*You are currently looking at **version 1.0** of this notebook. To download notebooks and datafiles, as well as get help on Jupyter notebooks in the Coursera platform, visit the Jupyter Notebook FAQ (https://www.coursera.org/learn/python-machine-learning/resources/bANLa) course resource.*

01-01 Key Concepts in Machine Learning

Key Types of Machine Learning Problems

Supervised Learning

Supervised Machine learning aims to predict **target values** from labelled data. Within this, there are 2 different types

- Classification - where target values are discrete classes
  - Whether a credit card transaction is fraudulent or legitimate
  - Whether a patient with a breast cancer tumour is malicious or benign

- Regression - where target values are continuous. This involves some training examples to find a model for this program, and then it can use this model to predict the y-value for some x it has not seen before.
  - Determine the housing price of some house given the land area.
  - Determine the size of a tumour with respect to stage when the tumour is discovered.

Supervised learning needs to have training examples. Training examples come from **explicit or implicit** sources.

- Explicit Sources
  - Training labels are typically provided by human judges. (think of you doing ReCaptcha).
  - Use of crowdsourcing platforms (Amazon's Mechanical Turk, Crowd Flower) have been a significant source of explicitly provided labels as well.
- Implicit Sources
  - If a search engine detects a user clicking on a result link and then sees no more activity for another minute or two before the user comes back to the search engine, the system might use that activity as a kind of an implicit label for that page. **In other words, if the user took some time to visit that page, it is more likely that that page was relevant to their query.**

Non-Supervised Learning

Unsupervised Machine learning - to find *structure* in **unlabelled data**.

Within this there are 2 kinds of problems:

- Clustering : Find groups of similar instances in the data
  - Structure here represents like finding clusters within the input data - where you can produce a useful summary of the input data or maybe visualising the structure.
  - If you run an ecommerce site and you have millions of customers, you wander if you can group the customers into different types. For example,
    - There might be *power users* who use more advance features of the site.
    - There might be *quick browser users* who only buy things on a cheap discount
    - There might be a *researchers* who spend a lot of time comparing items before purchasing.
- Outlier Detection: Finding unusual patterns in data.
  - For instance, sometimes someone may flag abnormal access to a server
  - Behaviour that is very different from the usual behaviour to your site.

    - So we have an unsupervised learning approach because we don't want to assume that a future attack is similar to an old kind of attack - but to detect any strange behaviour on our website (for instance).

Basic Machine Learning Workflow

Representation

- **Feature Representation** -Figure out how to represent the learning problem in terms of something the computer can understand. Take a data or a description of your object (your variables) that you're interested in recognising in a way that you can use input to an algorithm.
- **Type of Classifier to Use**
    - You also need to decide what type of learning algorithm to apply to this data.

For example, think of how to represent:

- An Image - Array of colored pixels? Metadata associated with the image?
- Credit card Transaction - time, place and amount of transaction etc ### Evaluation
- **Metric**
    - This provides some type of quality or accuracy score for the predictions coming out of the machine learning algorithm (like a classifiers). This allows you to compare the effectiveness of different classifiers.

Optimisation

We have to search for settings /parameters that give the best classifer for this evaluation criterion.

```
- Gradient Descent
- PCA
```

Note that machine learning is an **iterative process**

- where we always have to make some initial guess about what some good features are of the problem, and the classifier that might be appropriate.
- We then train the system using our training data.
- Produce an evaluation - see how well the classifier works and then based on what worked and what didn't work (i.e. which examples got classified correctly or incorrectly we can do a failure analysis to see where the system is still making mistakes)
- Refine the implementation and repeat.

Feature Representation

Your object is some *entity* represented by a collection of different properties. You can think of the input data containing this feature representation.

Applied Machine Learning, Module 1: A simple classification task

Import required modules and load data file

In [1]:

```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split

fruits = pd.read_table('readonly/fruit_data_with_colors.txt')
```

In [2]:

```
fruits.head()
```

Out[2]:

	fruit_label	fruit_name	fruit_subtype	mass	width	height	color_score
0	1	apple	granny_smith	192	8.4	7.3	0.55
1	1	apple	granny_smith	180	8.0	6.8	0.59
2	1	apple	granny_smith	176	7.4	7.2	0.60
3	2	mandarin	mandarin	86	6.2	4.7	0.80
4	2	mandarin	mandarin	84	6.0	4.6	0.79

In [3]:

```
# create a mapping from fruit label value to fruit name to make results easier t
o interpret
lookup_fruit_name = dict(zip(fruits.fruit_label.unique(), fruits.fruit_name.uniq
ue()))
lookup_fruit_name
```

Out[3]:

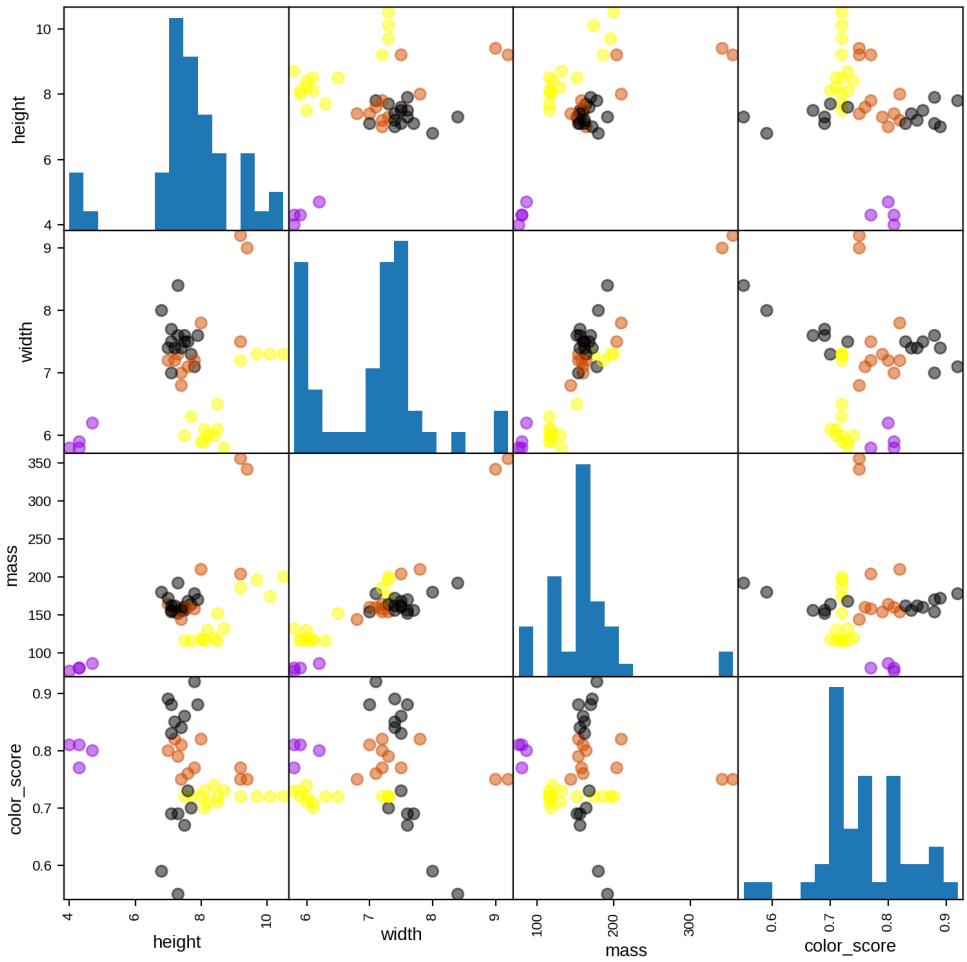
```
{1: 'apple', 2: 'mandarin', 3: 'orange', 4: 'lemon'}
```

The file contains the mass, height, and width of a selection of oranges, lemons and apples. The heights were measured along the core of the fruit. The widths were the widest width perpendicular to the height.

Examining the data

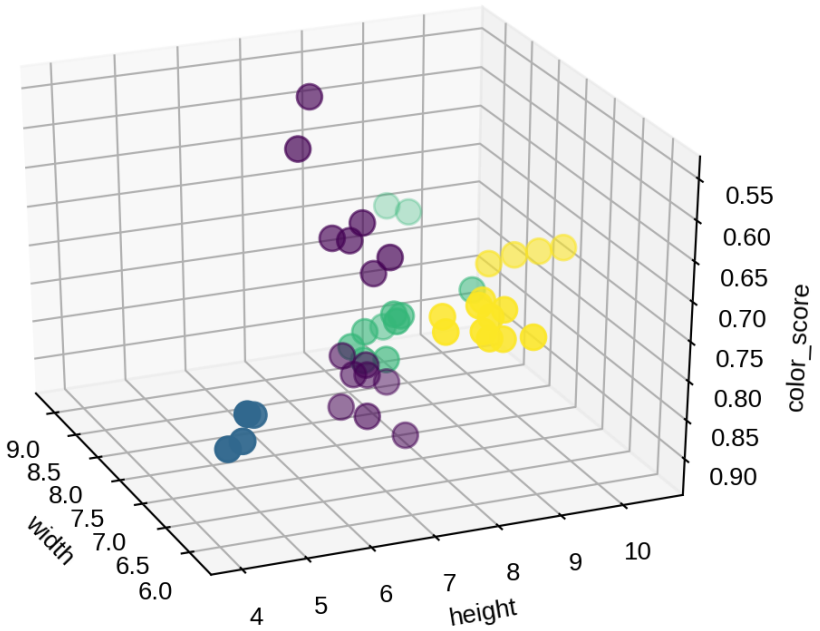
In [4]:

```
# plotting a scatter matrix
from matplotlib import cm

X = fruits[['height', 'width', 'mass', 'color_score']]
y = fruits['fruit_label']
# this function from scikit learn helps you randomly shuffle the dataset
# and splits off a certain percentage of the input samples for use as a training
set.
# and puts the remaining samples into a different variable for use as a test se
t.
# So in this example, we're using a 75-25% split of training vs test data.
# This is a good standard of training vs test data.
# X is typically a 2D array of n rows and k columns, y is a n dimensional vecto
r.
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
# ^ tuple unpacking used here - note the order!

# The above random_state parameter provides a seed value for the functions INTER
NAL random number
# generator. If we choose different values for this seed value, that will result
in different
# randomised splits for training and test.

cmap = cm.get_cmap('gnuplot')
scatter = pd.scatter_matrix(X_train, c= y_train, marker = 'o', s=40, hist_kwds={
'bins':15}, figsize=(9,9), cmap=cmap)
```


height
width
mass
color_score
10
8
6
4
9
8
7
6
350
300
250
200
150
100
0.9
0.8
0.7
0.6
4
6
8
10
6
7
8
9
100
200
300
0.6
0.7
0.8
0.9

In [5]:

```
# plotting a 3D scatter plot
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')
ax.scatter(X_train['width'], X_train['height'], X_train['color_score'], c = y_tr
ain, marker = 'o', s=100)
ax.set_xlabel('width')
ax.set_ylabel('height')
ax.set_zlabel('color_score')
plt.show()
```



Create train-test split

In [7]:

```
# For this example, we use the mass, width, and height features of each fruit in
stance
X = fruits[['mass', 'width', 'height']]
y = fruits['fruit_label']

# default is 75% / 25% train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

01-03 k- Nearest Neighbours Classification

We are going to use this dataset to train a classifier that will automatically identify any features of fruit that might come our way. This is based on the features available to the classifier such as the object's colour, size and mass. To do this, we'll use a popular and easy to understand type of machine learning algorithm, such as k-nearest neighbours (or k-NN). The K-Nearest Neighbours algorithm can be used for classification and regression. Though right now, we will focus on classification.

k-NN Classifiers

k-NN classifiers are an **instance based / memory based supervised learning**. What this means is that instance based learning methods work by memorising the labelled examples they see in the training set, and they use those **memorised examples** to classify new objects later. The k in k-NN refers to the number of nearest neighbours the classifier will retrieve and use in order ot make its prediction.

Create k-NN classifier object

In [8]:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors = 5)
```

Train the classifier (fit the estimator) using the training data

This k-NN classifier that we use in this case is a **derived class** from the `Estimator` class in Scikit-learn. So all estimators have a fit method that takes the training data, and then changes the state of the classifier or estimator object to essentially enable prediction once the training is finished.

It updates the `knn` variable that in the case of k-nearest neighbours it will memorise the training set examples in some kind of internal storage for future use.

That's really all there is to training the k-NN classifier, and the first thing we can do with this classifier is to see how well it will do on some new, previously unseen instances of the dataset.

In [9]:

```
knn.fit(X_train, y_train)
```

Out[9]:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkows
ki',
           metric_params=None, n_jobs=1, n_neighbors=5, p=2,
           weights='uniform')
```

Estimate the accuracy of the classifier on future data, using the test data

The first thing we can do with this classifier is to see how well it will do on some new, previously unseen instances of the dataset.

To do this, we can apply the classifier to all variables in the test set that we put aside, since these test instances were not explicitly included in the classifier's training. One simple way to assess if the classifer is likely to be good at predicting the label of future, previously unseen data instances, is to compute the classifier's accuracy on the test set data items.

Using `knn.score`

To do this, we use the score method for the classifier object. This will take the test set points as input and compute the accuracy.

The accuracy is defined as the fraction of test set items, whose true labels was correctly predicted by the classifier.

In [10]:

```
knn.score(X_test, y_test)
```

Out[10]:

```
0.53333333333333333
```

Using `knn.predict()`: Use the trained k-NN classifier model to classify new, previously unseen objects

This is a hypothetical piece of fruit that wasn't in our train set or test set.

In [11]:

```
# first example: a small fruit with mass 20g, width 4.3 cm, height 5.5 cm
fruit_prediction = knn.predict([[20, 4.3, 5.5]])
lookup_fruit_name[fruit_prediction[0]]
```

Out[11]:

```
'mandarin'
```

In [12]:

```
# second example: a larger, elongated fruit with mass 100g, width 6.3 cm, height
8.5 cm
fruit_prediction = knn.predict([[100, 6.3, 8.5]])
lookup_fruit_name[fruit_prediction[0]]
```

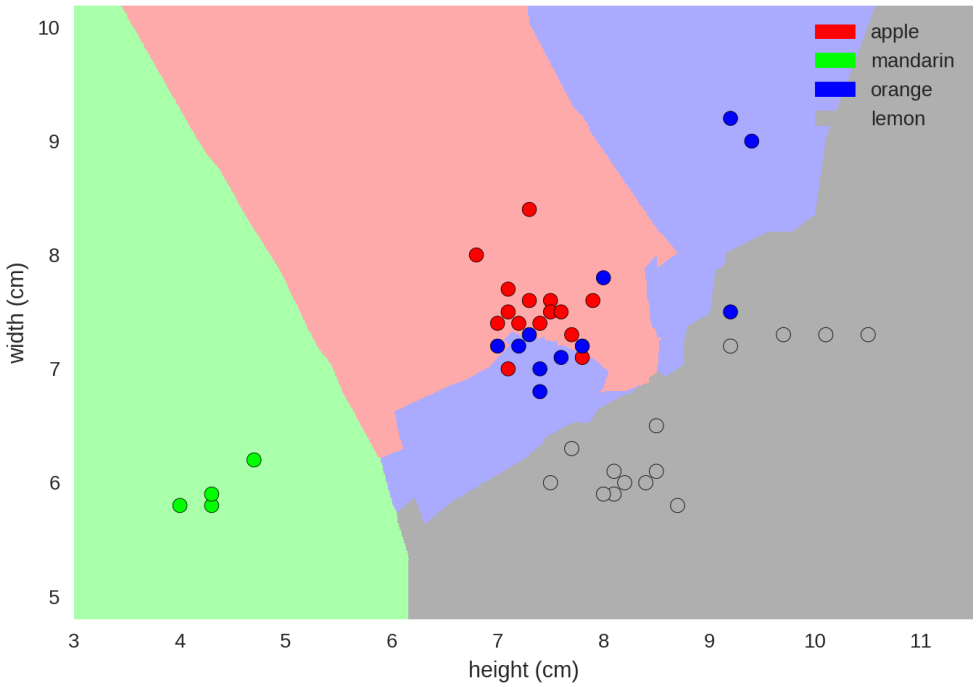
Out[12]:

```
'lemon'
```

Plot the decision boundaries of the k-NN classifier

In [13]:

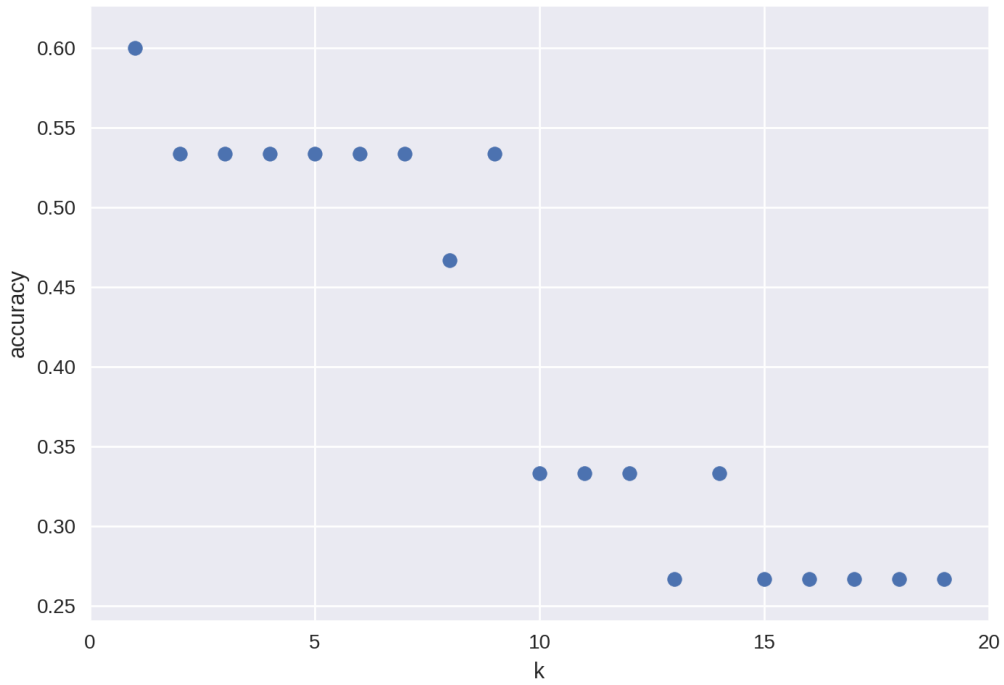
```
from adspy_shared_utilities import plot_fruit_knn

plot_fruit_knn(X_train, y_train, 5, 'uniform')   # we choose 5 nearest neighbors
```



How sensitive is k-NN classification accuracy to the choice of the 'k' parameter?

In [14]:

```
k_range = range(1,20)
scores = []

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(X_train, y_train)
    scores.append(knn.score(X_test, y_test))

plt.figure()
plt.xlabel('k')
plt.ylabel('accuracy')
plt.scatter(k_range, scores)
plt.xticks([0,5,10,15,20]);
```



How sensitive is k-NN classification accuracy to the train/test split proportion?

In [15]:

```
t = [0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2]

knn = KNeighborsClassifier(n_neighbors = 5)

plt.figure()

for s in t:

    scores = []
    for i in range(1,1000):
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 1-
s)
        knn.fit(X_train, y_train)
        scores.append(knn.score(X_test, y_test))
    plt.plot(s, np.mean(scores), 'bo')

plt.xlabel('Training set proportion (%)')
plt.ylabel('accuracy');
```