In [1]:

```
from __future__ import division, print_function, absolute_import, unicode_literals
```

Introduction to Machine Learning:

Examples of Unsupervised and Supervised Machine-Learning Algorithms

Version 0.1

Broadly speaking, machine-learning methods constitute a diverse collection of data-driven algorithms designed to classify/characterize/analyze sources in multi-dimensional spaces. The topics and studies that fall under the umbrella of machine learning is growing, and there is no good catch-all definition. The number (and variation) of algorithms is vast, and beyond the scope of these exercises. While we will discuss a few specific algorithms today, more importantly, we will explore the scope of the two general methods: unsupervised learning and supervised learning and introduce the powerful (and dangerous?) Python package `scikit-learn` (http://scikit-learn.org/stable/).

By AA Miller (Jet Propulsion Laboratory, California Institute of Technology.)

(c) 2016 California Institute of Technology. Government sponsorship acknowledged.

In [2]:

```
import numpy as np
from astropy.table import Table
import matplotlib.pyplot as plt
%matplotlib inline
```

Problem 1) Introduction to `scikit-learn`

At the most basic level, `scikit-learn` makes machine learning extremely easy within Python. By way of example, here is a short piece of code that builds a complex, non-linear model to classify sources in the Iris data set that we learned about yesterday:

```
from sklearn import datasets
from sklearn.ensemble import RandomForestClassifier
iris = datasets.load_iris()
RFclf = RandomForestClassifier().fit(iris.data, iris.target)
```

Those 4 lines of code have constructed a model that is superior to any system of hard cuts that we could have encoded while looking at the multidimensional space. This can be fast as well: execute the dummy code in the cell below to see how "easy" machine-learning is with `scikit-learn`.

In [6]:

```
# execute dummy code here
```

Generally speaking, the procedure for `scikit-learn` is uniform across all machine-learning algorithms. Models are accessed via the various modules (`ensemble`, `SVM`, `neighbors`, etc), with user-defined tuning parameters. The features (or data) for the models are stored in a 2D array, `X`, with rows representing individual sources and columns representing the corresponding feature values. [In a minority of cases, `X`, represents a similarity or distance matrix where each entry represents the distance to every other source in the data set.] In cases where there is a known classification or scalar value (typically supervised methods), this information is stored in a 1D array `y`.

Unsupervised models are fit by calling `.fit(X)` and supervised models are fit by calling `.fit(X, y)`. In both cases, predictions for new observations, `Xnew`, can be obtained by calling `.predict(Xnew)`. Those are the basics and beyond that, the details are algorithm specific, but the documentation for essentially everything within `scikit-learn` is excellent, so read the docs.

To further develop our intuition, we will now explore the Iris dataset a little further.

Problem 1a What is the pythonic type of `iris`?

In [11]:

```
type(iris)
```

Out[11]:

```
sklearn.datasets.base.Bunch
```

You likely haven't encountered a `scikit-learn Bunch` before. It's functionality is essentially the same as a dictionary.

Problem 1b What are the keys of iris?

In [12]:

```
iris.keys()
```

Out[12]:

```
['target_names', 'data', 'target', 'DESCR', 'feature_names']
```

Most importantly, iris contains `data` and `target` values. These are all you need for `scikit-learn`, though the feature and target names and description are useful.

Problem 1c What is the shape and content of the `iris` data?

In [13]:

```
print(np.shape(iris.data))
print(iris.data)
```

```
(150, 4)
[[ 5.1  3.5  1.4  0.2]
 [ 4.9  3.   1.4  0.2]
 [ 4.7  3.2  1.3  0.2]
 [ 4.6  3.1  1.5  0.2]
 [ 5.   3.6  1.4  0.2]
 [ 5.4  3.9  1.7  0.4]
 [ 4.6  3.4  1.4  0.3]
 [ 5.   3.4  1.5  0.2]
 [ 4.4  2.9  1.4  0.2]
 [ 4.9  3.1  1.5  0.1]
 [ 5.4  3.7  1.5  0.2]
 [ 4.8  3.4  1.6  0.2]
 [ 4.8  3.   1.4  0.1]
 [ 4.3  3.   1.1  0.1]
 [ 5.8  4.   1.2  0.2]
 [ 5.7  4.4  1.5  0.4]
 [ 5.4  3.9  1.3  0.4]
 [ 5.1  3.5  1.4  0.3]
 [ 5.7  3.8  1.7  0.3]
 [ 5.1  3.8  1.5  0.3]
 [ 5.4  3.4  1.7  0.2]
 [ 5.1  3.7  1.5  0.4]
 [ 4.6  3.6  1.   0.2]
 [ 5.1  3.3  1.7  0.5]
 [ 4.8  3.4  1.9  0.2]
 [ 5.   3.   1.6  0.2]
 [ 5.   3.4  1.6  0.4]
 [ 5.2  3.5  1.5  0.2]
 [ 5.2  3.4  1.4  0.2]
 [ 4.7  3.2  1.6  0.2]
 [ 4.8  3.1  1.6  0.2]
 [ 5.4  3.4  1.5  0.4]
 [ 5.2  4.1  1.5  0.1]
 [ 5.5  4.2  1.4  0.2]
 [ 4.9  3.1  1.5  0.1]
 [ 5.   3.2  1.2  0.2]
 [ 5.5  3.5  1.3  0.2]
 [ 4.9  3.1  1.5  0.1]
 [ 4.4  3.   1.3  0.2]
 [ 5.1  3.4  1.5  0.2]
 [ 5.   3.5  1.3  0.3]
 [ 4.5  2.3  1.3  0.3]
 [ 4.4  3.2  1.3  0.2]
 [ 5.   3.5  1.6  0.6]
 [ 5.1  3.8  1.9  0.4]
 [ 4.8  3.   1.4  0.3]
 [ 5.1  3.8  1.6  0.2]
```

```
 [ 4.6  3.2  1.4  0.2]
 [ 5.3  3.7  1.5  0.2]
 [ 5.   3.3  1.4  0.2]
 [ 7.   3.2  4.7  1.4]
 [ 6.4  3.2  4.5  1.5]
 [ 6.9  3.1  4.9  1.5]
 [ 5.5  2.3  4.   1.3]
 [ 6.5  2.8  4.6  1.5]
 [ 5.7  2.8  4.5  1.3]
 [ 6.3  3.3  4.7  1.6]
 [ 4.9  2.4  3.3  1. ]
 [ 6.6  2.9  4.6  1.3]
 [ 5.2  2.7  3.9  1.4]
 [ 5.   2.   3.5  1. ]
 [ 5.9  3.   4.2  1.5]
 [ 6.   2.2  4.   1. ]
 [ 6.1  2.9  4.7  1.4]
 [ 5.6  2.9  3.6  1.3]
 [ 6.7  3.1  4.4  1.4]
 [ 5.6  3.   4.5  1.5]
 [ 5.8  2.7  4.1  1. ]
 [ 6.2  2.2  4.5  1.5]
 [ 5.6  2.5  3.9  1.1]
 [ 5.9  3.2  4.8  1.8]
 [ 6.1  2.8  4.   1.3]
 [ 6.3  2.5  4.9  1.5]
 [ 6.1  2.8  4.7  1.2]
 [ 6.4  2.9  4.3  1.3]
 [ 6.6  3.   4.4  1.4]
 [ 6.8  2.8  4.8  1.4]
 [ 6.7  3.   5.   1.7]
 [ 6.   2.9  4.5  1.5]
 [ 5.7  2.6  3.5  1. ]
 [ 5.5  2.4  3.8  1.1]
 [ 5.5  2.4  3.7  1. ]
 [ 5.8  2.7  3.9  1.2]
 [ 6.   2.7  5.1  1.6]
 [ 5.4  3.   4.5  1.5]
 [ 6.   3.4  4.5  1.6]
 [ 6.7  3.1  4.7  1.5]
 [ 6.3  2.3  4.4  1.3]
 [ 5.6  3.   4.1  1.3]
 [ 5.5  2.5  4.   1.3]
 [ 5.5  2.6  4.4  1.2]
 [ 6.1  3.   4.6  1.4]
 [ 5.8  2.6  4.   1.2]
 [ 5.   2.3  3.3  1. ]
 [ 5.6  2.7  4.2  1.3]
 [ 5.7  3.   4.2  1.2]
 [ 5.7  2.9  4.2  1.3]
 [ 6.2  2.9  4.3  1.3]
 [ 5.1  2.5  3.   1.1]
 [ 5.7  2.8  4.1  1.3]
```

```
 [ 6.3  3.3  6.   2.5]
 [ 5.8  2.7  5.1  1.9]
 [ 7.1  3.   5.9  2.1]
 [ 6.3  2.9  5.6  1.8]
 [ 6.5  3.   5.8  2.2]
 [ 7.6  3.   6.6  2.1]
 [ 4.9  2.5  4.5  1.7]
 [ 7.3  2.9  6.3  1.8]
 [ 6.7  2.5  5.8  1.8]
 [ 7.2  3.6  6.1  2.5]
 [ 6.5  3.2  5.1  2. ]
 [ 6.4  2.7  5.3  1.9]
 [ 6.8  3.   5.5  2.1]
 [ 5.7  2.5  5.   2. ]
 [ 5.8  2.8  5.1  2.4]
 [ 6.4  3.2  5.3  2.3]
 [ 6.5  3.   5.5  1.8]
 [ 7.7  3.8  6.7  2.2]
 [ 7.7  2.6  6.9  2.3]
 [ 6.   2.2  5.   1.5]
 [ 6.9  3.2  5.7  2.3]
 [ 5.6  2.8  4.9  2. ]
 [ 7.7  2.8  6.7  2. ]
 [ 6.3  2.7  4.9  1.8]
 [ 6.7  3.3  5.7  2.1]
 [ 7.2  3.2  6.   1.8]
 [ 6.2  2.8  4.8  1.8]
 [ 6.1  3.   4.9  1.8]
 [ 6.4  2.8  5.6  2.1]
 [ 7.2  3.   5.8  1.6]
 [ 7.4  2.8  6.1  1.9]
 [ 7.9  3.8  6.4  2. ]
 [ 6.4  2.8  5.6  2.2]
 [ 6.3  2.8  5.1  1.5]
 [ 6.1  2.6  5.6  1.4]
 [ 7.7  3.   6.1  2.3]
 [ 6.3  3.4  5.6  2.4]
 [ 6.4  3.1  5.5  1.8]
 [ 6.   3.   4.8  1.8]
 [ 6.9  3.1  5.4  2.1]
 [ 6.7  3.1  5.6  2.4]
 [ 6.9  3.1  5.1  2.3]
 [ 5.8  2.7  5.1  1.9]
 [ 6.8  3.2  5.9  2.3]
 [ 6.7  3.3  5.7  2.5]
 [ 6.7  3.   5.2  2.3]
 [ 6.3  2.5  5.   1.9]
 [ 6.5  3.   5.2  2. ]
 [ 6.2  3.4  5.4  2.3]
 [ 5.9  3.   5.1  1.8]]
```

Problem 1d What is the shape and content of the `iris` target?

In [14]:

```
print(np.shape(iris.target))
print(iris.target)
```

```
(150,)
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

Finally, as a baseline for the exercises that follow, we will now make a simple 2D plot showing the separation of the 3 classes in the iris dataset. This plot will serve as the reference for examining the quality of the clustering algorithms.

Problem 1e Make a scatter plot showing sepal length vs. sepal width for the iris data set. Color the points according to their respective classes.

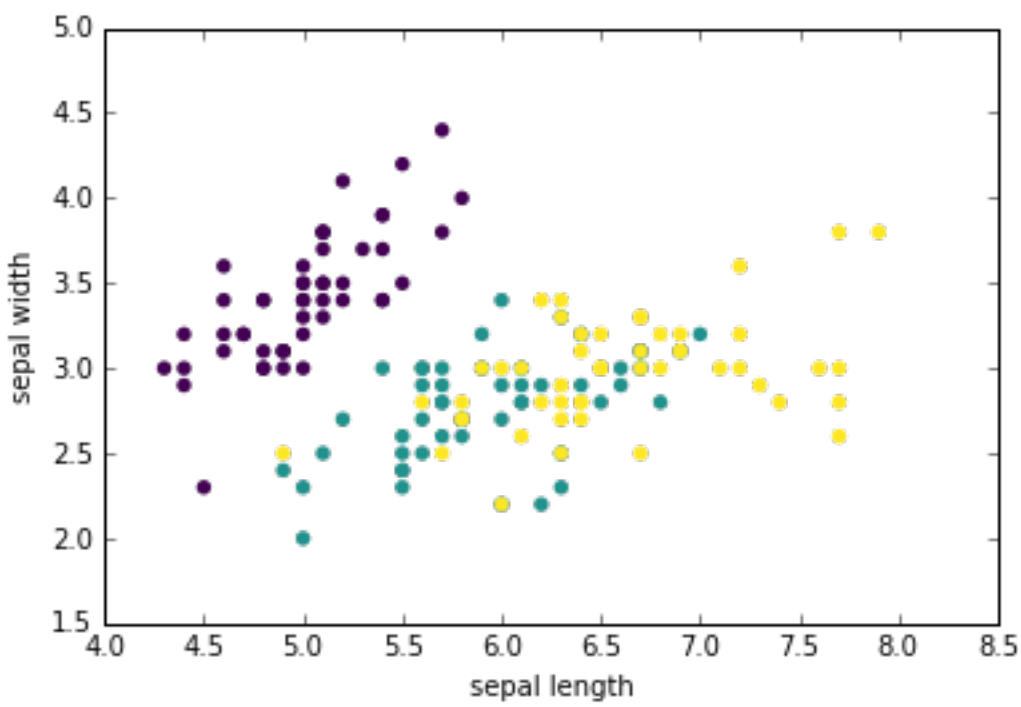
In [35]:

```
print(iris.feature_names)  # shows that sepal length is first feature and sepal wid

plt.scatter(iris.data[:,0], iris.data[:,1], c = iris.target, s = 30, edgecolor = "N
plt.xlabel('sepal length')
plt.ylabel('sepal width')
```

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

Out[35]:

```
<matplotlib.text.Text at 0x1142fd610>
```



Problem 2) Unsupervised Machine Learning

Unsupervised machine learning, sometimes referred to as clustering or data mining, aims to group or classify sources in the multidimensional feature space. The "unsupervised" comes from the fact that there are no target labels provided to the algorithm, so the machine is asked to cluster the data "on its own." The lack of labels means there is no (simple) method for validating the accuracy of the solution provided by the machine (though sometimes simple examination can show the results are **terrible**).

For this reason [*note* - this is my (AAM) opinion and there many be many others who disagree], unsupervised methods are not particularly useful for astronomy. Supposing one did find some useful clustering structure, an adversarial researcher could always claim that the current feature space does not accurately capture the physics of the system and as such the clustering result is not interesting or, worse, erroneous. The one potentially powerful exception to this broad statement is outlier detection, which can be a branch of both unsupervised and supervised learning. Finding weirdo objects is an astronomical pastime, and there are unsupervised methods that may help in that regard in the LSST era.

To begin today we will examine one of the most famous, and simple, clustering algorithms: k-means (https://en.wikipedia.org/wiki/K-means_clustering). k-means clustering looks to identify k convex clusters, where k is a user defined number. And here-in lies the rub: if we truly knew the number of clusters in advance, we likely wouldn't need to perform any clustering in the first place. This is the major downside to k-means. Operationally, pseudocode for the algorithm can be summarized as the following:

```
initiate search by identifying k points (i.e. the cluster centers)
loop
    assign each point in the data set to the closest cluster center
    calculate new cluster centers based on mean position of all points withi
n each cluster
    if diff(new center - old center) < threshold:
        stop (i.e. clusters are defined)
```

The threshold is defined by the user, though in some cases the total number of iterations is also An advantage of k-means is that the solution will always converge, though the solution may only be a local minimum. Disadvantages include the assumption of convexity, i.e. difficult to capture complex geometry, and the curse of dimensionality (though you can combat that with dimensionality reduction after yesterday).

In `scikit-learn` the `KMeans` (http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html#sklearn.cluster.KMeans) is implemented as part of the `sklearn.cluster` (http://scikit-learn.org/stable/modules/classes.html#module-sklearn.cluster) module.

Problem 2a Fit two different k-means models to the iris data, one with 2 clusters and one with 3 clusters. Plot the resulting clusters in the sepal length-sepal width plane (same plot as above). How do the results compare to the true classifications?

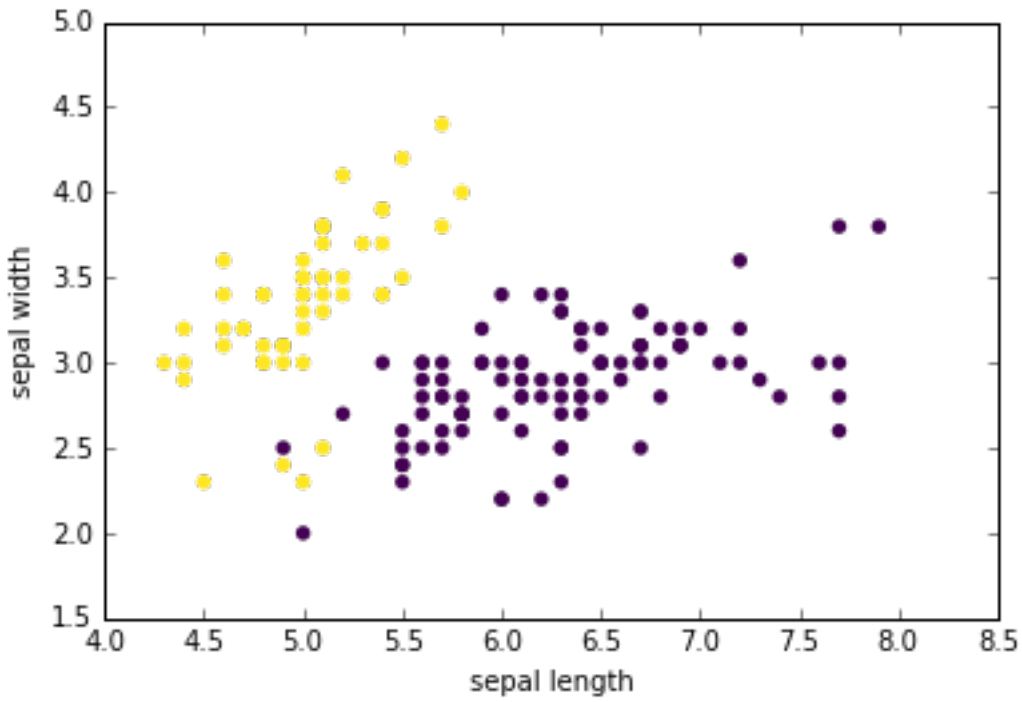
In [77]:

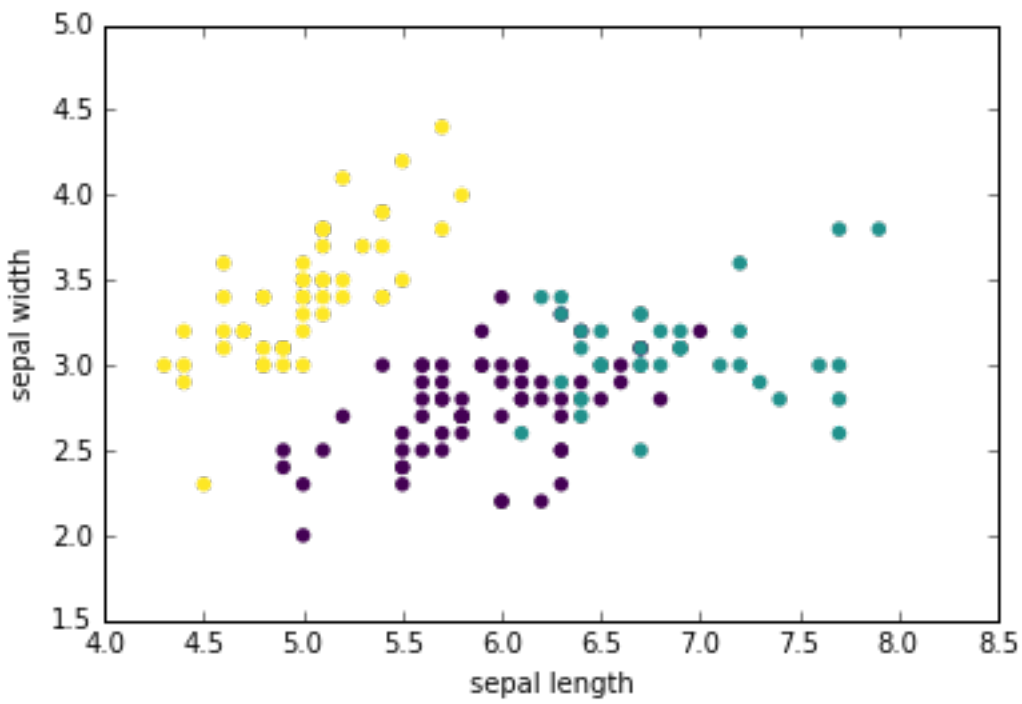
```
from sklearn.cluster import KMeans

Kcluster = KMeans(n_clusters = 2)
Kcluster.fit(iris.data)
```

```
plt.figure()
plt.scatter(iris.data[:,0], iris.data[:,1], c = Kcluster.labels_, s = 30, edgecolor
plt.xlabel('sepal length')
plt.ylabel('sepal width')

Kcluster = KMeans(n_clusters = 3)
Kcluster.fit(iris.data)

plt.figure()
plt.scatter(iris.data[:,0], iris.data[:,1], c = Kcluster.labels_, s = 30, edgecolor
plt.xlabel('sepal length')
plt.ylabel('sepal width')
```

Out[77]:

```
<matplotlib.text.Text at 0x119af6e90>
```





With 3 clusters the algorithm does a good job of separating the three classes. However, without the a priori knowledge that there are 3 different types of iris, the 2 cluster solution would appear to be superior.

Problem 2b How do the results change if the 3 cluster model is called with `n_init = 1` and `init = 'random'` options? Use `rs` for the random state [this allows me to cheat in service of making a point].

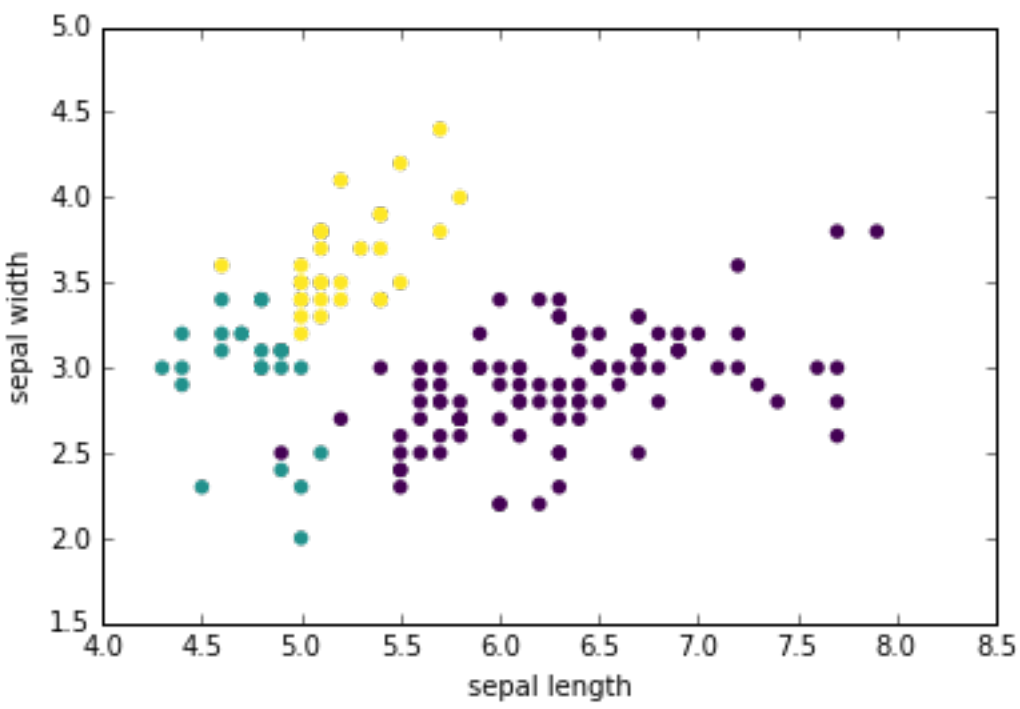
*Note - the respective defaults for these two parameters are 10 and `k-means++`, respectively. Read the docs to see why these choices are, likely, better than those in 2b.

In [70]:

```
rs = 14
Kcluster1 = KMeans(n_clusters = 3, n_init = 1, init = 'random', random_state = rs)
Kcluster1.fit(iris.data)

plt.figure()
plt.scatter(iris.data[:,0], iris.data[:,1], c = Kcluster1.labels_, s = 30, edgecolo
plt.xlabel('sepal length')
plt.ylabel('sepal width')
```

Out[70]:

```
<matplotlib.text.Text at 0x119318390>
```



A random aside that is not particularly relevant here

k-means evaluates the Euclidean distance between individual sources and cluster centers, thus, the magnitude of the individual features has a strong effect on the final clustering outcome.

Problem 2c Calculate the mean, standard deviation, min, and max of each feature in the iris data set. Based on these summaries, which feature is most important for clustering?

In [74]:

```
print("feature\t\t\tmean\tstd\tmin\tmax")
for featnum, feat in enumerate(iris.feature_names):
    print("{:s}\t{:.2f}\t{:.2f}\t{:.2f}\t{:.2f}".format(feat, np.mean(iris.data[:,fe
                                                        np.std(iris.data[:,featnum]
                                                        np.max(iris.data[:,featnum]
```

```
feature                 mean    std     min     max
sepal length (cm)       5.84    0.83    4.30    7.90
sepal width (cm)        3.05    0.43    2.00    4.40
petal length (cm)       3.76    1.76    1.00    6.90
petal width (cm)        1.20    0.76    0.10    2.50
```

Petal length has the largest range and standard deviation, thus, it will have the most "weight" when determining the k clusters.

The truth is that the iris data set is fairly small and straightfoward. Nevertheless, we will now examine the clustering results after re-scaling the features. [Some algorithms, *cough* Support Vector Machines *cough*, are notoriously sensitive to the feature scaling, so it is important to know about this step.] Imagine you are classifying stellar light curves: the data set will include contact binaries with periods of ~ 0.1 d and Mira variables with periods of $\gg 100$ d. Without re-scaling, this feature that covers 4 orders of magnitude may dominate all others in the final model projections.

The two most common forms of re-scaling are to rescale to a guassian with mean $= 0$ and variance $= 1$, or to rescale the min and max of the feature to $[0, 1]$. The best normalization is problem dependent. The `sklearn.preprocessing` (http://scikit-learn.org/stable/modules/classes.html#module-sklearn.preprocessing) module makes it easy to re-scale the feature set. **It is essential that the same scaling used for the training set be used for all other data run through the model.** The testing, validation, and field observations cannot be re-scaled independently. This would result in meaningless final classifications/predictions.

Problem 2d Re-scale the features to normal distributions, and perform k-means clustering on the iris data. How do the results compare to those obtained earlier?

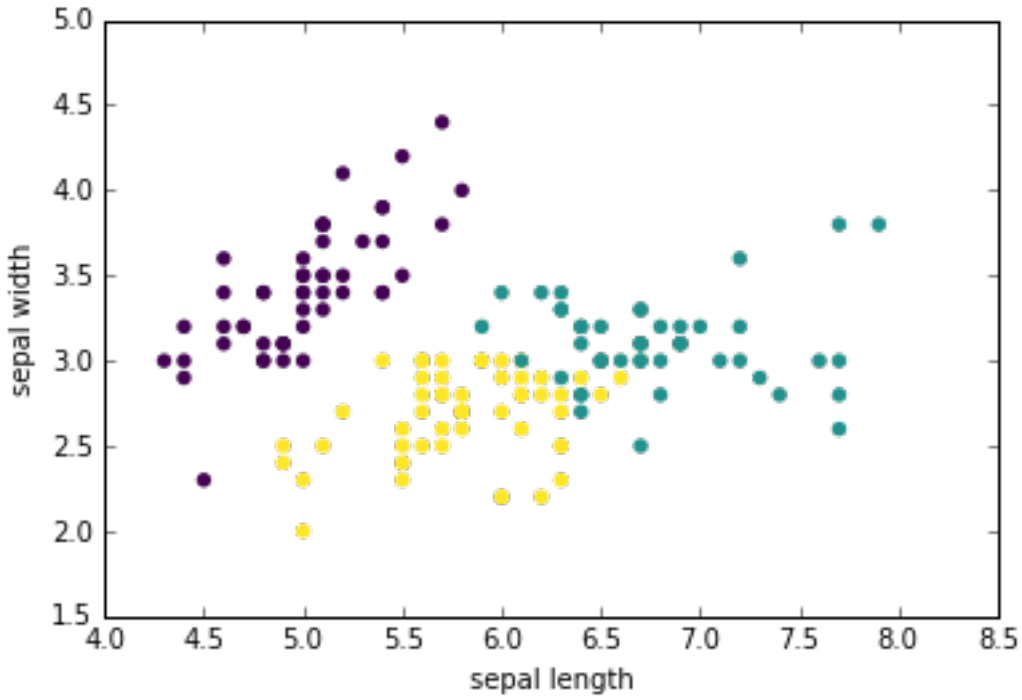
Hint - you may find `'StandardScaler()'` (http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardS within the `sklearn.preprocessing` module useful.

In [76]:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler().fit(iris.data)

Kcluster = KMeans(n_clusters = 3)
Kcluster.fit(scaler.transform(iris.data))

plt.figure()
plt.scatter(iris.data[:,0], iris.data[:,1], c = Kcluster.labels_, s = 30, edgecolor
plt.xlabel('sepal length')
plt.ylabel('sepal width')
```

Out[76]:

```
<matplotlib.text.Text at 0x1195cca10>
```



These results are almost identical to those obtained without scaling. This is due to the simplicity of the iris data set.

How do I test the accuracy of my clusters?

Essentially - you don't. There are some methods that are available, but they essentially compare clusters to labeled samples, and if the samples are labeled it is likely that supervised learning is more useful anyway. If you are curious, `scikit-learn` does provide some built-in functions for analyzing clustering (http://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation), but again, it is difficult to evaluate the validity of any newly discovered clusters.

What if I don't know how many clusters are present in the data?

An excellent question, as you will almost never know this a priori. Many algorithms, like k-means, do require the number of clusters to be specified, but some other methods do not. As an example `DBSCAN` (https://en.wikipedia.org/wiki/DBSCAN). In brief, `DBSCAN` requires two parameters: `minPts`, the minimum

number of points necessary for a cluster, and ϵ, a distance measure. Clusters are grown by identifying *core points*, objects that have at least `minPts` located within a distance ϵ. *Reachable points* are those within a distance ϵ of at least one *core point* but less than `minPts` *core points*. Identically, these points define the outskirts of the clusters. Finally, there are also *outliers* which are points that are $> \epsilon$ away from any *core points*. Thus, `DBSCAN` naturally identifies clusters, does not assume clusters are convex, and even provides a notion of outliers. The downsides to the algorithm are that the results are highly dependent on the two tuning parameters, and that clusters of highly different densities can be difficult to recover (because ϵ and `minPts` is specified for all clusters.

In `scitkit-learn` the `DBSCAN` (http://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html#sklearn.cluster.DBSCAN) algorithm is part of the `sklearn.cluster` module. ϵ and `minPts` are set by `eps` and `min_samples`, respectively.

Problem 2e Cluster the iris data using `DBSCAN`. Play around with the tuning parameters to see how they affect the final clustering results. How does the use of `DBSCAN` compare to k-means? Can you obtain 3 clusters with `DBSCAN`? If not, given the knowledge that the iris dataset has 3 classes - does this invalidate `DBSCAN` as a viable algorithm?

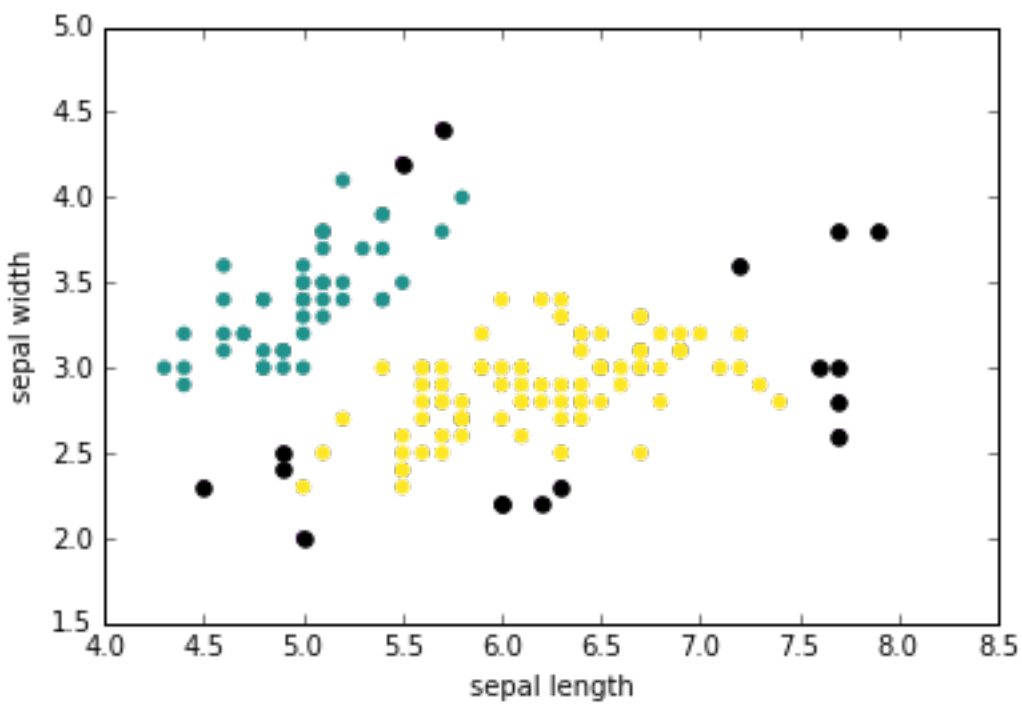
Note - DBSCAN labels outliers as -1, and thus, `plt.scatter()`, will plot all these points as the same color.

In [109]:

```
from sklearn.cluster import DBSCAN

dbs = DBSCAN(eps = 0.7, min_samples = 7)
dbs.fit(scaler.transform(iris.data)) # best to use re-scaled data since eps is in al

dbs_outliers = dbs.labels_ == -1

plt.figure()
plt.scatter(iris.data[:,0], iris.data[:,1], c = dbs.labels_, s = 30, edgecolor = "No
plt.scatter(iris.data[:,0][dbs_outliers], iris.data[:,1][dbs_outliers], s = 30, c =


plt.xlabel('sepal length')
plt.ylabel('sepal width')
```

Out[109]:

```
<matplotlib.text.Text at 0x11bed9bd0>
```



I was unable to obtain 3 clusters with `DBSCAN`. While these results are, on the surface, worse than what we got with k-means, my suspicion is that the 4 features do not adequately separate the 3 classes. [See - a nayseyer can always make that argument.] This is not a problem for `DBSCAN` as an algorithm, but rather, evidence that no single algorithm works well in all cases.

Challenge Problem) Cluster SDSS Galaxy Data

The following query will select 10k likely galaxies from the SDSS database and return the results of that query into an `astropy.Table` (http://docs.astropy.org/en/stable/table/) object. (For now, if you are not familiar with the SDSS DB schema, don't worry about this query, just know that it returns a bunch of photometric features.)

In [112]:

```
from astroquery.sdss import SDSS  # enables direct queries to the SDSS database

GALquery = """SELECT TOP 10000
             p.dered_u - p.dered_g as ug, p.dered_g - p.dered_r as gr,
             p.dered_g - p.dered_i as gi, p.dered_g - p.dered_z as gz,
             p.petroRad_i, p.petroR50_i, p.deVAB_i
             FROM PhotoObjAll AS p JOIN specObjAll s ON s.bestobjid = p.objid
             WHERE p.mode = 1 AND s.sciencePrimary = 1 AND p.clean = 1 AND p.type =
               """
SDSSgals = SDSS.query_sql(GALquery)
SDSSgals
```

Out[112]:

<Table length=10000>

ug	gr	gi	gz	petroRad_i	petroR50_i	deVAB_i
float64	**float64**	**float64**	**float64**	**float64**	**float64**	**float64**
1.93272	1.175575	1.667498	2.017296	3.373127	1.529253	0.7076685
2.781702	1.809624	2.473322	2.801765	3.549374	1.65708	0.7291917
0.8051357	1.788557	2.753258	3.218267	2.406485	1.144235	0.3733464
2.26919	1.796471	2.389879	2.839649	2.690604	1.187831	0.9998215
1.172142	0.9655094	1.409794	1.694769	6.612225	3.220872	0.8142815
2.68491	1.919027	2.509253	2.903473	6.601803	1.310293	0.6660755
0.1612816	1.607111	2.484976	2.922279	3.75017	1.572535	0.5757828
3.981474	1.596598	2.146681	2.513348	4.562647	1.891203	0.8520364
0.6872368	1.553604	2.516878	2.940353	2.708638	1.264196	0.7275088
0.5068417	0.5828114	0.6043072	0.4324818	1.60555	0.8807057	0.423754
...	...	...	...	...	...	...
1.688602	1.669693	2.879099	3.548115	2.259612	0.9042092	0.740374
1.022163	1.881332	2.495893	2.920853	1.982942	0.9034599	0.8675485
2.337614	1.617163	2.165649	2.525099	3.22549	1.320168	0.8328338
3.07198	1.461004	2.62735	3.190205	3.130382	1.120104	0.862262
1.164221	1.81299	2.812445	3.528988	1.927245	0.9313561	0.5349207
2.407913	1.657169	2.275141	2.685274	3.498855	1.504274	0.7847368
1.410875	1.385931	2.236004	2.742559	2.7224	1.117109	0.6138035
2.761852	1.355759	2.058468	2.471798	1.372076	0.7143538	0.09999066
1.816816	2.252766	3.272234	3.671343	1.87972	0.9189188	0.8029428

0.2271004	1.715874	2.925756	3.363508	7.705968	1.184086	0.7054715

I have used my own domain knowledge to specificly choose features that may be useful when clustering galaxies. If you know a bit about SDSS and can think of other features that may be useful feel free to add them to the query.

One nice feature of `astropy` tables is that they can readily be turned into `pandas DataFrames`, which can in turn easily be turned into a `sklearn X` array with `NumPy`. For example:

```
X = np.array(SDSSgals.to_pandas())
```

And you are ready to go.

Challenge Problem Using the SDSS dataset above, identify interesting clusters within the data [this is intentionally very open ended, if you uncover anything especially exciting you'll have a chance to share it with the group]. Feel free to use the algorithms discussed above, or any other packages available via `sklearn`. Can you make sense of the clusters in the context of galaxy evolution?

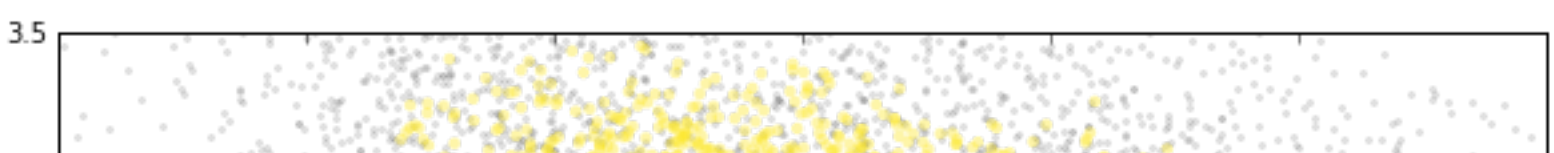
Hint - don't fret if you know nothing about galaxy evolution (neither do I!). Just take a critical look at the clusters that are identified

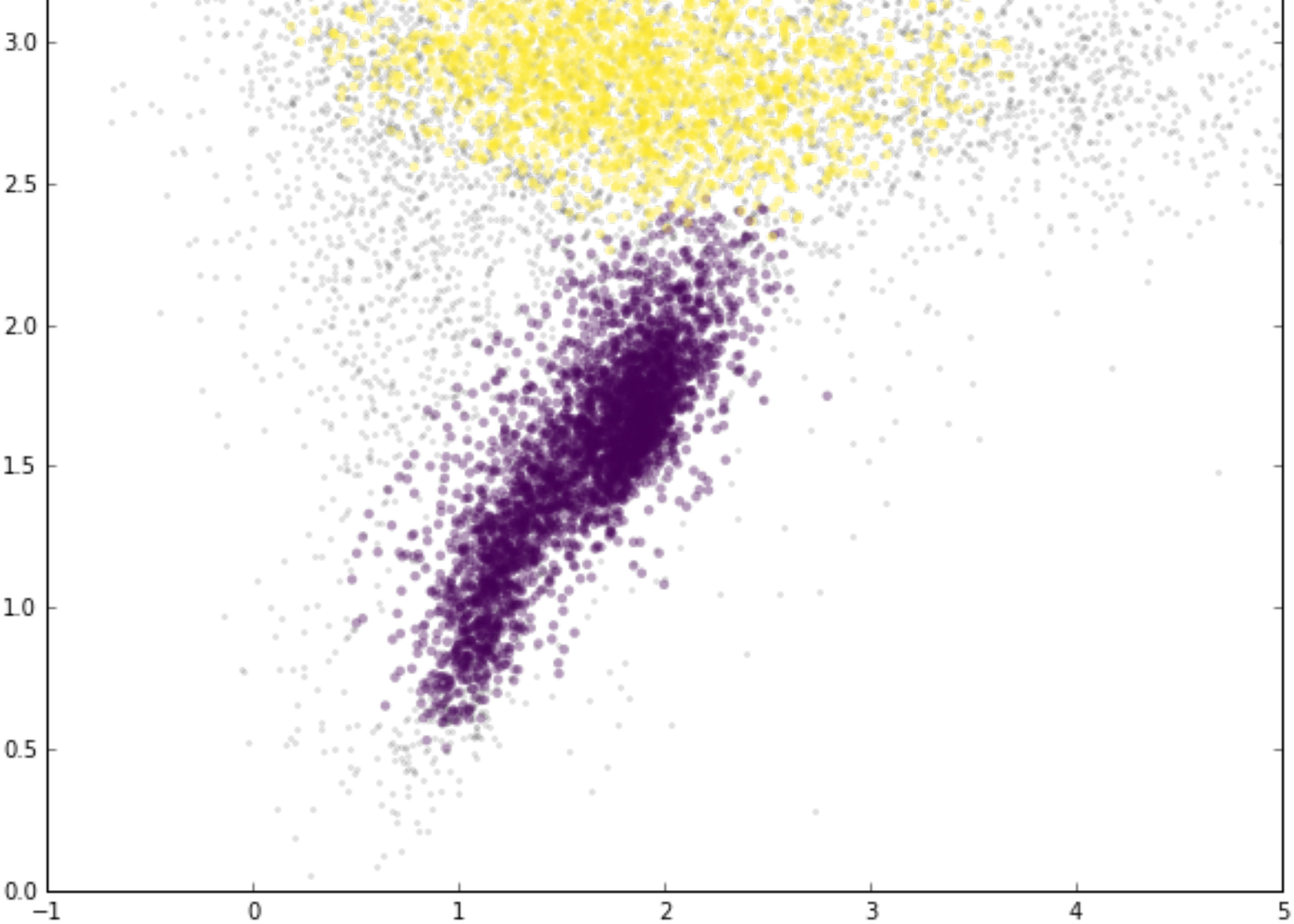
In [155]:

```
Xgal = np.array(SDSSgals.to_pandas())

galScaler = StandardScaler().fit(Xgal)

dbs = DBSCAN(eps = 0.5, min_samples=80)

dbs.fit(galScaler.transform(Xgal))

cluster_members = dbs.labels_ != -1
outliers = dbs.labels_ == -1

plt.figure(figsize = (10,8))
plt.scatter(Xgal[:,0][outliers], Xgal[:,3][outliers],
            c = "k",
            s = 4, alpha = 0.1)
plt.scatter(Xgal[:,0][cluster_members], Xgal[:,3][cluster_members],
            c = dbs.labels_[cluster_members],
            alpha = 0.4, edgecolor = "None", cmap = "viridis")

plt.xlim(-1,5)
plt.ylim(-0,3.5)
```

Out[155]:

```
(0, 3.5)
```





Note - the above solution seems to separate out elliptical galaxies from blue star forming galaxies, however, the results are highly, highly dependent upon the tuning parameters.

In [ ]: