MECH60017/MECH96014/MECH96038 STATISTICS COMPUTING TUTORIAL SHEET II

Instructions and relevant commands are written in **blue** for Python and in **red** for R.The resulting figures presented may differ depending on which language you use.

The following exercises using Python/R are (1) to help you become familiar with simulating from probability models, and (2) become familiar with some standard probability distributions.

1 Simulation of the probability that athletes took drugs given a positive test result

In this exercise you will simulate the conditional probability that an athlete took drugs, given he/she failed a drug test. Useful commands in Python are `uniform` from the `random` module of the `NumPy` library, with `uniform(0, 1, n)` generating `n` random numbers between 0 and 1 and `randint` from the `random` module of the `NumPy` library, with `randint(low=..., high=..., size=n)` generating `n` random integers between the value of the `low` argument and the `high` argument minus 1 (due to the Python indexing being slightly different). In R, one can use `runif(n, 0, 1)` to generate `n` random numbers between 0 and 1 and `sample.int(n, size=..., replace=...)` to generate as many random integers as specified by the `size` argument, which will be between 1 and `n`, while the `replace` argument takes either `TRUE` or `FALSE` as inputs, to indicate if sampling is done with replacement.

Scenario: Suppose you have a population of 100 athletes. It is known that 10% of athletes take drugs. There is a test that will correctly identify 80% of drug users and also incorrectly identify 20% of clean athletes as drug users.

Simulate this situation: Each athlete in population of size 100 has probability 0.1 of being a drug taker. For each simulated athlete, generate a random integer 1-10 - if this is 1 the simulated athlete takes drugs, if this is 2-10 does not take drugs. 80% of the population is correctly identified by the test - for each simulated athlete, generate a second random integer 1-10 - if this is 1 or 2 the simulated athlete's test is incorrect, otherwise correct.

Athletes who fail the test include drug takers correctly identified as cheats and clean athletes incorrectly identified as cheats. So a simulated athlete is deemed to have failed if his/her 1st random integer is 1 and 2nd random integer is 3-10 OR 1st integer 2-10 and 2nd integer 1-2.

You are now in a position to calculate the first simulated conditional probability of drug taking among the athletes who failed the test.

<u>Tasks</u>

1. Repeat this simulation many times (for example, 1000 times). Plot (histogram or density) the resulting simulated probabilities. What would you consider to be an unusual result?
2. Try changing the assumptions/ parameters of the model - for instance, change the rate of drug taking in the population of athletes, the proportion of drug takers correctly identified by the test, the proportion of clean athletes incorrectly identified (in this example these last two proportions are equal, but this would usually not be the case).
3. Now simulate the situation where the athlete is only deemed to have failed if he/she has failed two tests, where the tests are conditionally independent given the athlete's drug-taking status.
4. Can you suggest ways to improve the simulation (e.g. write code efficiently)?
5. Use Bayes' Theorem to calculate the conditional probability of drug taking given failing (a) one test (b) two tests that are conditionally independent given drug-taking status. (Use the probabilites as specified in the original scenario.) Does this agree with your simulation?

2 Probability Distributions

Python/R has commands to calculate the probability mass function (p.m.f.) / probability density function (p.d.f.) and cumulative distribution function (c.d.f.) of a number of common probability distributions, and also to draw random samples from these distributions.

- Binomial (Use help to look up the commands: `binomial` from the `random` module of the `NumPy` library, `binom.pmf` and `binom.cdf` from the `stats` module of the `SciPy` library / `rbinom`, `dbinom` and `pbinom`).
- Geometric (Use help to look up the commands: `geometric` from the `random` module of the `NumPy` library, `geom.pmf` and `geom.cdf` from the `stats` module of the `SciPy` library / `rgeom`, `dgeom` and `pgeom`).

  Note that there are different parametrisations of the geometric distribution, so it's worth checking if Python/R is using the one you want it to use.
- Poisson (Use help to look up the commands: `poisson` from the `random` module of the `NumPy` library, `poisson.pmf` and `poisson.cdf` from the `stats` module of the `SciPy` library / `rpois`, `dpois` and `ppois`).
- Continuous Uniform (Use help to look up the commands: `uniform` from the `random` module of the `NumPy` library, `uniform.pdf` and `uniform.cdf` from the `stats` module of the `SciPy` library / `runif`, `dunif` and `punif`).
- Exponential (Use help to look up the commands: `exponential` from the `random` module of the `NumPy` library, `expon.pdf` and `expon.cdf` from the `stats` module of the `SciPy` library / `rexp`, `dexp` and `pexp`).
- Normal (Use help to look up the commands: `normal` from the `random` module of the `NumPy` library, `norm.pdf` and `norm.cdf` from the `stats` module of the `SciPy` library / `rnorm`, `dnorm` and `pnorm`).

Note that Python/R functions for the Normal distribution require as input the standard deviation σ, not the variance σ^2.

- Student t (Use help to look up the commands: `standard_t` from the `random` module of the `NumPy` library, `t.pdf` and `t.cdf` from the `stats` module of the `SciPy` library / `rt`, `dt` and `pt`).

  Note that the `standard_t` from the `random` module of the `NumPy` library / `rt` function generates random numbers from $X \sim \text{Student}(0, 1, \nu)$. You need to apply the transformation $Z = \sigma X + \mu$ to get $Z \sim \text{Student}(\mu, \sigma^2, \nu)$.

- Gamma (Use help to look up the commands: `gamma` from the `random` module of the NumPy library, `gamma.pdf` and `gamma.cdf` from the `stats` module of the `SciPy` library / `rgamma`, `dgamma` and `pgamma`).

  The Exponential distribution is a special case of the Gamma distribution.

<u>Tasks</u>

Initially choose two distributions (one discrete and one continuous).

1. Plot the p.m.f. or p.d.f. and c.d.f. for a suitable choice of parameters for each distribution. You decide the range of values and parameter(s) for each distribution. For example, `z = norm.pdf(x, 1, 2)` / `z = rnorm(x, 1, 2)` will calculate the probability densities for a $N(1, 4)$ distribution at each value in `x` - where `x` would typically be a vector of values suitable to plot the p.d.f. - for instance here `x = linspace(-10, 10, 100)` (`linspace` can be loaded from the `NumPy` library) / `x = seq(-10, 10, l=100)`.

2. Generate $n = 100$ random numbers from each distribution, e.g. `x = random.normal(1, 2, 100)` / `x = rnorm(100, 1, 2)` will generate 100 random numbers from a Normal distribution with mean 1 and variance 4.

   (a) Plot histograms of the *relative* frequency distribution of your random sample using appropriate bin widths. In Python, you may use `histogram` from the `NumPy` library. In R, you may use `hist`.

   (b) Overlay the true probability density function. Do they match?

   (c) Analytically calculate the theoretical mean and variance given your choice of parameters.

   (d) Calculate the sample mean (`mean`) and variance (`var`) for your random sample. Do they match the theoretical quantities?

   (e) Repeat (a-d) using $n = 200, 500, 1000$. What happens as the sample size increases?

Repeat the tasks for the other distributions as a homework exercise.