COPADS III (Compendium of Distributions II): Cauchy, Cosine, Exponential, Hypergeometric, Logarithmic, Semicircular, Triangular, and Weibull

Kenneth FQ Chen
School of Chemical and Life Sciences, Singapore Polytechnic, Singapore
kennethjoel@hotmail.co.uk

Maurice HT Ling
Department of Zoology, The University of Melbourne, Australia
mauriceling@acm.org

Abstract

This manuscript illustrates the implementation and testing of eight statistical distributions, namely Cauchy, Cosine, Exponential, Hypergeometric, Logarithmic, Semicircular, Triangular, and Weibull distribution, where each distribution consists of three common functions – Probability Density Function (PDF), Cumulative Density Function (CDF) and the inverse of CDF (inverseCDF). These codes had been incorporated into COPADS codebase (https://github.com/copads/ copads) are licensed under Lesser General Public Licence version 3.

1. Description

Statistical distributions play a central role in statistical inferences to provide a probabilistic measure for use in hypothesis testing. As such, the implementation of statistical distributions and functions is fundamental to high-throughput scientific analyses. Ten statistical distributions had been implemented in previous reports (Ling, 2009a; Ling, 2009a). This manuscript expands on the work of Ling (2009a) with another eight statistical distributions; namely Cauchy, Cosine, Exponential, Hypergeometric, Logarithm, Semicircular, Triangular, and Weibull; where each distribution consists of three common functions – Probability Density Function (PDF), Cumulative Density Function (CDF) and the inverse of CDF (inverseCDF) – as modelled after Ling (2009b).

Each distribution can be briefly described as follows:

- Cauchy distribution is a continuous distribution named after Augustine Cauchy. An important characteristic of Cauchy distribution compared to other distributions is that the mean and variance of Cauchy distribution cannot be algebraically defined.
- Cosine distribution is a continuous distribution and had been used in mechanics for modelling and simulating scattered particle movements (Greenwood, 2002).
- Exponential distribution is also known as negative exponential distribution. It is commonly used to estimate probabilities between events, such as average time between failures (Balakrishnan et al., 2009). Hence, exponential distribution can be seen as the continuous counterpart of geometric distribution (Ling, 2009a).
- Hypergeometric distribution is a discrete distribution commonly used in sampling methodologies (Sathakathulla and Murthy, 2012) to estimate the probability of k-

successes in n-trials without replacement. This is in contrast to binomial distribution (Ling, 2009a), which is used to estimate the probability of k-successes in n-trials with replacement.

- Logarithmic distribution, also known as logarithmic series distribution or the log-series distribution is a discrete distribution derived from the Maclaurin series expansion. It had been used to estimate animal population from capture-release-recapture methodology (Darwin, 1960) and successes in exploration studies (Ghannadpour et al., 2013).
- Semicircular distribution is a continuous distribution and commonly used as a simplified estimation of normal distribution.
- Triangular distribution is a continuous distribution with an upper and lower boundary, as well as a peak probability. Hence, it is commonly used for three-point estimations of best-case, most likely, and worst-case estimates in risk analyses. Due to its simplicity, triangular distribution had been used as proxy for beta distribution (Johnson, 1997; Joo and Casella, 2001).
- Weibull distribution is a continuous distribution with a wide variety of applications (Forbes et al., 2011; Rinne, 2010). These include survivorship or failure analysis (Pinder III et al., 1978), meteorology (Islam et al., 2011), estimating measurement uncertainties (Bermejo et al., 2012), financial risk estimation (Chen and Gerlach, 2013), and inventory management (Yang, 2012). Weibull distribution is also known as Frechet distribution.

Given that the equation of a distribution is *p(x)* and area under a distribution is standardized to 1: the Cumulative Density Function (CDF) of value *x* is the area under the distribution bounded by negative infinity to *x*; the Probability Density Function (PDF) is the probability of *x* for discrete distributions and between *x-h* and *x+h* for continuous distribution where *h* is a small float number; the inverse of CDF (inverseCDF) gives the value of *x* when given a probability.

$$CDF(x) = \int_{-\infty}^{x} p(x)\,dx$$

$$PDF(x) = \int_{x+h}^{x-h} p(x)\,dx$$

These codes had been incorporated into COPADS codebase (https://github.com/copads/copads) are licensed under Lesser General Public Licence version 3.

2. Code Files

The implementation of the distributions and testing codes are presented in 2 files:

- copadsIII.py file contains the implementation of each distribution.
- t_copadsIII.py file contains the test codes for each distribution.

File: copadsIII.py

```python
import random
import math

PI = 3.14159265358979323846
PI2 = 6.2831853071795864769252867665590057683943387987502
```

```
def bico(n, k):
    """
    Binomial coefficient. Returns n!/(k!(n-k)!)
    Depend: factln, gammln
    @see: NRP 6.1

    @see: Ling, MHT. 2009. Compendium of Distributions, I: Beta, Binomial,
    Chi-Square, F, Gamma, Geometric, Poisson, Student's t, and Uniform.
    The Python Papers Source Codes 1:4

    @param n: total number of items
    @param k: required number of items
    @return: floating point number

    @status: Tested function
    @since: version 0.1
    """
    return math.floor(math.exp(factln(n) - factln(k) - factln(n-k)))

def factln(n):
    """
    Natural logarithm of factorial: ln(n!)
    @see: NRP 6.1
    @see: Ling, MHT. 2009. Compendium of Distributions, I: Beta, Binomial,
    Chi-Square, F, Gamma, Geometric, Poisson, Student's t, and Uniform.
    The Python Papers Source Codes 1:4

    @param n: positive integer
    @return: natural logarithm of factorial of n
    """
    return gammln(n + 1.0)

def gammln(n):
    """
    Complete Gamma function.
    @see: NRP 6.1
    @see: http://mail.python.org/pipermail/python-list/
    2000-June/671838.html
    @see: Ling, MHT. 2009. Compendium of Distributions, I: Beta, Binomial,
    Chi-Square, F, Gamma, Geometric, Poisson, Student's t, and Uniform.
    The Python Papers Source Codes 1:4

    @param n: float number
    @return: float number

    @status: Tested function
    @since: version 0.1
    """
    gammln_cof = [76.18009173, -86.50532033, 24.01409822,
                  -1.231739516e0, 0.120858003e-2, -0.536382e-5]
    x = n - 1.0
    tmp = x + 5.5
    tmp = (x + 0.5) * math.log(tmp) - tmp
    ser = 1.0
    for j in range(6):
        x = x + 1.
        ser = ser + gammln_cof[j] / x
    return tmp + math.log(2.50662827465 * ser)

class Distribution:
    """
    Abstract class for all statistical distributions.
    Due to the large variations of parameters for each distribution,
    it is unlikely to be able to standardize a parameter list for each
    method that is meaningful for all distributions. Instead, the
    parameters to construct each distribution is to be given as
    keyword arguments.
    """

    def __init__(self, **parameters):
        """
        Constructor method. The parameters are used to construct the
        probability distribution.
        """
        raise NotImplementedError

    def CDF(self, x):
        """
        Cumulative Distribution Function, which gives the cumulative
        probability (area under the probability curve) from -infinity
        or 0 to a give x-value on the x-axis where y-axis is the
        probability. CDF is also known as density function.
        """
        raise NotImplementedError

    def PDF(self, x):
        """
        Partial Distribution Function, which gives the probability for
        The particular value of x, or the area under probability
        distribution from x-h to x+h for continuous distribution.
        """
        raise NotImplementedError

    def inverseCDF(self, probability, start=0.0, step=0.01):
        """
        It does the reverse of CDF() method, it takes a probability
        value and returns the corresponding value on the x-axis.
        """
        raise NotImplementedError

    def mean(self):
        """
        Gives the arithmetic mean of the sample.
        """
        raise NotImplementedError

    def mode(self):
        """
        Gives the mode of the sample, if closed-form is available.
        """
        raise NotImplementedError

    def kurtosis(self):
        """
        Gives the kurtosis of the sample.
        """
        raise NotImplementedError

    def skew(self):
        """
        Gives the skew of the sample.
        """
        raise NotImplementedError

    def variance(self):
        """
        Gives the variance of the sample.
        """
        raise NotImplementedError


class CauchyDistribution(Distribution):
    """
    Class for Cauchy Distribution.

    @status: Tested method
    @since: version 0.4
    """

    def __init__(self, location=0.0, scale=1.0):
        """
        Constructor method. The parameters are used to construct the
        probability distribution.

        @param location: the mean; default = 0.0
        @param scale: spread of the distribution, S{lambda}; default = 1.0
        """
        self.location = location
        self.scale = scale

    def CDF(self, x):
        """
        Cumulative Distribution Function, which gives the cumulative
        probability (area under the probability curve) from -infinity or
        0 to a give x-value on the x-axis where y-axis is the probability.
        """
        return 0.5 + 1 / PI * math.atan((x - self.location) / self.scale)

    def PDF(self, x):
        """
        Partial Distribution Function, which gives the probability for the
        particular value of x, or the area under probability distribution
        from x-h to x+h for continuous distribution.
        """
        return 1 / (PI * self.scale * \
            (1 + (((x - self.location) / self.scale) ** 2)))

    def inverseCDF(self, probability, start=0.0, step=0.01):
        """
        It does the reverse of CDF() method, it takes a probability value
        and returns the corresponding value on the x-axis.
        """
        cprob = self.CDF(start)
        if probability < cprob: return (start, cprob)
        while (probability > cprob):
            start = start + step
            cprob = self.CDF(start)
            # print start, cprob
        return (start, cprob)

    def mean(self):
        """Gives the arithmetic mean of the sample."""
        raise DistributionFunctionError('Mean for Cauchy Distribution is \
            undefined')

    def mode(self):
        """Gives the mode of the sample."""
        return self.location

    def median(self):
        """Gives the median of the sample."""
        return self.location

    def quantile1(self):
        """Gives the 1st quantile of the sample."""
        return self.location - self.scale

    def quantile3(self):
        """Gives the 3rd quantile of the sample."""
        return self.location + self.scale

    def qmode(self):
        """Gives the quantile of the mode of the sample."""
        return 0.5

    def random(self, seed):
        """Gives a random number based on the distribution."""
        while 1:
            seed = self.loaction + (self.scale * \
                                    math.tan(PI * (seed - 0.5)))
            yield seed


class CosineDistribution(Distribution):
    """
    Cosine distribution is sometimes used as a simple approximation to
    Normal distribution.

    @status: Tested method
    @since: version 0.4
    """

    def __init__(self, location=0.0, scale=1.0):
        """
        Constructor method. The parameters are used to construct the
        probability distribution.

        @param location: the mean; default = 0.0
        @param scale: spread of the distribution, S{lambda}; default = 1.0
        """
        self.location = location
        self.scale = scale

    def CDF(self, x):
        """
        Cumulative Distribution Function, which gives the cumulative
        probability (area under the probability curve) from -infinity or
        0 to a give x-value on the x-axis where y-axis is the probability.
        """
        n = PI + (x - self.location) / self.scale + \
            math.sin((x - self.location) / self.scale)
        return n / PI2

    def PDF(self, x):
        """
        Partial Distribution Function, which gives the probability for the
        particular value of x, or the area under probability distribution
        from x-h to x+h for continuous distribution.
        """
        return (1 / (PI2 * self.scale)) * \
                (1 + math.cos((x - self.location) / self.scale))

    def inverseCDF(self, probability, start=0.0, step=0.01):
        """
        It does the reverse of CDF() method, it takes a probability value
        and returns the corresponding value on the x-axis.
        """
        cprob = self.CDF(start)
        if probability < cprob: return (start, cprob)
        while (probability > cprob):
            start = start + step
            cprob = self.CDF(start)
            # print start, cprob
        return (start, cprob)

    def mean(self):
        """Gives the arithmetic mean of the sample."""
        return self.location

    def mode(self):
        """Gives the mode of the sample."""
        return self.location

    def median(self):
        """Gives the median of the sample."""
        return self.location

    def kurtosis(self):
        """Gives the kurtosis of the sample."""
        return -0.5938

    def skew(self):
        """Gives the skew of the sample."""
        return 0.0

    def variance(self):
        """Gives the variance of the sample."""
        return (((PI * PI)/3) - 2) * (self.scale ** 2)

    def quantile1(self):
        """Gives the 1st quantile of the sample."""
        return self.location - (0.8317 * self.scale)

    def quantile3(self):
        """Gives the 13rd quantile of the sample."""
        return self.location + (0.8317 * self.scale)

    def qmean(self):
        """Gives the quantile of the arithmetic mean of the sample."""
        return 0.5

    def qmode(self):
        """Gives the quantile of the mode of the sample."""
        return 0.5
```

```
class ExponentialDistribution(Distribution):
    """
    Exponential distribution is the continuous version of Geometric
    distribution. It is also a special case of Gamma distribution where
    shape = 1

    @status: Tested method
    @since: version 0.4
    """

    def __init__(self, location=0.0, scale=1.0):
        """
        Constructor method. The parameters are used to construct the
        probability distribution.

        @param location: position of the distribution, default = 0.0
        @param scale: spread of the distribution, S{lambda}; default = 1.0
        """
        self.location = location
        self.scale = scale

    def CDF(self, x):
        """
        Cumulative Distribution Function, which gives the cumulative
        probability (area under the probability curve) from -infinity or
        0 to a give x-value on the x-axis where y-axis is the probability.
        """
        return 1 - math.exp((self.location - x) / self.scale)

    def PDF(self, x):
        """
        Partial Distribution Function, which gives the probability for the
        particular value of x, or the area under probability distribution
        from x-h to x+h for continuous distribution.
        """
        return (1/self.scale) * math.exp((self.location - x)/self.scale)

    def inverseCDF(self, probability, start=0.0, step=0.01):
        """
        It does the reverse of CDF() method, it takes a probability value
        and returns the corresponding value on the x-axis.
        """
        cprob = self.CDF(start)
        if probability < cprob: return (start, cprob)
        while (probability > cprob):
            start = start + step
            cprob = self.CDF(start)
            # print start, cprob
        return (start, cprob)

    def mean(self):
        """Gives the arithmetic mean of the sample."""
        return self.location + self.scale

    def mode(self):
        """Gives the mode of the sample."""
        return self.location

    def median(self):
        """Gives the median of the sample."""
        return self.location + (self.scale * math.log10(2))

    def kurtosis(self):
        """Gives the kurtosis of the sample."""
        return 6.0

    def skew(self):
        """Gives the skew of the sample."""
        return 2.0

    def variance(self):
        """Gives the variance of the sample."""
        return self.scale * self.scale

    def quantile1(self):
        """Gives the 1st quantile of the sample."""
        return self.location + (self.scale * math.log10(1.333))

    def quantile3(self):
        """Gives the 3rd quantile of the sample."""
        return self.location + (self.scale * math.log10(4))

    def qmean(self):
        """Gives the quantile of the arithmetic mean of the sample."""
        return 0.6321

    def qmode(self):
        """Gives the quantile of the mode of the sample."""
        return 0.0

    def random(self):
        """Gives a random number based on the distribution."""
        return random.expovariate(1/self.location)


class HypergeometricDistribution(Distribution):
    """
    Class for Hypergeometric distribution

    @status: Tested method
    @since: version 0.4
    """

    def __init__(self, sample_size,
                 population_size=100,
                 population_success=50):
        """
        Constructor method. The parameters are used to construct the
        probability distribution.

        @param sample_size: sample size (not more than population size)
        @type sample_size: integer
        @param population_size: population size; default = 100
        @type population_size: integer
        @param population_success: number of successes in the population
        (cannot be more than population size); default = 10
        @type population_success: integer"""
        if population_success > population_size:
            raise AttributeError('population_success cannot be more \
            than population_size')
        elif sample_size > population_size:
                raise AttributeError('sample_size cannot be more \
                than population_size')
            else:
                self.psize = int(population_size)
                self.psuccess = int(population_success)
                self.ssize = int(sample_size)

    def CDF(self, sample_success):
        """
        Cumulative Distribution Function, which gives the cumulative
        probability (area under the probability curve) from -infinity or
        0 to a give x-value (sample_success, an integer that is not more
        than sample size) on the x-axis where y-axis is the probability.
        """
        if sample_success > self.ssize:
            raise AttributeError('sample_success cannot be more \
            than sample_size')
        else:
            return sum([self.PDF(n) for n in range(1, sample_success+1)])

    def PDF(self, sample_success):
        """
        Partial Distribution Function, which gives the probability for the
        particular value of x (sample_success, an integer that is not more
        than sample size), or the area under probability distribution from
        x-h to x+h for continuous distribution."""
        if sample_success > self.ssize:
            raise AttributeError('sample_success cannot be more \
            than sample_size')
        else:
            sample_success = int(sample_success)
            numerator = bico(self.psuccess, sample_success)
            numerator = numerator  * bico(self.psize-self.psuccess,
                                          self.ssize-sample_success)
            denominator = bico(self.psize, self.ssize)
            return float(numerator)/float(denominator)

    def inverseCDF(self, probability, start=1, step=1):
        """
        It does the reverse of CDF() method, it takes a probability value
        and returns the corresponding value on the x-axis."""
        cprob = self.CDF(start)
        if probability < cprob: return (start, cprob)
        while (probability > cprob):
            start = start + step
            cprob = self.CDF(start)
            # print start, cprob
        return (int(start), cprob)

    def mean(self):
        """Gives the arithmetic mean of the sample."""
        return self.ssize * (float(self.psuccess)/float(self.psize))

    def mode(self):
        """Gives the mode of the sample."""
        temp = (self.ssize + 1) * (self.psuccess + 1)
        return float(temp)/float(self.psize + 2)

    def variance(self):
        """Gives the variance of the sample."""
        t1 = float(self.psize-self.psuccess)/float(self.psize)
```

```
            t2 = float(self.psize-self.ssize)/float(self.psize-1)
            return self.mean() * t1 * t2


class LogarithmicDistribution(Distribution):
    """
    Class for Logarithmic Distribution.

    @status: Tested method
    @since: version 0.4
    """

    def __init__(self, shape):
        """Constructor method. The parameters are used to construct the
        probability distribution.

        @param shape: the spread of the distribution"""
        self.shape = shape

    def CDF(self, x):
        """
        Cumulative Distribution Function, which gives the cumulative
        probability (area under the probability curve) from -infinity or
        0 to a give x-value on the x-axis where y-axis is the probability.
        """
        summation = 0.0
        for i in range(int(x)): summation = summation + self.PDF(i)
        return summation

    def PDF(self, x):
        """
        Partial Distribution Function, which gives the probability for the
        particular value of x, or the area under probability distribution
        from x-h to x+h for continuous distribution.
        """
        return (-1 * (self.shape ** x)) / (math.log10(1 - self.shape) * x)

    def inverseCDF(self, probability, start=0.0, step=0.01):
        """
        It does the reverse of CDF() method, it takes a probability value
        and returns the corresponding value on the x-axis.
        """
        cprob = self.CDF(start)
        if probability < cprob: return (start, cprob)
        while (probability > cprob):
            start = start + step
            cprob = self.CDF(start)
            # print start, cprob
        return (start, cprob)

    def mean(self):
        """Gives the arithmetic mean of the sample."""
        return (-1 * self.shape) / ((1 - self.shape) * \
                math.log10(1 - self.shape))

    def mode(self):
        """Gives the mode of the sample."""
        return 1.0

    def variance(self):
        """Gives the variance of the sample."""
        n = (-1 * self.shape) * (self.shape + math.log10(1 - self.shape))
        d = ((1 - self.shape) ** 2) * math.log10(1 - self.shape) * \
            math.log10(1 - self.shape)
        return n / d


class SemicircularDistribution(Distribution):
    """
    Class for Semicircular Distribution.

    @status: Tested method
    @since: version 0.4
    """

    def __init__(self, location=0.0, scale=1.0):
        """
        Constructor method. The parameters are used to construct the
        probability distribution.

        @param location: mean of the distribution, default = 0.0
        @param scale: spread of the distribution, default = 1.0"""
        self.location = location
        self.scale = scale

    def CDF(self, x):
        """
        Cumulative Distribution Function, which gives the cumulative
        probability (area under the probability curve) from -infinity or
        0 to a give x-value on the x-axis where y-axis is the probability.
        """
        t = (x - self.location) / self.scale
        return 0.5 + (1 / PI) * \
            (t * math.sqrt(1 - (t ** 2)) + math.asin(t))

    def PDF(self, x):
        """
        Partial Distribution Function, which gives the probability for the
        particular value of x, or the area under probability distribution
        from x-h to x+h for continuous distribution.
        """
        return (2 / (self.scale * PI)) * \
                math.sqrt(1 - ((x - self.location) / self.scale) ** 2)

    def inverseCDF(self, probability, start=-10.0, step=0.01):
        """
        It does the reverse of CDF() method, it takes a probability value
        and returns the corresponding value on the x-axis.
        """
        if start < -1 * self.scale:
            start = -1 * self.scale
        cprob = self.CDF(start)
        if probability < cprob: return (start, cprob)
        while (probability > cprob):
            start = start + step
            cprob = self.CDF(start)
            # print start, cprob
        return (start, cprob)

    def mean(self):
        """Gives the arithmetic mean of the sample."""
        return self.location
    def mode(self):
        """Gives the mode of the sample."""
        return self.location

    def kurtosis(self):
        """Gives the kurtosis of the sample."""
        return -1.0

    def skew(self):
        """Gives the skew of the sample."""
        return 0.0

    def variance(self):
        """Gives the variance of the sample."""
        return 0.25 * (self.scale ** 2)

    def quantile1(self):
        """Gives the 1st quantile of the sample."""
        return self.location - (0.404 * self.scale)

    def quantile3(self):
        """Gives the 3rd quantile of the sample."""
        return self.location + (0.404 * self.scale)

    def qmean(self):
        """Gives the quantile of the arithmetic mean of the sample."""
        return 0.5

    def qmode(self):
        """Gives the quantile of the mode of the sample."""
        return 0.5


class TriangularDistribution(Distribution):
    """
    Class for Triangular Distribution.

    @status: Tested method
    @since: version 0.4
    """
    def __init__(self, upper_limit, peak, lower_limit=0):
        """
        Constructor method. The parameters are used to construct the
        probability distribution.

        @param upper_limit: upper limit of the distrbution
        @type upper_limit: float
        @param peak: peak of the distrbution, which has to be between
        the lower and upper limits of the distribution
        @type peak: float
        @param lower_limit: lower limit of the distrbution,
        default = 0
        @type lower_limit: float"""
        self.lower_limit = lower_limit
        if upper_limit < self.lower_limit:
            raise AttributeError
        else:
            self.upper_limit = upper_limit
        if peak > upper_limit:
            raise AttributeError
        if peak < lower_limit + 0.001:
            raise AttributeError
        else:
            self.mode = peak

    def CDF(self, x):
        """
        Cumulative Distribution Function, which gives the cumulative
        probability (area under the probability curve) from -infinity or
        0 to a give x-value on the x-axis where y-axis is the probability.
        """
        if x < self.lower_limit:
            raise AttributeError
        if x > self.mode:
            raise AttributeError
        else:
            return (( x - self.lower_limit) ** 2) / \
                ((self.upper_limit - self.lower_limit) * \
                 (self.mode - self.lower_limit))

    def PDF(self, x):
        """
        Partial Distribution Function, which gives the probability for the
        particular value of x, or the area under probability distribution
        from x-h to x+h for continuous distribution."""
        if x < self.lower_limit:
            raise AttributeError
        if x > self.mode:
            raise AttributeError
        else:
            return ((2 * (x - self.lower_limit)) / \
                    ((self.upper_limit - self.lower_limit) * \
                     (self.mode - self.lower_limit)))

    def inverseCDF(self, probability, start=0, step=0.01):
        """
        It does the reverse of CDF() method, it takes a probability value
        and returns the corresponding value on the x-axis."""
        start = self.lower_limit
        cprob = self.CDF(start)
        if probability < cprob: return (start, cprob)
        while (probability > cprob):
            start = start + step
            cprob = self.CDF(start)
            # print start, cprob
        return (start, cprob)

    def mean(self):
        """Gives the arithmetic mean of the sample."""
        return (float(self.lower_limit +
                self.upper_limit + self.mode) / 3)

    def mode(self):
        """Gives the mode of the sample."""
        return (self.mode)

    def kurtosis(self):
        """Gives the kurtosis of the sample."""
        return ((-3)*(5 ** - 1))

    def skew(self):
        """Gives the skew of the sample."""
        return (math.sqrt(2) * \
                (self.lower_limit + self.upper_limit - 2 * self.mode) * \
                (2 * self.lower_limit - self.self.upper_limit - \
                 self.mode) * (self.lower_limit - 2 * \
                self.upper_limit + self.mode)) / \
                (self.lower_limit ** 2 + self.upper_limit ** 2 + \
                 self.mode ** 2 - self.lower_limit * self.upper_limit + \
                 self.mode ** 2 - self.lower_limit * \
                 (self.upper_limit - self.mode))

    def variance(self):
        """Gives the variance of the sample."""
        return (self.lower_limit ** 2 + self.upper_limit ** 2 + \
                self.mode ** 2 - \
                (self.lower_limit * self.upper_limit) - \
                (self.lower_limit * self.mode) - \
                (self.upper_limit * self.mode)) * (18 ** -1)

    def quantile1(self):
        """Gives the 1st quantile of the sample."""
        if ((self.mode - self.lower_limit) * \
        (self.upper_limit - self.lower_limit) ** -1) > 0.25:
            return self.lower_limit + \
                (0.5 * math.sqrt((self.upper_limit - \
                self.lower_limit) * (self.mode - self.lower_limit)))
        else:
            return self.upper_limit - \
                ((0.5) * math.sqrt (3 * (self.upper_limit -\
                self.lower_limit) * (self.upper_limit - self.mode)))

    def quantile3(self):
        """Gives the 3rd quantile of the sample."""
        if ((self.mode - self.lower_limit) * \
        (self.upper_limit - self.lower_limit) ** -1) > 0.75:
            return self.lower_limit + \
                (0.5 * math.sqrt(3 * (self.upper_limit - \
                self.lower_limit) * (self.mode - self.lower_limit)))
        else:
            return self.upper_limit - \
                ((0.5) * math.sqrt ((self.upper_limit -\
                self.lower_limit) * (self.upper_limit - self.mode)))

    def qmean(self):
        """Gives the quantile of the arithmetic mean of the sample."""
        if self.mode > ((self.lower_limit + self.upper_limit) * 0.5):
            return ((self.upper_limit + self.mode - 2 * \
                     self.lower_limit) ** 2) * (9 * \
                    (self.upper_limit - self.lower_limit) * \
                    (self.mode - self.lower_limit))
        else:
            return (self.lower_limit ** 2 + (5 * self.lower_limit * \
                    self.upper_limit) - (5 * (self.upper_limit ** 2)) - \
                    (7 * self.lower_limit * self.mode) + \
                    (5 * self. upper_limit * self.mode) + self.mode ** 2)

    def qmode(self):
        """Gives the quantile of the mode of the sample."""
        return (self.mode - self.lower_limit) * (self.upper_limit \
        - self.lower_limit) ** - 1
```

```
class WeiBullDistribution(Distribution):
    """
    Class for Weibull distribution.

    @status: Tested method
    @since: version 0.4
    """
    def __init__(self, location=1.0, scale=1.0):
        """Constructor method. The parameters are used to construct the
        probability distribution.

        @param location: position of the distribution, default = 1.0
        @param scale: shape of the distribution, default = 1.0"""
        self.location = location
        self.scale = scale

    def CDF(self, x):
        """
        Cumulative Distribution Function, which gives the cumulative
        probability (area under the  probability curve) from -infinity
        or 0 to a give x-value on the x-axis where y-axis is the
        probability.
        """
        power = -1 * ((float(x) / self.location) ** self.scale)
        return 1 - (math.e ** power)

    def PDF(self, x):
        """
        Partial Distribution Function, which gives the probability for the
        particular value of x, or the area under probability distribution
        from x-h to x+h for continuous distribution.
        """
        if x < 0:
            return 0
        else:
            power = -1 * ((float(x) / self.location) ** self.scale)
            t3 = math.e ** power
            t2 = (float(x) / self.location) ** (self.scale - 1)
            t1 = self.scale / self.location
            return t1 * t2 * t3

    def inverseCDF(self, probability, start=0.0, step=0.01):
        """
        It does the reverse of CDF() method, it takes a probability value
        and returns the corresponding value on the x-axis.
        """
        cprob = self.CDF(start)
        if probability < cprob: return (start, cprob)
        while (probability > cprob):
            start = start + step
            cprob = self.CDF(start)
            # print start, cprob
        return (start, cprob)

    def median(self):
        """Gives the median of the sample."""
        return self.location * \
            (math.log(2, math.e) ** (1/float(self.scale)))

    def mode(self):
        """Gives the mode of the sample."""
        if self.scale > 1:
            t = ((self.scale - 1) / float(self.scale))
            return self.location * (t ** (1/float(self.scale)))
        else:
            return 0

    def random(self):
        """Gives a random number based on the distribution."""
        return random.weibullvariate(self.scale, self.shape)


def FrechetDistribution(**parameters):
    """
    Frechet distribution is an alias of Weibull distribution."""
    return WeibullDistribution(**parameters)
```

File: t_copadsIII.py

```
import sys
import os
import unittest

import copadsIII as N


class testCauchy(unittest.TestCase):
    def testCDF1(self):
        p = N.CauchyDistribution(location = 0.0, scale = 1.0).CDF(0.0)
        self.assertAlmostEqual(p, 0.5, places=4)
    def testCDF2(self):
        p = N.CauchyDistribution(location = 0.0, scale = 1.0).CDF(1.0)
        self.assertAlmostEqual(p, 0.75, places=4)
    def testCDF3(self):
        p = N.CauchyDistribution(location = 0.0, scale = 1.0).CDF(2.0)
        self.assertAlmostEqual(p, 0.85241, places=4)
    def testPDF1(self):
        p = N.CauchyDistribution(location = 0.0, scale = 1.0).PDF(0.0)
        self.assertAlmostEqual(p, 0.31830, places=4)
    def testPDF2(self):
        p = N.CauchyDistribution(location = 0.0, scale = 1.0).PDF(1.0)
        self.assertAlmostEqual(p, 0.15915, places=4)
    def testPDF3(self):
        p = N.CauchyDistribution(location = 0.0, scale = 1.0).PDF(2.0)
        self.assertAlmostEqual(p, 0.06366, places=4)
    def testinverseCDF1(self):
        p = N.CauchyDistribution(location = 0.0,
                                 scale = 1.0).inverseCDF(0.5)[0]
        self.assertAlmostEqual(p, 0)
    def testinverseCDF2(self):
        p = N.CauchyDistribution(location = 0.0,
                                 scale = 1.0).inverseCDF(0.75)[0]
        self.assertAlmostEqual(p, 1.0)
    def testinverseCDF3(self):
        p = N.CauchyDistribution(location = 0.0,
                                 scale = 1.0).inverseCDF(0.8524163)[0]
        self.assertAlmostEqual(p, 2.0)


class testCosine(unittest.TestCase):
    def testCDF1(self):
        p = N.CosineDistribution(location = 0.0, scale = 1.0).CDF(0.0)
        self.assertAlmostEqual(p, 0.5, places=2)
    def testCDF2(self):
        p = N.CosineDistribution(location = 0.0, scale = 1.0).CDF(1.0)
        self.assertAlmostEqual(p, 0.793079, places=4)
    def testCDF3(self):
        p = N.CosineDistribution(location = 0.0, scale = 1.0).CDF(10.0)
        self.assertAlmostEqual(p, 2.00496578, places=4)
    def testinverseCDF1(self):
        p = N.CosineDistribution(location = 0.0,
                                 scale = 1.0).inverseCDF(2.00496578)[0]
        self.assertAlmostEqual(p, 10.0, places=2)
    def testinverseCDF2(self):
        p =  N.CosineDistribution(location = 0.0,
                                  scale = 1.0).inverseCDF(0.5)[0]
        self.assertAlmostEqual(p, 0.0, places=2)
    def testPDF1(self):
        p = N.CosineDistribution(location = 0.0, scale = 1.0).PDF(0.0)
        self.assertAlmostEqual(p, 0.318309, places=4)
    def testPDF2(self):
        p = N.CosineDistribution(location = 0.0, scale = 1.0).PDF(1.0)
        self.assertAlmostEqual(p, 0.245147, places=4)
    def testPDF3(self):
        p = N.CosineDistribution(location = 0.0, scale = 1.0).PDF(10.0)
        self.assertAlmostEqual(p, 0.02561256, places=4)
    def testVariance(self):
        p = N.CosineDistribution(location = 0.0, scale = 1.0).variance()
        self.assertAlmostEqual(p, 1.289868, places=4)


class testExponential(unittest.TestCase):
    def testCDF1(self):
        p = N.ExponentialDistribution(location = 0.0,
                                      scale = 1.0).CDF(0.0)
        self.assertAlmostEqual(p, 0.0, places = 2)
    def testCDF2(self):
        p = N.ExponentialDistribution(location = 0.0,
                                      scale = 1.0).CDF(2.0)
        self.assertAlmostEqual(p, 0.86466, places = 4)
    def testCDF3(self):
        p = N.ExponentialDistribution(location = 1.0,
                                      scale = 1.0).CDF(0.0)
        self.assertAlmostEqual(p, - 1.7182818, places = 4)
    def testPDF1(self):
        p = N.ExponentialDistribution(location = 0.0,
                                      scale = 1.0).PDF(0.0)
        self.assertAlmostEqual(p, 1.0, places = 4)
    def testPDF2(self):
        p = N.ExponentialDistribution(location = 0.0,
                                      scale = 1.0).PDF(1.0)
        self.assertAlmostEqual(p, 0.3679, places = 4)
    def testvariance(self):
        p = N.ExponentialDistribution(location = 0.0,
                                      scale = 1.0).variance()
        self.assertAlmostEqual(p, 1.0, places = 4)
    def testmean(self):
        p = N.ExponentialDistribution(location = 0.0, scale = 1.0).mean()
        self.assertAlmostEqual(p, 1.0, places = 4)
    def testmedian(self):
        p = N.ExponentialDistribution(location = 0.0,
                                          scale = 1.0).median()
        self.assertAlmostEqual(p, 0.30103, places = 4)


class testHypergeometric(unittest.TestCase):
    def testPDF(self):
        p = N.HypergeometricDistribution(sample_size=10,
                                    population_size=100,
                                    population_success=50).PDF(5)
        self.assertAlmostEqual(p, 0.259333,  places = 2)
    def testCDF(self):
        p = N.HypergeometricDistribution(sample_size=10,
                                    population_size=100,
                                    population_success=50).CDF(5)
        self.assertAlmostEqual(p, 0.629073,  places = 2)
    def testinverseCDF(self):
        p = N.HypergeometricDistribution(sample_size=10,
                                population_size=100,
                                population_success=50).inverseCDF(0.629073)[0]
        self.assertAlmostEqual(p, 5,  places = 2)
    def testmean(self):
        p = N.HypergeometricDistribution(sample_size=10,
                                    population_size=100,
                                    population_success=50).mean()
        self.assertAlmostEqual(p, 5.000,  places = 2)
    def testmode(self):
        p = N.HypergeometricDistribution(sample_size=10,
                                    population_size=100,
                                    population_success=50).mode()
        self.assertAlmostEqual(p, 5.500,  places = 2)
    def testvariance(self):
        p = N.HypergeometricDistribution(sample_size=10,
                                    population_size=100,
                                    population_success=50).variance()
        self.assertAlmostEqual(p, 2.272727,  places = 2)


class testLogarithmic(unittest.TestCase):
    def testPDF(self):
        p = N.LogarithmicDistribution(shape=0.45).PDF(1.0)
        self.assertAlmostEqual(p, 1.7332, places=4)
    def testCDF(self):
        p = N.LogarithmicDistribution(shape=0.45).CDF(0.0)
        self.assertAlmostEqual(p, 0.0, places=2)
    def testinverseCDF(self):
        p = N.LogarithmicDistribution(shape=0.45).CDF(0.0)
        self.assertAlmostEqual(p,0.0, places=2)
    def testmode(self):
        p = N.LogarithmicDistribution(shape=0.45).mode()
        self.assertAlmostEqual(p, 1.0, places=2)
    def testmean(self):
        p = N.LogarithmicDistribution(shape=0.45).mean()
        self.assertAlmostEqual(p, 3.15124, places=4)


class testSemicircular(unittest.TestCase):
    def testPDF1(self):
        p = N.SemicircularDistribution(location=0.0, scale=1.0).PDF(0.0)
        self.assertAlmostEqual(p, 0.63662, places=4)
    def testPDF2(self):
        p = N.SemicircularDistribution(location=0.0, scale=1.0).PDF(0.5)
        self.assertAlmostEqual(p, 0.55133, places=4)
    def testCDF1(self):
        p = N.SemicircularDistribution(location=0.0, scale=1.0).CDF(0.0)
        self.assertAlmostEqual(p, 0.5, places=4)
    def testCDF2(self):
        p = N.SemicircularDistribution(location=0.0, scale=1.0).CDF(0.5)
        self.assertAlmostEqual(p, 0.804498, places = 4)
    def testinverseCDF(self):
        p = N.SemicircularDistribution(location=0.0,
                                       scale=1.0).inverseCDF(0.5)[0]
        self.assertAlmostEqual(p, 0.0, places = 2)


class testTriangular(unittest.TestCase):
    def testCDF1(self):
       p = N.TriangularDistribution(2.0, 1.0).CDF(1.0)
       self.assertAlmostEqual(p, 0.5000, places=4)
    def testCDF2(self):
       p = N.TriangularDistribution(2.0, 1.0).CDF(0.8)
       self.assertAlmostEqual(p, 0.3200, places=4)
    def testCDF3(self):
       p = N.TriangularDistribution(4.0, 2.0, -4).CDF(1.0)
       self.assertAlmostEqual(p, 0.5208, places=4)
    def testPDF1(self):
       p = N.TriangularDistribution(2.0, 1.0).PDF(0.08)
       self.assertAlmostEqual(p, 0.08000, places=4)
    def testPDF2(self):
       p = N.TriangularDistribution(4.0, 2.0, -4).PDF(1.0)
       self.assertAlmostEqual(p, 0.20833, places=4)
    def testkurtosis(self):
       p = N.TriangularDistribution(2.0, 1.0).kurtosis()
       self.assertAlmostEqual(p, -0.6, places=4)


class testWeibull(unittest.TestCase):
    def testCDF1(self):
        p = N.WeiBullDistribution(location=1.0,
                    scale=1.0).CDF(2)
        self.assertAlmostEqual(p, 0.864664, places=5)
    def testCDF2(self):
        p = N.WeiBullDistribution(location=2.0,
                    scale=2.0).CDF(2)
        self.assertAlmostEqual(p, 0.632120, places=5)
    def testPDF1(self):
        p = N.WeiBullDistribution(location=1.0,
                    scale=1.0).PDF(2)
        self.assertAlmostEqual(p, 0.135335, places=5)
    def testinverseCDF1(self):
        p = N.WeiBullDistribution(location=1.0,
                    scale=1.0).inverseCDF(0.864664)[0]
        self.assertAlmostEqual(p, 2.000000, places=5)
    def testinverseCDF2(self):
        p = N.WeiBullDistribution(location=2.0,
                    scale=2.0).inverseCDF(0.632120)[0]
        self.assertAlmostEqual(p, 2.000000, places=5)
    def testmedian(self):
        p = N.WeiBullDistribution(location=2.0,
                    scale=2.0).median()
        self.assertAlmostEqual(p, 1.665109, places=5)
    def testmode(self):
        p = N.WeiBullDistribution(location=2.0,
                        scale=2.0).mode()
        self.assertAlmostEqual(p, 1.414213, places=5)


if __name__ == '__main__':
    unittest.main()
```

3. References

Balakrishnan, N., Xie, Q., and Kundu, D. (2009). Exact inference for a simple step-stress model from the exponential distribution under time constraint. Annals of the Institute of Statistical Mathematics, 61(1): 251-274.

Bermejo, R., Supancic, P., and Danzer, R. (2012). Influence of measurement uncertainties on the determination of the Weibull distribution. Journal of the European Ceramic Society, 32(2): 251-255.

Chen, Q., and Gerlach, RH. (2013). The two-sided Weibull distribution and forecasting financial tail risk. International Journal of Forecasting, 29(4): 527-540.

Darwin, JH. (1960). An ecological distribution akin to Fisher's logarithmic distribution. Biometrics, 16(1): 51-60.

Forbes, C., Evans, M., Hastings, N., and Peacock, B. (2011). Weibull distribution. Statistical Distributions, Fourth Edition: 193-201.

Ghannadpour, SS., Hezarkhani, A., and Eshqi, H. (2013). Average and variance estimation programming in normal logarithmic distribution. Global Journal of Computer Science, 2(1).

Greenwood, John. 2002. The correct and incorrect generation of a cosine distribution of scattered particles for Monte-Carlo modelling of vacuum systems. Vacuum 67.2: 217-222.

Islam, MR., Saidur, R., and Rahim, NA. (2011). Assessment of wind energy potentiality at Kudat and Labuan, Malaysia using Weibull distribution function. Energy, 36(2): 985-992.

Johnson, D. (1997). The triangular distribution as a proxy for the beta distribution in risk analysis. Journal of the Royal Statistical Society: Series D (The Statistician), 46(3): 387-398.

Joo, Y. and Casella, G. (2001). Predictive distributions in risk analysis and estimation for the triangular distribution. Environmetrics, 12(7): 647-658.

Ling, Maurice HT. (2009a). Compendium of distributions, I: Beta, binomial, chi-square, F, gamma, geometric, poisson, Student's t, and uniform. The Python Papers Source Codes 1:4.

Ling, Maurice HT. (2009b). Ten Z-test routines from Gopal Kanji's 100 Statistical Tests. The Python Papers Source Codes 1:5.

Pinder III, JE., Wiener, JG., and Smith, MH. (1978). The Weibull distribution: a new method of summarizing survivorship data. Ecology, 175-179.

Rinne, H. (2010). The Weibull distribution: a handbook. CRC Press.

Sathakathulla, AA., and Murthy, BN. (2012). Single, double and multiple sampling plans: Hypergeometric distribution. Journal of Interdisciplinary Mathematics, 15(4-5): 275-338.

Yang, HL. (2012). Two-warehouse partial backlogging inventory models with three-parameter Weibull distribution deterioration under inflation. International Journal of Production Economics, 138(1): 107-116.