Stats and Monte-Carlo

Statistics Libraries

- Due to their prevalence in the empirical/experimental sciences interpreted languages often come well equipped with statistics libraries for use in data-analysis and hypothesis testing
- Python is no exception and the main library is `scipy.stats` which contains an extensive range of function for computing data statistics such as,
  - chi squared
  - students t test
  - moments and averages
  - histograms
  - common statistical distributions
- primarily this is here and useful, so look through it and see what it has

Principal Component Analysis

- Principal Component Analysis (PCA) is a technique in data analytics used when looking at multidimensional data. The basis that a dataset is stored in (normally derived from how it is measured) is rarely the most useful basis for analysis, and PCA is a method of finding this.
- Real data is often highly correlated and this correlation rarely contains useful information. Preforming PCA on a dataset finds a set of uncorrelated (or minimally correlated) basis vectors which can be used to represent the data. This is formulated as an eigenvalue problem on the covariance matrix of the data.
- As an example multi-channel satellite images contain data for different wavelength bands at each pixel typically,
  - Blue
  - Green
  - Red
  - near IR
  - mid IR
- The different channels are highly correlated and contain much of the same information. In this instance the main contribution to all of the above channels will be the absolute intensity which typically corresponds to surface topology. Doing PCA on this will (normally) produce the following channels,
  - Intesity: Corresponds to topography
  - High near IR and Green, low Red: Vegetation
  - Red and IR: Soils
  - Blue: Iron bearing minerals.

- The basis resulting from preforming PCA on datasets is referred to as the principal components labeled from 1 to the number of dimensions with PC-1 being the dimension with the greatest variability (i.e. it's the principal eigenvector of the covariance matrix)
- In addition it is often found for very large dimensional datasets that after PCA some of the resulting principal components have very little variation in them. When compressing datasets these dimensions can often be dropped as there is not much information in them.
- The algorithm for preforming PCA is as follows,
    1. Start with data stored as N samples of dimension M the data can be written as X_{ij} where the first index denotes the sample and the second the 'observable'
    2. Zero-mean the data i.e. subtract the mean of each 'observable' from each variable,

$$Y_{ij} = X_{ij} - \bar{X}_j \tag{1}$$

1. Compute the covariance matrix of the data.This is given as,

$$\mathrm{Cov}_{jk} = \frac{1}{N+1} \sum_{1}^{N} Y_{ij} Y_{ik} \tag{2}$$

1. Solve an for the eigenvalues/vectors for Cov_{jk}
2. Order the eigenvectors according to the eigenvalues that corresponding to the largest eigenvalue is the principal component 1 (PC-1). The other Principle components are arranged in order of decreasing eigenvalue.
3. Transform the data into the new basis (i.e. take dot products).

FFT and Correlation

- The time taken for an algorithm to run varies with the number of datapoints. The more datapoints in the algorithm the longer it will take to run.
- Almost no algorithms have a completion time which scales as O(N) with most scaling with $O(N^2)$. An example of a slow algorithm is matrix inversion which is typically $O(N^3)$, which is why you should never invert matrices (numerically).
- Fast Fourier Transforms is an algorithm to preform Fourier transforms on a dataset. These are very good algorithm as it scales as O(N log N). Many algorithms make use of fast Fourier transforms for this reason.
- python has two implementations of fft one in `numpy` and one in `scipy`. As a break from form these two are quite different and mostly incompatible, of the two (and again unusually) `numpy.fft` probably better as it has more functionality and has a slightly more sensible representation of the Fourier transformed arrays compared to `scipy.fftpack`. `scipy.fftpack` is supposedly faster though (although it is unlikely you'll notice the difference).

- In general algorithms that preform arithmetic operations involving multiple members of an array (as appose to preforming the operations elementwise) are generally best done with fft. For instance Game-of-Life can be more efficiently programmed using convolutions with a window function.
- Examples of algorithms which can be computed with fft,

```
import numpy as np

def derivative(x):
  k = np.fft.fftfreq(x.size)
  return np.fft.ifft(2.0j*np.pi*k*np.fft.fft(x))
```

- computes the derivative of x

```
def correlate(ar1,ar2):
  far1 = np.fft.fftn(ar1)
  far2 = np.fft.fftn(ar2)
  fconvolve = np.dot(far1,far2.conjugate().T)
  return np.fft.ifftn(fconvolve)
```

- correlation of two (equal size) datasets

```
def resample(x,N):
  fx = np.fft.fft(x)
  return np.fft.irfft(x,n=N)
```

- resampling of a dataset (appears to be broken)

```
def apply_kernal(ar1,kernal):
  s = ar1.shape
  far1 = np.fft.fftn(ar1,s=s)
  fkernal = np.fft.fftn(kernal,s=s)
  fconvolve = far1*far2
  return np.fft.ifftn(fconvolve)
```

- apply an image kernal to an array, which can be used to smooth an image:

```
def smooth(ar1):
  kernal = np.array([[1,2,1],
                     [2,4,2],
                     [1,2,1]])/16.0
  return apply_kernal(ar1,kernal)
```

Monte-Carlo Algorithms

- Monte-Carlo Algorithms are used to model stochastic processes, or to study large parameter spaces. A Monte-Carlo algorithm makes a number of 'draws' and samples the parameter space according to a probability distribution. The advantage of a Monte-Carlo algorithm over say a grid

based method is a Monte-Carlo approach will concentrate it's sampling in regions of the parameter space with the highest probability

- While it is normal to think of probability distributions in terms of probability density functions (for instance normal/Gaussian distributions), this is less useful algorithmicaly. Instead it is the quantile function that is useful. A Quantile function is the inverse of a cumulative distribution function. Thus a sample from the parameter space can be obtained by passing a random number in the range [0,1] to the quantile function.
- The Monte-car lo algorithm can be summarised as follows:
  1. Generate a random number (or array of numbers) in the range [0,1]
  2. Pass the number (array) to the quantile function to obtain a point in the parameter space
  3. Repeat for the required number of draws.
  4. That's it really.
- Once the sample set has been obtained the density of samples can be calculated to enable statistics to be calculated on the parameter space.

Markov-Chains

- Markov-Chains are used when a stochastic process depends on the previous state of the system. This is different from the standard Monte-Carlo approach which assumes that each draw is independent.
- Markov-Chains can be used to compute random walks.
- Markov-Chains obey the following recurrence relation:

$$(X_{t+1}|X_u, u \leq t) = (X_{t+1}|X_t) \tag{3}$$

- A Markov-Chain is typically programmed as follows:
  1. Initialise an X0 from a probability distribution
  2. Obtain X_{t+1} either from
     - Draw from a conditional probability distribution $p(X_{t+1}|X_t)$
     - Draw some U_t from a probability distribution and use a recurrence relation $X_{t+1} = g(X_t|U_t)$
  3. again repeat as necessary
- Based on Kroese et al (2013)

MCMC

- Markov Chain Monte-Carlo (MCMC) uses Markov chains to construct more effective Monte-Carlo methods. It can be loosely thought of as constructing the Monte-Carlo sample set using a random walk generated using Markov-chains.
- The general algorithm for MCMC is the Metropolis-Hastings Algorithm. for a probability distribution $f(x)$ known up to a normalisation factor,

1. Select a proposal density q(y | x) which is used to obtain a new trial sample y from the previous x in a similar as in the Markov-Chains described above.
2. Initialise with some X_0
3. For X_t generate Y_t from q(Y_t | X_t)
4. calculate the acceptance probability:

$$\alpha(x, y) = \min\left[\frac{f(y)q(x|y)}{f(x)q(y|x)}, 1\right] \tag{4}$$

1. generate a random number $U \in [0, 1]$ and obtain X_{t+1},

$$X_{t+1} = \begin{cases} Y, U \leq \alpha(X_t, Y_t), \\ X_t, \text{ otherwise} \end{cases} \tag{5}$$

- Based on Kroese et al (2013)
- Different forms of MCMC make use of different trial functions.
- One method of preforming MCMC on multi-dimensional datasets, particularly useful if the domain is disjoint is the hit and run algorithm. This can be summarised as follows,
  1. Start with a point X_0 in the domain.
  2. Pick a direction (random unit vector)
  3. Pick a step to move along that direction according to a distribution function. Use the accept/reject method above.
- MCMC tend towards the correct distribution with large sample sizes.

Exercises

- Not really an exercise but look at `scipy.stats`, maybe write a hypothesis test
- Preform PCA on provided dataset (this is a fairly trivial example)
- Write a Monte-Carlo algorithm that samples an N-dimensional Gaussian distribution
- Write a random walker.
- Write a model for interaction with random scatterers. In it have a particle move deterministic ally (for instance according to newtons laws) and have a periodic interaction with a second particle with properties (e.g. velocity and direction) selected from a distribution function. The classic set up for this would be in gravitational N-body interactions:
  - Have a pair of point masses in which orbit each other due to Newtonian Gravity. Hence they will follow Ellipses. Note these will still need to be integrated forward as the time dependence is non-trivial if the orbit is eccentric.
  - Periodically (or better yet selected from a distribution) one of the particles has an encounter with a low mass scatterer (100-1000 times

lower mass works well). This approaches from infinity on a hyperbolic orbit and induces the following change in velocity in a frame moving with the particle,

$$\Delta V_{\parallel} = \frac{2m_s|u|\sin^2(\theta)}{m_s + m_a} \tag{6}$$

$$\Delta V_{\perp} = \frac{-m_s|u|\sin\theta}{m_s + m_a} \tag{7}$$

  - Where $\Delta V_{\parallel}$ is the change in velocity parallel to the particles initial motion; $\Delta V_{\perp}$ perpendicular; m_s the scatterer mass; m_a the particle mass; u the scatterer velocity and θ the scattering angle of the scatterer.
  - The scatterer properties should be selected from a probability distribution.
  - The two particles will follow a new orbit until the next scattering event occurs.
- Sample a square (or some other finite domain) uniformly

References

- Kroese, D.P., Taimre, T. and Botev, Z.I., 2013. Handbook of monte carlo methods (Vol. 706). John Wiley & Sons.