Statistics Demonstrations-Updated

September 23, 2024

0.1 Statistics Demonstrations

- Distributions (Binomial and Normal)
- Random Numbers
- Combinations

[4]:
```
#Importing the libraries
import numpy as np
#Introducing the scipy.stats library
import scipy.stats as ss # Scipy is scientific python , stats is statistics
```

0.1.1 Distributions

1. Binomial Distribution A `binomial distribution` is utilised when the following conditions are met - Total number of trials is fixed at **n**

- Each trial is binary, i.e., has only two possible outcomes - success or failure
- Probability of success is same in all trials, denoted by **p** Now if you want to find the probability of **k** successes or `P(X = k)`, then you can use the following formula:

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

Where:

- $P(X = k)$ is the probability of getting exactly k successes in n trials.
- $\binom{n}{k}$ is the binomial coefficient, calculated as $\frac{n!}{k!(n-k)!}$, representing the number of ways to choose k successes from n trials.
- p is the probability of success on a single trial.
- $1 - p$ is the probability of failure.
- n is the number of trials.
- k is the number of successes.

[6]:
```
n = 10    # number of times i tossed a coin , that is 10 times i tossed
p = 0.4   # probability if 4%
k = 4     # 4 times in got head , this the measure of success
```

[7]:
```
?ss.binom.pmf
```

```
Signature: ss.binom.pmf(k, *args, **kwds)
Docstring:
Probability mass function at k of the given RV.

Parameters
----------
k : array_like
    Quantiles.
arg1, arg2, arg3,… : array_like
    The shape parameter(s) for the distribution (see docstring of the
    instance object for more information)
loc : array_like, optional
    Location parameter (default=0).

Returns
-------
pmf : array_like
    Probability mass function evaluated at k
File:      /opt/conda/envs/anaconda-panel-2023.05-py310/lib/python3.11/
 ↪site-packages/scipy/stats/_distn_infrastructure.py
Type:      method
```

[8]:
```
ss.binom.pmf(k,n,p)
```

[8]:
```
0.25082265599999987
```

[9]:
```
np.round(ss.binom.pmf(k,n,p),3)
```

[9]:
```
0.251
```

1 Normal Distribution

Let's say you're given a normal distribution with

- mean = M
- standard deviation = sd

Now you are given a value **x** and you want to compute P(X<=**x**)

[11]:
```
M = 60
sd = 10
x = 40
```

[12]:
```
?ss.norm.cdf
```

```
Signature: ss.norm.cdf(x, *args, **kwds)
Docstring:
Cumulative distribution function of the given RV.

Parameters
----------
x : array_like
    quantiles
arg1, arg2, arg3,… : array_like
    The shape parameter(s) for the distribution (see docstring of the
    instance object for more information)
loc : array_like, optional
    location parameter (default=0)
scale : array_like, optional
    scale parameter (default=1)

Returns
-------
cdf : ndarray
    Cumulative distribution function evaluated at `x`
File:      /opt/conda/envs/anaconda-panel-2023.05-py310/lib/python3.11/
 ↪site-packages/scipy/stats/_distn_infrastructure.py
Type:      method
```

[13]:
```
ss.norm.cdf(x,M,sd)
```

[13]: 0.022750131948179195

[14]:
```
ss.norm(60,10).cdf(40)
```

[14]: 0.022750131948179195

[15]:
```
ss.norm.cdf(80,60,10)
```

[15]: 0.9772498680518208

[16]:
```
ss.norm.cdf(40,60,10)
```

[16]: 0.022750131948179195

[17]:
```
ss.norm.cdf(80,60,10) - ss.norm.cdf(40,60,10)
```

[17]: 0.9544997361036416

[18]:
```
# Generate random numbers for a given distribution
```

[19]:
```
?np.random
```

```
Type:        module
String form: <module 'numpy.random' from '/opt/conda/envs/anaconda-panel-2023.
 ↪05-py310/lib/python3.11/site-packages/numpy/random/__init__.py'>
File:        /opt/conda/envs/anaconda-panel-2023.05-py310/lib/python3.11/
 ↪site-packages/numpy/random/__init__.py
Docstring:
========================
Random Number Generation
========================

Use ``default_rng()`` to create a `Generator` and call its methods.

=============== ==========================================================
Generator
--------------- ----------------------------------------------------------
Generator       Class implementing all of the random number distributions
default_rng     Default constructor for ``Generator``
=============== ==========================================================

============================================ ===
BitGenerator Streams that work with Generator
-------------------------------------------- ---
MT19937
PCG64
PCG64DXSM
Philox
SFC64
============================================ ===

============================================ ===
Getting entropy to initialize a BitGenerator
-------------------------------------------- ---
SeedSequence
============================================ ===


Legacy
------

For backwards compatibility with previous versions of numpy before 1.17, the
various aliases to the global `RandomState` methods are left alone and do not
use the new `Generator` API.

==================== =========================================================
Utility functions
```

```
------------------- ------------------------------------------------------------
random              Uniformly distributed floats over ``[0, 1)``
bytes               Uniformly distributed random bytes.
permutation         Randomly permute a sequence / generate a random sequence.
shuffle             Randomly permute a sequence in place.
choice              Random sample from 1-D array.
=================== ============================================================

=================== ============================================================
Compatibility
functions - removed
in the new API
------------------- ------------------------------------------------------------
rand                Uniformly distributed values.
randn               Normally distributed values.
ranf                Uniformly distributed floating point numbers.
random_integers     Uniformly distributed integers in a given range.
                    (deprecated, use ``integers(…, closed=True)`` instead)
random_sample       Alias for `random_sample`
randint             Uniformly distributed integers in a given range
seed                Seed the legacy random number generator.
=================== ============================================================

=================== ============================================================
Univariate
distributions
------------------- ------------------------------------------------------------
beta                Beta distribution over ``[0, 1]``.
binomial            Binomial distribution.
chisquare           :math:`\chi^2` distribution.
exponential         Exponential distribution.
f                   F (Fisher-Snedecor) distribution.
gamma               Gamma distribution.
geometric           Geometric distribution.
gumbel              Gumbel distribution.
hypergeometric      Hypergeometric distribution.
laplace             Laplace distribution.
logistic            Logistic distribution.
lognormal           Log-normal distribution.
logseries           Logarithmic series distribution.
negative_binomial   Negative binomial distribution.
noncentral_chisquare Non-central chi-square distribution.
noncentral_f        Non-central F distribution.
normal              Normal / Gaussian distribution.
pareto              Pareto distribution.
poisson             Poisson distribution.
power               Power distribution.
rayleigh            Rayleigh distribution.
```

```
triangular           Triangular distribution.
uniform              Uniform distribution.
vonmises             Von Mises circular distribution.
wald                 Wald (inverse Gaussian) distribution.
weibull              Weibull distribution.
zipf                 Zipf's distribution over ranked data.
==================== ============================================================

==================== ============================================================
Multivariate
distributions
-------------------- ------------------------------------------------------------
dirichlet            Multivariate generalization of Beta distribution.
multinomial          Multivariate generalization of the binomial distribution.
multivariate_normal  Multivariate generalization of the normal distribution.
==================== ============================================================

==================== ============================================================
Standard
distributions
-------------------- ------------------------------------------------------------
standard_cauchy      Standard Cauchy-Lorentz distribution.
standard_exponential Standard exponential distribution.
standard_gamma       Standard Gamma distribution.
standard_normal      Standard normal distribution.
standard_t           Standard Student's t-distribution.
==================== ============================================================

==================== ============================================================
Internal functions
-------------------- ------------------------------------------------------------
get_state            Get tuple representing internal state of generator.
set_state            Set state of generator.
==================== ============================================================
```

1.0.1 Binomial Distribution

For binomial distribution, given the value of **n** (the number of trials) and **p** (the probability of success for each trial) we can generate random sequence of possible number of successes **r** when we run this experiment multiple times.

[21]:
```
?np.random.binomial
```

```
Docstring:
binomial(n, p, size=None)

Draw samples from a binomial distribution.
```

```
Samples are drawn from a binomial distribution with specified
parameters, n trials and p probability of success where
n an integer >= 0 and p is in the interval [0,1]. (n may be
input as a float, but it is truncated to an integer in use)

.. note::
    New code should use the `~numpy.random.Generator.binomial`
    method of a `~numpy.random.Generator` instance instead;
    please see the :ref:`random-quick-start`.

Parameters
----------
n : int or array_like of ints
    Parameter of the distribution, >= 0. Floats are also accepted,
    but they will be truncated to integers.
p : float or array_like of floats
    Parameter of the distribution, >= 0 and <=1.
size : int or tuple of ints, optional
    Output shape.  If the given shape is, e.g., ``(m, n, k)``, then
    ``m * n * k`` samples are drawn.  If size is ``None`` (default),
    a single value is returned if ``n`` and ``p`` are both scalars.
    Otherwise, ``np.broadcast(n, p).size`` samples are drawn.

Returns
-------
out : ndarray or scalar
    Drawn samples from the parameterized binomial distribution, where
    each sample is equal to the number of successes over the n trials.

See Also
--------
scipy.stats.binom : probability density function, distribution or
    cumulative density function, etc.
random.Generator.binomial: which should be used for new code.

Notes
-----
The probability density for the binomial distribution is

.. math:: P(N) = \binom{n}{N}p^N(1-p)^{n-N},

where :math:`n` is the number of trials, :math:`p` is the probability
of success, and :math:`N` is the number of successes.

When estimating the standard error of a proportion in a population by
using a random sample, the normal distribution works well unless the
product p*n <=5, where p = population proportion estimate, and n =
number of samples, in which case the binomial distribution is used
```

```
instead. For example, a sample of 15 people shows 4 who are left
handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4,
so the binomial distribution should be used in this case.

References
----------
.. [1] Dalgaard, Peter, "Introductory Statistics with R",
       Springer-Verlag, 2002.
.. [2] Glantz, Stanton A. "Primer of Biostatistics.", McGraw-Hill,
       Fifth Edition, 2002.
.. [3] Lentner, Marvin, "Elementary Applied Statistics", Bogden
       and Quigley, 1972.
.. [4] Weisstein, Eric W. "Binomial Distribution." From MathWorld--A
       Wolfram Web Resource.
       http://mathworld.wolfram.com/BinomialDistribution.html
.. [5] Wikipedia, "Binomial distribution",
       https://en.wikipedia.org/wiki/Binomial_distribution

Examples
--------
Draw samples from the distribution:

>>> n, p = 10, .5  # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.

A real world example. A company drills 9 wild-cat oil exploration
wells, each with an estimated probability of success of 0.1. All nine
wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that
generate zero positive results.

>>> sum(np.random.binomial(9, 0.1, 20000) == 0)/20000.
# answer = 0.38885, or 38%.
Type:      builtin_function_or_method
```

[22]:
```
# Let us generate 10 random numbers from Binomial
```

[23]:
```
n = 10
p = 0.4
size = 10
np.random.binomial(n,p,size)
```

[23]:
```
array([7, 4, 3, 3, 6, 0, 4, 4, 3, 4])
```

1.0.2 Normal Distribution

For normal distribution, given the value of mean (**M**) and standard deviation (**sd**) we can generate random sequence of values belonging to this distribution

[25]:
```
?np.random.normal # ? is the help function (What is np.random.normal?)
```

```
Object `np.random.normal # ? is the help function (What is np.random.normal?)`
not found.
```

[26]:
```
M = 60
sd = 10
np.random.normal(M,sd,10)
```

[26]:
```
array([65.42709854, 71.05547796, 72.81903122, 64.88387427, 47.88066188,
       56.43691405, 54.52811441, 60.78701889, 69.74555022, 63.47721017])
```

1.0.3 In order to fix the output we can set the seed value

[28]:
```
?np.random.seed
```

```
Docstring:
seed(seed=None)

Reseed the singleton RandomState instance.

Notes
-----
This is a convenience, legacy function that exists to support
older code that uses the singleton RandomState. Best practice
is to use a dedicated ``Generator`` instance rather than
the random variate generation methods exposed directly in
the random module.

See Also
--------
numpy.random.Generator
Type:      builtin_function_or_method
```

[29]:
```
seed = 100
np.random.seed(seed)
```

[30]:
```
np.random.normal(M,sd,10)
```

[30]:
```
array([42.50234527, 63.42680403, 71.53035803, 57.47563963, 69.81320787,
       65.14218841, 62.21179669, 49.29956669, 58.10504169, 62.55001444])
```

[31]:
```
np.random.normal(M,sd,10)
```

[31]:
```
array([55.41973014, 64.35163488, 54.1640495 , 68.16847072, 66.72720806,
       58.95588857, 54.68719623, 70.29732685, 55.61864377, 48.81681754])
```

The seed has to be initialised each time you run the code to get the same output

1.0.4 Combinations

Given a list containing **n** number of objects, we can find all the combinations of size **r**

[34]:
```
from  itertools import combinations
```

[35]:
```
?combinations
```

```
Init signature: combinations(iterable, r)
Docstring:
Return successive r-length combinations of elements in the iterable.

combinations(range(4), 3) --> (0,1,2), (0,1,3), (0,2,3), (1,2,3)
Type:           type
Subclasses:
```

[36]:
```
alpha = 'ABCD'
p = combinations(alpha, 2)
p
```

[36]:
```
<itertools.combinations at 0x7faca1baf380>
```

[37]:
```
list(p)
```

[37]:
```
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]
```

[38]:
```
alpha = ['A','B','C','D']
p = combinations(alpha, 2)
list(p)
```

[38]:
```
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]
```

[39]:
```
num = [1,2,3]
num
```

[39]:
```
[1, 2, 3]
```

[40]:
```
list(combinations(num, 2))
```

[40]:
```
[(1, 2), (1, 3), (2, 3)]
```

[41]:
```
for c in combinations(num, 2):
    if 2 in c:
```

```
        print(c)
```

```
(1, 2)
(2, 3)
```

[42]:
```
for c in combinations(alpha, 2):
    if 'A' in c:
        print(c)
```

```
('A', 'B')
('A', 'C')
('A', 'D')
```

1.0.5 Working on Beautiful Soup library to fetch data from webpages

[44]:
```
pip install requests beautifulsoup4
```

```
Defaulting to user installation because normal site-packages is not writeable
Looking in links: /usr/share/pip-wheels
Requirement already satisfied: requests in /opt/conda/envs/anaconda-
panel-2023.05-py310/lib/python3.11/site-packages (2.31.0)
Requirement already satisfied: beautifulsoup4 in /opt/conda/envs/anaconda-
panel-2023.05-py310/lib/python3.11/site-packages (4.12.2)
Requirement already satisfied: charset-normalizer<4,>=2 in
/opt/conda/envs/anaconda-panel-2023.05-py310/lib/python3.11/site-packages (from
requests) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in /opt/conda/envs/anaconda-
panel-2023.05-py310/lib/python3.11/site-packages (from requests) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /opt/conda/envs/anaconda-
panel-2023.05-py310/lib/python3.11/site-packages (from requests) (1.26.16)
Requirement already satisfied: certifi>=2017.4.17 in /opt/conda/envs/anaconda-
panel-2023.05-py310/lib/python3.11/site-packages (from requests) (2023.7.22)
Requirement already satisfied: soupsieve>1.2 in /opt/conda/envs/anaconda-
panel-2023.05-py310/lib/python3.11/site-packages (from beautifulsoup4) (2.4)
Note: you may need to restart the kernel to use updated packages.
```

[45]:
```
import requests
from bs4 import BeautifulSoup
```

[46]:
```
url = "https://www.bing.com/images/search?q=Archana+Hebbar&form=RESTAB&first=1"
response = requests.get(url)
soup = BeautifulSoup(response.content,'html.parser')
```

[47]:
```
print(soup.title.string)
```

```
Archana Hebbar - Search Images
```

[ ]:
```

```