['i']
  features[:, 3] = data['i'] - data['z']
  targets = data['redshift']
  return features, targets


if __name__ == "__main__":
  # load the data
  data = np.load('sdss_galaxy_colors.npy')

  # call our function
  features, targets = get_features_targets(data)

  # print the shape of the returned arrays
  print(features[:2])
  print(targets[:2])
```

.

ILLUMINATI
PHYSICS SOCIETY
ARSD COLLEGE, UNIVERSITY OF DELHI

We are now going to use our features and targets to train a decision tree and then make a prediction. We are going to use the DecisionTreeRegressor class from the **sklearn.tree** module.
The decision tree regression learning algorithm is initialised with:

```
dtr = DecisionTreeRegressor()
```

We will discuss some optimisations later in the activity, but for now we are just going to use the default values.

To train the model, we use the **fit** method with the **features** and **targets** we created earlier:

```
dtr.fit(features, targets)
```

The decision tree is now trained and ready to make a prediction:

```
predictions = dtr.predict(features)
```

predictions is an array of predicted values corresponding to each of the input variables in the array.

Your task is to put this together for our photometric redshift data. Copy your **get_features_targets** from the previous problem. Use the comments to guide your implementation.
Finally, print the first 4 predictions. It should print this:

```
[ 0.539301    0.1645703   0.04190006  0.04427702]
```

.

So we trained a decision tree! Great...but how do we know if the tree is actually any good at predicting redshifts?

In regression we compare the predictions generated by our model with the actual values to test how well our model is performing. The difference between the predicted values and actual values (sometimes referred to as residuals) can tell us a lot about where our model is performing well and where it is not.

While there are a few different ways to characterise these differences, in this tutorial we will use the median of the differences between our predicted and actual values. This is given by:

$$\text{med\_diff} = \text{median}(|Y_{i,\text{pred}} - Y_{i,\text{act}}|)$$

Where || denotes the absolute value of the difference.

.

ILLUMINATI PHYSICS SOCIETY

In this problem we will implement the function **`median_diff`**. The function should calculate the median residual error of our model, i.e. the median difference between our predicted and actual redshifts.
The **`median_diff`** function takes two arguments – the predicted and actual/target values. When we use this function later in the tutorial, these will correspond to the predicted redshifts from our decision tree and their corresponding measured/target values.
The median of differences should be calculated according to the formula: $\text{med\_diff} = \text{median}(|Y_{i,\text{pred}} - Y_{i,\text{act}}|)$

.

ample solution

median_differences.py

```
import numpy as np
```

```
# Write a function that calculates the median of the differences between our predicted and actual values
def median_diff(predicted, actual):
  return np.median(np.abs(predicted[:] - actual[:]))

if __name__ == "__main__":
  # load testing data
  targets = np.load('targets.npy')
  predictions = np.load('predictions.npy')

  # call your function to get the median difference between the predicted values and the target values
  diff = median_diff(predictions, targets)

  # print the median difference
  print("Median difference: {:0.3f}".format(diff))
```

Try this solution

.

We previously used the *same data* for training and testing our decision trees.

This gives an unrealistic estimate of how accurate the model will be on previously unseen galaxies because the model has been optimised to get the best results on the training data.

The simplest way to solve this problem is to split our data into *training* and *testing* subsets:

```
# initialise and train the decision tree
dtr = DecisionTreeRegressor()
dtr.fit(train_features, train_targets)

# get a set of prediction from the test input features
predictions = dtr.predict(test_features)

# compare the accuracy of the prediction against the actual values
print(calculate_rmsd(predictions, test_targets))
```

PHYSICS SOCIETY
ARSD COLLEGE, UNIVERSITY OF DELHI

This method of validation is the most basic approach to validation and is called held-out validation. We will use the `med_diff` accuracy measure and hold-out validation in the next problem to assess the accuracy of our decision tree.

.

In this problem, we will use **median_diff** from the previous question to validate the decision tree model. Your task is to complete the **validate_model** function.
The function should split the **features** and **targets** into train and test subsets, like this 50:50 split for **features**:

```
split = features.shape[0]//2
train_features = features[:split]
test_features = features[split:]
```

Your function should then use the training split (**train_features** and **train_targets**) to train the model.

Finally, it should measure the accuracy of the model using **median_diff** on the **test_targets** and the predicted redshifts from **test_features**.
The function should take 3 arguments:

- **model**: the decision tree regressor;
- **features** - the **features** for the data set;
- **targets** - The **targets** for the data set.

.

We use a index (**split_index**) to divide
our **features** and **targets** into **train_features**, **test_features**, **train_targets** and **test_targets**.
We then train the model with the **train_features** and **train_targets** arrays, get some predictions from the **test_features** and get median of the differences between the predicted and actual values (**predicted_redshifts** and **test_targets** respectively).

Sample solution

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor

# paste your get_features_targets function here
def get_features_targets(data):
  features = np.zeros((data.shape[0], 4))
  features[:, 0] = data['u'] - data['g']
  features[:, 1] = data['g'] - data['r']
  features[:, 2] = data['r'] - data['i']
  features[:, 3] = data['i'] - data['z']
  targets = data['redshift']
  return features, targets

# paste your median_diff function here
def median_diff(predicted, actual):
  return np.median(np.abs(predicted - actual))

# write a function that splits the data into training and testing subsets
# trains the model and returns the prediction accuracy with median_diff
def validate_model(model, features, targets):
  # split the data into training and testing
  split = 2*features.shape[0]//3
  train_features, test_features = features[:split], features[split:]
  train_targets, test_targets = targets[:split], targets[split:]

  # train the model
  model.fit(train_features, train_targets)

  # get the predicted_redshifts
  predictions = model.predict(test_features)

  # use median_diff function to calculate the accuracy
  return median_diff(test_targets, predictions)


if __name__ == "__main__":
```

ILLUMINATI
PHYSICS SOCIETY
ARSD COLLEGE, UNIVERSITY OF DELHI

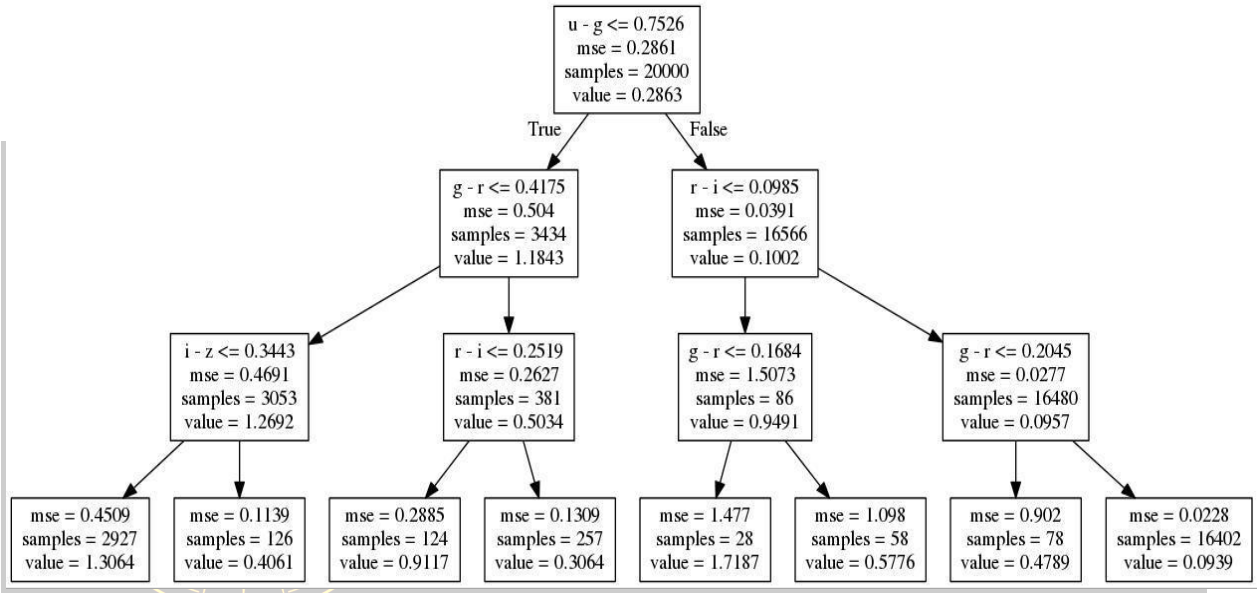
```
data = np.load('sdss_galaxy_colors.npy')
features, targets = get_features_targets(data)

# initialize model
dtr = DecisionTreeRegressor()

# validate the model and print the med_diff
diff = validate_model(dtr, features, targets)
print('Median difference: {:f}'.format(diff))
```

.

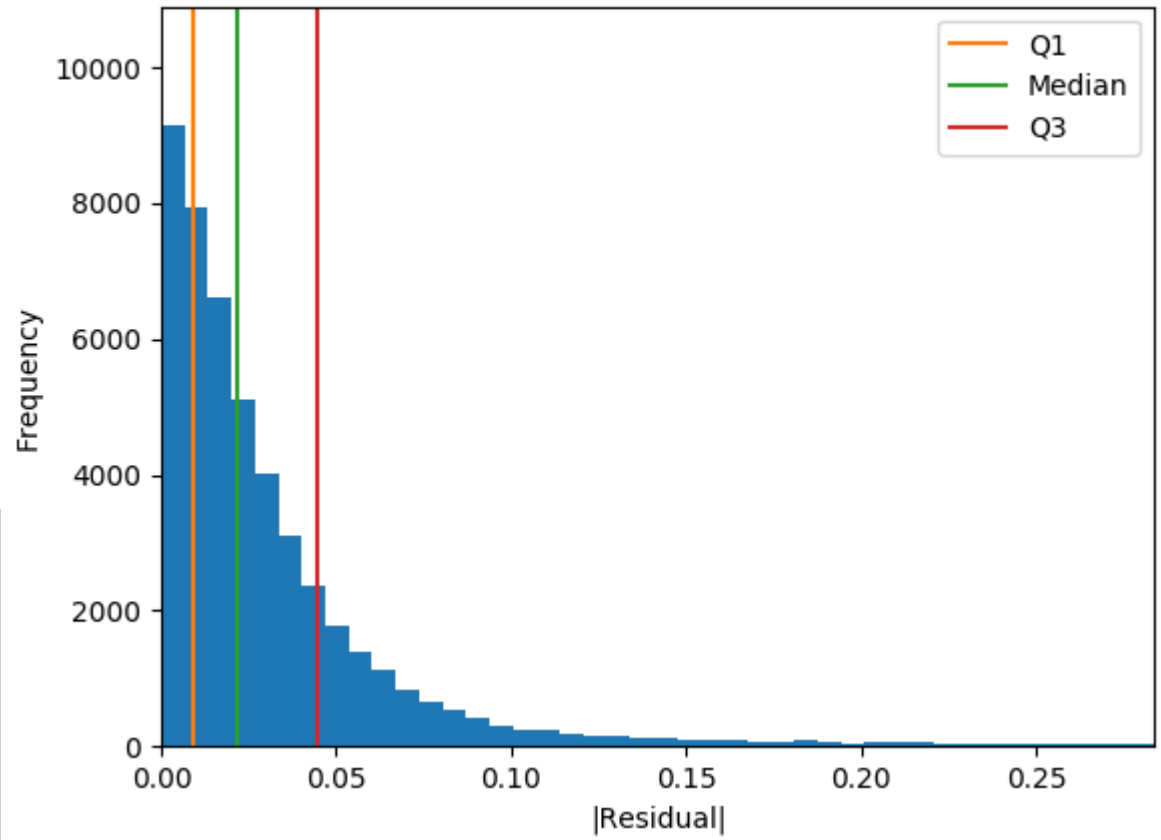
But what does the decision tree actually look like?



Decision tree visualisation. Click to enlarge.

You can see how the decision is made at each node as well as the number of samples which reach that node. We won't go through how to make these plots in the tutorial, but you can download a demo script and data to try at home.

The median of differences of ≈0.02. This means that half of our galaxies have a error in the prediction of <0.02, which is pretty good. One of the reasons we chose the median of differences as our accuracy measure is that it gives a fair representation of the errors especially when the distribution of errors is skewed. The graph below shows the distribution of residuals (differences) for our model along with the median and interquartile values.



Residual distribution. Click to enlarge.

PHYSICS SOCIETY
ARSD COLLEGE, UNIVERSITY OF DELHI

As you can tell the distribution is very skewed. We have zoomed in here, but the tail of the distribution goes all the way out to 6.

.

The number of galaxies we use to train the model has a big impact on how accurate our predictions will be. This is the same with most machine learning methods: the more data that they are trained with, the more accurate their prediction will be.

Here is how our median difference changes with training set size:

Training galaxies	`median_diff`
50	0.048
500	0.026
5000	0.023
50000	0.022

Understanding how the accuracy of the model changes with sample size is important to understanding the limitations of our model. We are approaching the

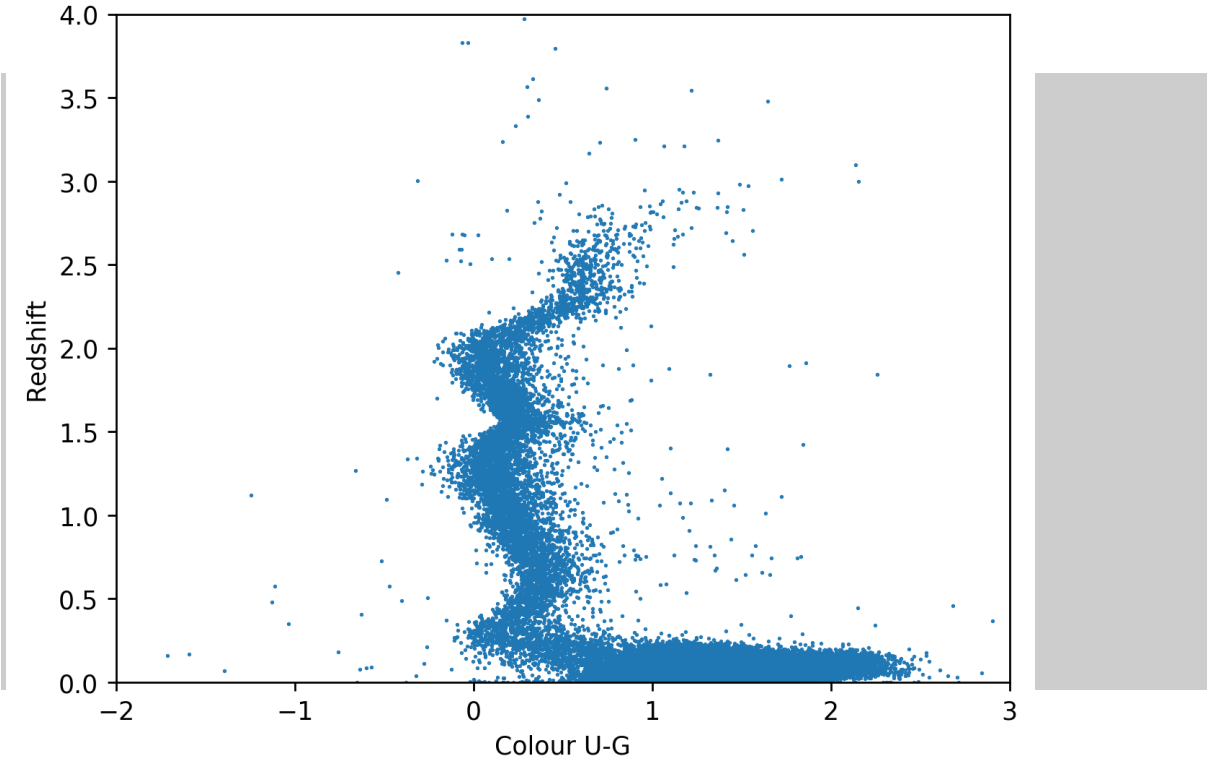
accuracy limit of the decision tree model (for our redshift problem) with a training sample size of 25,000 galaxies.

The only way we could further improve our model would be to use more features. This might include more colour indices or even the errors associated with the measured flux magnitudes.

.

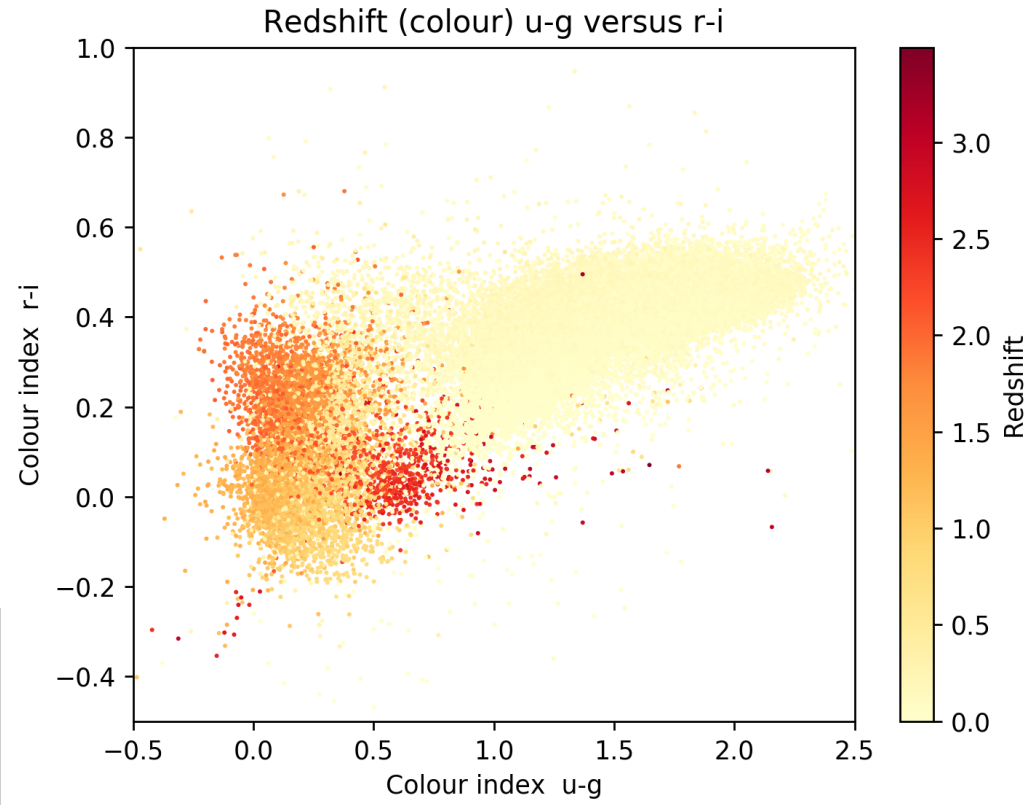
Before machine learning, we would have tried to solve this problem with regression — by constructing an empirical model to predict how the dependent variable (redshift) varies with one or more independent variables (the colour indices).

A plot of how the colours change with redshift tells us that it is quite a complex function, for example redshift versus `u - g`:



One approach would be to construct a multivariate non-linear regression model. Perhaps using a least squares fitting to try and determine the best fit parameters. The model would be quite complex; based on the above plot, a dampened inverse sine function would be a good starting point for such a model.

While we could try such an approach the function would be overly complex and there is no guarantee that it would yield very promising results. Another approach would be to plot a colour-index vs colour-index plot using an additional colour scale to show redshift. The following plot is an example of such a plot.

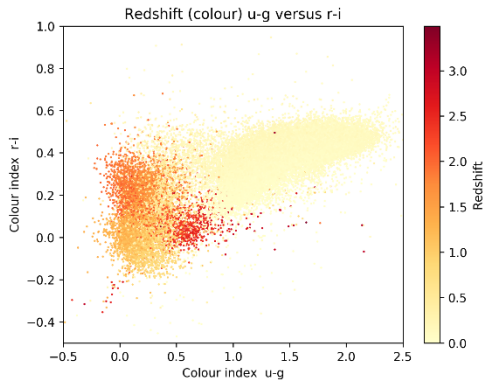


It shows that we get reasonably well defined regions where redshifts are similar. If we were to make a contour map of the redshifts in the colour index vs colour index space we would be able to get an estimate of the redshift for new data points based on a combination of their colour indices. This would lead to redshift estimates with significant uncertainties attached to them.

In the next problem you will re-create the Colour-index vs Colour-index plot with redshift as colour bar.

.

Your task here is simply to try and re-create the following plot.



You should use the **pyplot** module of **matplotlib** which has already been imported and can be accessed through **plt**. In particular you can use the **plt.scatter()** function, with additional arguments **s**, **c** and **cmap**.

We are interested in the **u−g** and **r−i** colour indices.
You can make use of the **plt.colorbar()** function to show your scatter plots colour argument(**c**) to a colour bar on the side of the plot.
Make sure you implement x and y labels and give your plot a title.

.

We start by creating the three data arrays that we want to use in our plot, **u_g**, **r_i** and **redshift**. We then plot our data with...

plot = plt.scatter(u_g, r_i, s=0.5, lw=0, c=redshift, cmap=cmap)
Where **cmap** is the supplied colourmap, **s=0.5** specifies the thickness of the points, lw=0 sets the linewidth to zero so it doesn't saturate the points and **c=redshift** sets the colour of the points. We use the returned plot to add a colorbar in the next line.
cb = plt.colorbar(plot)
We then add a colorbar label with..

cb.set_label('Redshift')
The remainder of the code sets the x and y axis labels as well as the title. We also set limits on both axes to focus on the area of interest.

Sample solution

col_col_redshift.py

```
import numpy as np
from matplotlib import pyplot as plt

# Complete the following to make the plot
if __name__ == "__main__":
    data = np.load('sdss_galaxy_colors_limz.npy')
    # Get a colour map
    cmap = plt.get_cmap('YlOrRd')

    # Define our colour indexes u-g and r-i
    u_g = data['u'] - data['g']
    r_i = data['r'] - data['i']

    # Make a redshift array
    redshift = data['redshift']

    # Create the plot with plt.scatter
    plot = plt.scatter(u_g, r_i, s=0.5, lw=0, c=redshift, cmap=cmap)

    cb = plt.colorbar(plot)
    cb.set_label('Redshift')

    # Define your axis labels and plot title
    plt.xlabel('Colour index  u-g')
    plt.ylabel('Colour index  r-i')
    plt.title('Redshift (colour) u-g versus r-i')

    # Set any axis limits
    plt.xlim(-0.5, 2.5)
    plt.ylim(-0.5, 1)
```

ILLUMINATI PHYSICS SOCIETY ARSD COLLEGE, UNIVERSITY OF DELHI

```
plt.show()
```

.

We will also look at *k*-fold cross validation. This is a more robust method of validation than the held-out method we used previously.

In *k*-fold cross validation, we can test every example once. This is done by splitting the data set into *k* subsets and training/testing the model *k* times using different combinations of the subsets.

Finally, we look at how accurate our model is on QSOs compared with other galaxies. As mentioned in the lectures, QSOs are galaxies that have an Active Galactic Nucleus (AGN). The AGN makes the galaxy brighter and as such they are detectable with the SDSS instruments out to much higher redshifts.

We will use the same data set as the first activity and even some of the functions we wrote in previous questions.

.

Decision trees have many advantages: they are simple to implement, easy to interpret, the data doesn't require too much preparation, and they are reasonably efficient computationally.

Naïvely we'd expect, the deeper the tree, the better it should perform. However, as the model overfits we see a difference in its accuracy on the training data and the more general testing data.

We can control the depth of decision tree learned, using an argument to **DecisionTreeRegressor**. For example, to set the maximum depth to 5:

```
dtr = DecisionTreeRegressor(max_depth=5)
```

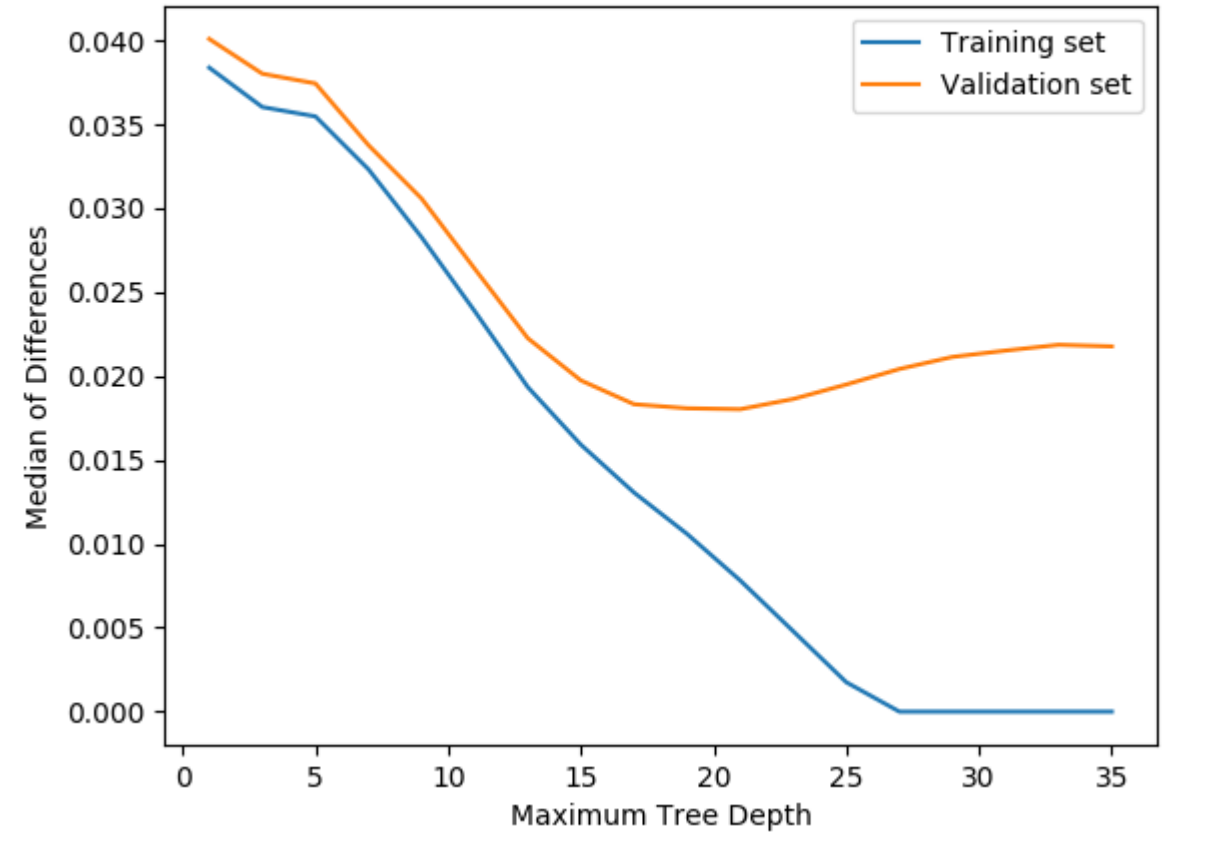
.

Complete the function **`accuracy_by_treedepth`**. The function should return the median difference for both the testing and training data sets for each of the tree depths in **`depths`**.
`accuracy_by_treedepth` should take the following arguments:

- **`features`** and **`targets`** (as in previous problems);
- **`depths`**: an array of tree depths to be used as the **`max_depth`** of the decision tree regressor.

Your function should return two lists (or arrays) containing the **`median_diff`** values for the predictions made on the training and test sets using the maximum tree depths given by the **`depths`**.

For example, if `depths` is `[3, 5, 7]`, then your function should return two lists of length 3. You can choose the size of the split between your testing and training data (if in doubt, 50:50 is fine).
We've included code to plot the differences as a function of tree depths. You should take a moment to familiarise yourself with what each line is doing. If your code is working well then your plot should look a bit like the following:



.

Sample solution

solution.py

```
import numpy as np
from matplotlib import pyplot as plt
from sklearn.tree import DecisionTreeRegressor

# paste your get_features_targets function here
def get_features_targets(data):
```

```
  features = np.zeros((data.shape[0], 4))
  features[:, 0] = data['u'] - data['g']
  features[:, 1] = data['g'] - data['r']
  features[:, 2] = data['r'] - data['i']
  features[:, 3] = data['i'] - data['z']
  targets = data['redshift']
  return features, targets

# paste your median_diff function here
def median_diff(predicted, actual):
  return np.median(np.abs(predicted - actual))

# Complete the following function
def accuracy_by_treedepth(features, targets, depths):
  # split the data into testing and training sets
  split = features.shape[0]//2
  train_features, test_features = features[:split], features[split:]
  train_targets, test_targets = targets[:split], targets[split:]

  # Initialise arrays or lists to store the accuracies for the below loop
  train_diffs = []
  test_diffs = []

  # Loop through depths
  for depth in depths:
    # initialize the model with the maximum depth.
    dtr = DecisionTreeRegressor(max_depth=depth)

    # train the model using the training set
    dtr.fit(train_features, train_targets)

    # Get the predictions for the training set and calculate their med_diff
    predictions = dtr.predict(train_features)
    train_diffs.append(median_diff(train_targets, predictions))

    # Get the predictions for the testing set and calculate their med_diff
    predictions = dtr.predict(test_features)
    test_diffs.append(median_diff(test_targets, predictions))

  # Return the accuracies for the training and testing sets
  return train_diffs, test_diffs

if __name__ == "__main__":
  data = np.load('sdss_galaxy_colors.npy')
  features, targets = get_features_targets(data)

  # Generate several depths to test
  tree_depths = [i for i in range(1, 36, 2)]

  # Call the function
  train_med_diffs, test_med_diffs = accuracy_by_treedepth(features, targets, tree_depths)
  print("Depth with lowest median difference :
{}".format(tree_depths[test_med_diffs.index(min(test_med_diffs))]))

  # Plot the results
  train_plot = plt.plot(tree_depths, train_med_diffs, label='Training set')
  test_plot = plt.plot(tree_depths, test_med_diffs, label='Validation set')
  plt.xlabel("Maximum Tree Depth")
```

ILLUMINATI
PHYSICS SOCIETY
ARSD COLLEGE, UNIVERSITY OF DELHI

```
plt.ylabel("Median of Differences")
plt.legend()
plt.show()
```

.

The method we used to validate our model so far is known as hold-out validation. Hold out validation splits the data in two, one set to test with and the other to train with. Hold out validation is the most basic form of validation.

While hold-out validation is better than no validation, the measured accuracy (i.e. our median of differences) will vary depending on how we split the data into testing and training subsets. The `med_diff` that we get from one randomly sampled training set will vary to that of a different random training set of the same size.
In order to be more certain of our models accuracy we should use k-fold cross validation. k-fold validation works in a similar way to hold-out except that we split the data into k subsets. We train and test the model k times, recording the accuracy each time. Each time we use a different combination of $k-1$ subsets to train the model and the final k^{th} subset to test. We take the average of the k accuracy measurements to be the overall accuracy of the the model.

.

KFold

ILLUMINATI
PHYSICS SOCIETY
ARSD COLLEGE, UNIVERSITY OF DELHI

- **model**, **features**, and **targets** as in previous problems;
- **k** in our *k*-fold. This is the number of subsets to train and test.

KFolds usage

We have created the **KFold** object to give you a set of training and testing indices for each of the *k* runs. It is worth taking a moment to understand this.
Specifically, the object is initialised with

```
kf = KFold(n_splits=k, shuffle=True)
```

The **n_splits=k** passes our desired number of subsets/folds. We want to shuffle the data (as previously explained). The iterator is then used with:

```
for train_indices, test_indices in kf.split(features):
```

The **kf.split(features)** is an iterator that, for each of the *k* iterations, returns two arrays of indices to be used with our feature and target arrays,
i.e. **features[train_indices], targets[train_indices]**

.

This is quite similar to our earlier implementation for validation in the last tutorial. The main differences are that we are using the **train_indices** and **test_indices** to split our features and target arrays.
Another difference is that we are collecting our calculated **med_diff** values in a list to be returned at the end of the function.

Sample solution

solution.py

```
import numpy as np
from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeRegressor

# paste your get_features_targets function here
def get_features_targets(data):
  features = np.zeros((data.shape[0], 4))
  features[:, 0] = data['u'] - data['g']
  features[:, 1] = data['g'] - data['r']
  features[:, 2] = data['r'] - data['i']
  features[:, 3] = data['i'] - data['z']
  targets = data['redshift']
  return features, targets

# paste your median_diff function here
def median_diff(predicted, actual):
  return np.median(np.abs(predicted - actual))

# complete this function
def cross_validate_model(model, features, targets, k):
  kf = KFold(n_splits=k, shuffle=True)
  # initialise a list to collect median_diffs for each iteration of the loop below
  diffs = []

  for train_indices, test_indices in kf.split(features):
    train_features, test_features = features[train_indices], features[test_indices]
    train_targets, test_targets = targets[train_indices], targets[test_indices]

    # fit the model for the current set
    model.fit(train_features, train_targets)

    # predict using the model
    predictions = model.predict(test_features)

    # calculate the median_diff from predicted values and append to results array
    diffs.append(median_diff(predictions, test_targets))

  # return the list with your median difference values
  return diffs
```

ILLUMINATI
PHYSICS SOCIETY
ARSD COLLEGE, UNIVERSITY OF DELHI

```
if __name__ == "__main__":
  data = np.load('./sdss_galaxy_colors.npy')
  features, targets = get_features_targets(data)

  # initialize model with a maximum depth of 19
  dtr = DecisionTreeRegressor(max_depth=19)

  # call your cross validation function
  diffs = cross_validate_model(dtr, features, targets, 10)

  # Print the values
  print('Differences: {}'.format(', '.join(['{:.3f}'.format(val) for val in diffs])))
  print('Mean difference: {:.3f}'.format(np.mean(diffs)))
```

Try this solution

.

Cross validation is an important part of ensuring that our model is returning values that are at least partially accurate. The problem with held-out validation is that we are only able to get prediction values for the data in our test set.

With *k*-fold cross validation each galaxy is tested at least once and because of this we are able to get a prediction value for every galaxy. We'll do this in the next question...

.

Complete the function `cross_validate_predictions`. This is very similar to the previous question except instead of returning the `med_diff` accuracy measurements we would like to return a predicted value for each of the galaxies. The function takes the same 4 arguments as the previous question, i.e. `model`, `features`, `targets` and `k`.
Your function should return a single variable. The returned variable should be a 1-D numpy array of length m, where m is the number of galaxies in our data set. You should make sure that you maintain the order of galaxies when giving your predictions, such that the first prediction in your array corresponds to the first galaxy in the `features` and `targets` arrays.
.

This is very similar to the previous problem. Here instead of using the predicted values for calculating the accuracy we simply put them into an array which the function returns. We initialise an array to store the predictions from each validation with...

```
all_predictions = np.zeros(shape = (len(targets)))
```

We then use the `test_indices` to keep the correct order when populating the array.

```
all_predictions[test_indices] = predicted
```

This ensures that we can compare the predictions of their corresponding target values later when calculating the median difference and plotting the predicted values against actual values.

Sample solution

solution.py

```
import numpy as np
from matplotlib import pyplot as plt
from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeRegressor

# paste your get_features_targets function here
def get_features_targets(data):
  features = np.zeros((data.shape[0], 4))
  features[:, 0] = data['u'] - data['g']
  features[:, 1] = data['g'] - data['r']
  features[:, 2] = data['r'] - data['i']
  features[:, 3] = data['i'] - data['z']
  targets = data['redshift']
  return features, targets

# paste your median_diff function here
def median_diff(predicted, actual):
  return np.median(np.abs(predicted - actual))

# complete this function
def cross_validate_predictions(model, features, targets, k):
  kf = KFold(n_splits=k, shuffle=True)

  # declare an array for predicted redshifts from each iteration
  all_predictions = np.zeros_like(targets)

  for train_indices, test_indices in kf.split(features):
    # split the data into training and testing
    train_features, test_features = features[train_indices], features[test_indices]
    train_targets, test_targets = targets[train_indices], targets[test_indices]

    # fit the model for the current set
    model.fit(train_features, train_targets)

    # predict using the model
    predictions = model.predict(test_features)
```

ILLUMINATI
PHYSICS SOCIETY
ARSD COLLEGE, UNIVERSITY OF DELHI

```
    # put the predicted values in the all_predictions array defined above
    all_predictions[test_indices] = predictions

  # return the predictions
  return all_predictions


if __name__ == "__main__":
  data = np.load('./sdss_galaxy_colors.npy')
  features, targets = get_features_targets(data)

  # initialize model
  dtr = DecisionTreeRegressor(max_depth=19)

  # call your cross validation function
  predictions = cross_validate_predictions(dtr, features, targets, 10)

  # calculate and print the Median Difference as a sanity check
  diffs = median_diff(predictions, targets)
  print('Median difference: {:.3f}'.format(diffs))

  # plot the results to see how well our model looks
  plt.scatter(targets, predictions, s=0.4)
  plt.xlim((0, targets.max()))
  plt.ylim((0, predictions.max()))
  plt.xlabel('Measured Redshift')
  plt.ylabel('Predicted Redshift')
  plt.show()
```

.

ILLUMINATI
PHYSICS SOCIETY
ARSD COLLEGE, UNIVERSITY OF DELHI

Classification vs Regression

In classification, the predictions are from a fixed set of classes, whereas in regression the prediction typically corresponds to a continuum of possible values.

In regression, we measure accuracy by looking at the size of the differences between the predicted values and the actual values. In contrast, in classification problems a prediction can either be correct or incorrect. This makes measuring the accuracy of our model a lot simpler.

In terms of implementation using decision trees, there is very little difference between classification and regression. The only notable difference is that our targets are classes rather than real values. When calculating the accuracy, we check whether the predicted class matches as the actual class.

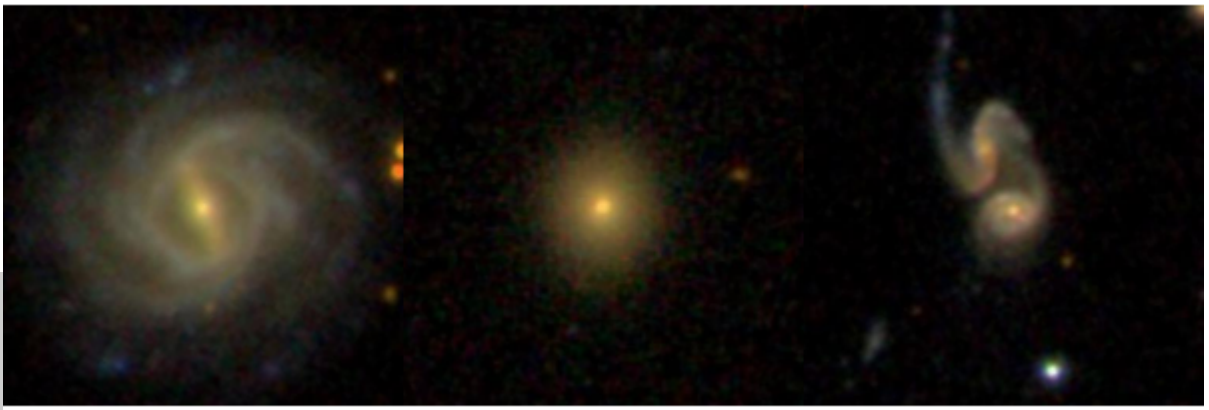
A note on decision tree regression

In decision tree regression, the possible outputs are a finite set of values that correspond to the number of leaves/end points in the tree. Ideally we want as many

points as possible to give a good approximation of the 'continuous' parameter space, whilst avoiding overfitting.

.

In the last activity, you were a human classifier for the Galaxy Zoo project and probably saw a wide range of galaxy types observed by the Sloan Digital Sky Survey. In this activity, we will limit our dataset to three types of galaxy: **spirals**, **ellipticals** and **mergers**.



The three galaxy classes. Click to see full size.

The galaxy catalogue we are using is a sample of galaxies where at least 20 human classifiers (such as yourself) have come to a consensus on the galaxy type.

Examples of spiral and elliptical galaxies were selected where there was a unanimous classification. Due to low sample numbers, we included merger examples where at least 80% of human classifiers selected the merger class.

We need this high quality data to train our classifier.

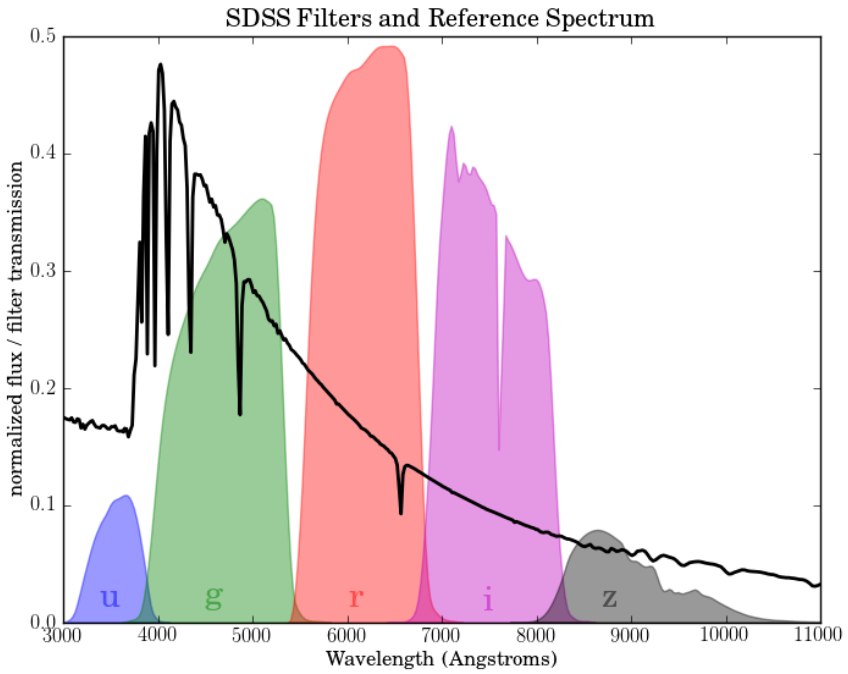
.

Just like in the regression activities, we need to decide on a set of key features that represent our data.

While approaches exist that determine their own feature representation and use the raw pixel values as inputs, e.g. neural networks and deep learning, the majority of existing machine learning in astronomy requires an expert to design the feature set.

In this activity we will be using a set of features derived from fitting images according to known galaxy profiles.

Most of the features we use here are based on the five observed flux magnitudes from the Sloan Digital Sky Survey filters:



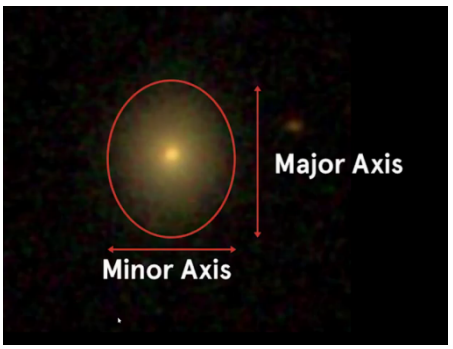
Sloan Digital Sky Servey filters. Click to enlarge.

.

The features that we will be using to do our galaxy classification are *colour index*, *adaptive moments*, *eccentricities* and *concentrations*. These features are provided as part of the SDSS catalogue.

We briefly describe these below. Further information how they are calculated can be found here.

Colour indices are the same colours (`u-g`, `g-r`, `r-i`, and `i-z`) we used for regression. Studies of galaxy evolution tell us that spiral galaxies have younger star populations and therefore are 'bluer' (brighter at lower wavelengths). Elliptical galaxies have an older star population and are brighter at higher wavelengths ('redder').
Eccentricity approximates the shape of the galaxy by fitting an ellipse to its profile. Eccentricity is the ratio of the two axes (semi-major and semi-minor). The De Vaucouleurs model was used to attain these two axes. To simplify our experiments, we will use the median eccentricity across the 5 filters.



Adaptive moments also describe the shape of a galaxy. They are used in image analysis to detect similar objects at different sizes and orientations. We use the fourth moment here for each band.

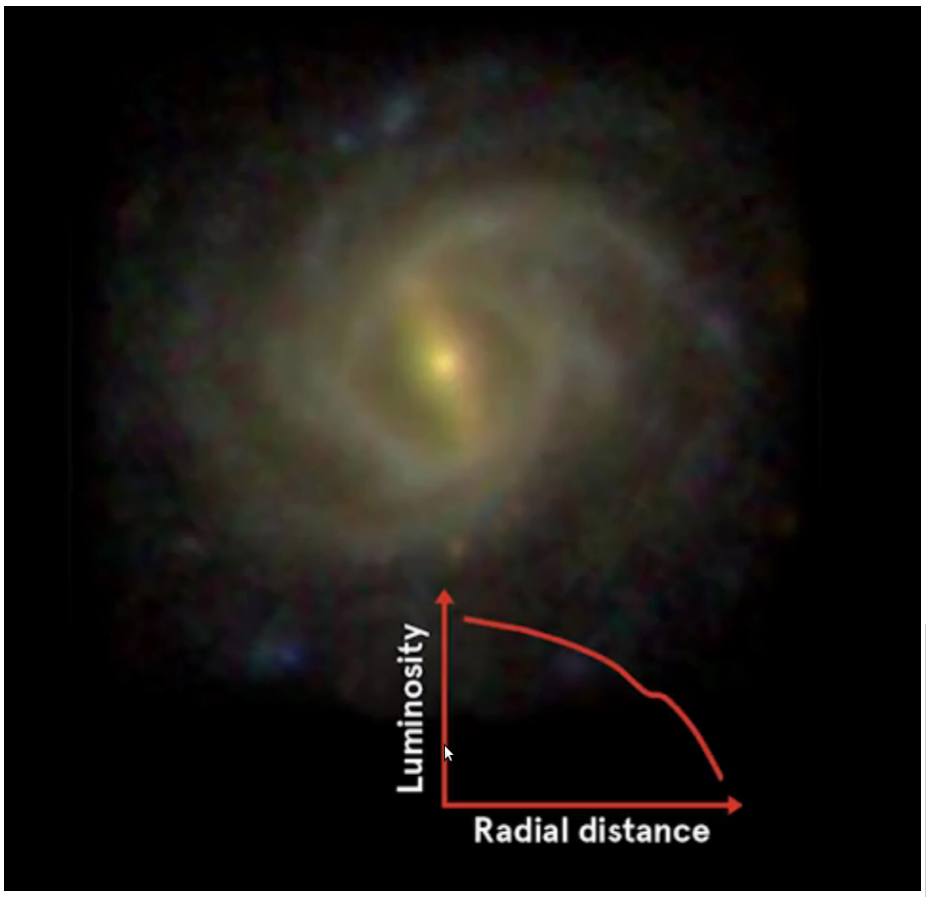
Concentration is similar to the luminosity profile of the galaxy, which measures what proportion of a galaxy's total light is emitted within what radius. A simplified way to represent this is to take the ratio of the radii containing 50% and 90% of the Petrosian flux.

The Petrosian method allows us to compare the radial profiles of galaxies at different distances. If you are interested, you can read more here on the need for Petrosian approach.

For these experiments, we will define concentration as:

$$conc = petro_{R50} petro_{R90}$$

We will use the concentration from the `u`, `r` and `z` bands.



ILLUMINATI
PHYSICS SOCIETY

We have extracted the SDSS and Galaxy Zoo data for 780 galaxies into a NumPy binary file. You can load the file using:

```
import numpy as np
data = np.load('galaxy_catalogue.npy')
```

The `data` is a NumPy array of 780 records. Each record is a single galaxy. You can access the columns using field names. For example, to access the `u-g` colour filter for the first galaxy, you use:

```
data[0]['u-g']
```

Here's a snippet that prints the field and values for the first galaxy:

```
import numpy as np
data = np.load('galaxy_catalogue.npy')
for name, value in zip(data.dtype.names, data[0]):
  print('{:10} {:.6}'.format(name, value))
```

It shows the four colour features, median eccentricity, fourth adaptive moment of each filter, the Petrosian fluxes at a radius of 50% and 90% for the `u`, `r` and `z` filters, and finally the class:

```
u-g        1.85765
g-r        0.67158
r-i        0.4231
```

```
i-z       0.3061
ecc       0.585428
m4_u      2.25195
m4_g      2.33985
m4_r      2.38065
m4_i      2.35974
m4_z      2.39553
petroR50_u 3.09512
petroR50_r 3.81892
petroR50_z 3.82623
petroR90_u 5.17481
petroR90_r 8.26301
petroR90_z 11.4773
class     merger
```

NumPy also allows you to access a field for all of the rows at once, i.e. a column, using the field's name:

```
data['u-g']
```

.

To start, we need to split the data into training and testing sets.

Your task is to implement the **splitdata_train_test** function. It takes a NumPy array and splits it into a training and testing NumPy array based on the specified training fraction. The function takes two arguments and should return two values:

Arguments

ARSD COLLEGE, UNIVERSITY OF DELHI

- **data**: the NumPy array containing the galaxies in the form described in the previous slide;
- **fraction_training**: the fraction of the data to use for training. This will be a float between 0 and 1.

The number of training rows should be truncated if necessary. For example, with a fraction of 0.67 and our 780 galaxies, the number of training rows is **780*0.67 = 722.6**, which should be truncated to **722** using `int`. The remaining rows should be used for testing.

Return values

- **training_set**: the first value is a NumPy array training set;
- **testing_set**: the second value is a NumPy array testing set.

Using the supplied driver code, and our input data and a fraction of 0.7, the program should print the following values:

```
Number data galaxies: 780
Train fraction: 0.7
Number of galaxies in training set: 546
Number of galaxies in testing set: 234
```

Good practice: randomize the dataset order

You shouldn't assume that the data has already been shuffled. If you look at **data['class']** you will see that the **merger**, **elliptical** and **spiral** examples appear together. Failing to shuffle the data will produce a very bad classifier! You can use:
np.random.seed(0)
np.random.shuffle(data)
The first statement ensures the shuffle is the same for each experiment, so you get consistent results, the second shuffles the rows of the data array in place.

.

Here we define a **split_index** that can be used to separate the **data** array into two the training and testing sets respectively.
The **split_index** is determined by multiplying the **fraction_training** and multiplying it by the total number of data points. Converting to an integer will automatically truncate any fractional component of the floating point value.
We also use **np.random.shuffle** to shuffle the position of the indexes. This is not strictly required, but it is always a good idea to prevent selection bias.

Sample solution

```
import numpy as np

def splitdata_train_test(data, fraction_training):
  # complete this function
  np.random.shuffle(data)
  split_index = int(fraction_training*len(data))
  return data[:split_index], data[split_index:]

if __name__ == "__main__":
  data = np.load('galaxy_catalogue.npy')

  # set the fraction of data which should be in the training set
  fraction_training = 0.7

  # split the data using your function
  training, testing = splitdata_train_test(data, fraction_training)

  # print the key values
  print('Number data galaxies:', len(data))
  print('Train fraction:', fraction_training)
  print('Number of galaxies in training set:', len(training))
  print('Number of galaxies in testing set:', len(testing))
```

Try this solution

ILLUMINATI PHYSICS SOCIETY
HINDU COLLEGE, UNIVERSITY OF DELHI