Ten Z-Test Routines from *Gopal Kanji's* 100 Statistical Tests

Maurice HT Ling
School of Chemical and Life Sciences, Singapore Polytechnic, Singapore
Department of Zoology, The University of Melbourne, Australia
mauriceling@acm.org

Abstract
This manuscript presents the implementation and testing of 10 Z-test routines from Gopal Kanji's book entitled "100 Statistical Tests".

1. Introduction

Kanji (2006)'s 100 Statistical Tests is a collection of commonly used statistical tests, ranging from single-sample to multi-samples parametric and non-parametric tests. At the same time, a sample calculation is provided for each test. Written in a "recipe" style, Kanji (2006) serves as a useful desktop reference for statistics practitioners. It is therefore conceivable that Python implementation of these tests will be useful on its own or for inclusion into other Python-based statistical packages, such as SalStat (Salmoni, 2008).

This manuscript presents the implementation (file name: ZTests.py) and testing (file name: testfile.py) of 10 Z-test routines from Kanji (2006). The details and limitations of each test are given as docstrings in the codes below. A statistical test harness is implemented (testfile.test function) to standardize the implementation of each hypothesis testing routines. This harness can be used for both 1-tailed or 2-tailed hypothesis tests as it reports the upper and lower critical value given the confidence level and state whether the calculated test statistic is more than the upper critical value (in the upper critical region) or lower than the lower critical value (in the lower critical region) or not. The only limitation of this test harness is the assumption that the test is symmetrical (upper critical region is equal to lower critical region in terms of probability density).

These Z-tests use the standardized normal distribution where mean is zero and variance is 1 (class NormalDistribution). Briefly, the cumulative density function (CDF) calculated the cumulative probability density from negative infinity to a give x-value using the complementary error function (Press et al., 1989) while the inverse CDF (calculates the x-value given the probability density) uses a loop of 0.01 step increment the x-value from -10.0 to positive infinity to measure the point whereby the cumulative probability density from CDF exceeds the given probability density and reports the x-value at that point.

2. Code Files

File: ZTests.py

```
"""
This file contains the implementation of 10 Z-test routines from Gopal
Kanji's book[1] and published under The Python Papers Source Codes [2].

The implemented statistical tests are:
1. Z-test for a population mean (variance known), test 1 of [1]
2. Z-test for two population means (variances known and equal),
   test 2 of [1]
3. Z-test for two population means (variances known and unequal),
   test 3 of [1]
4. Z-test for a proportion (binomial distribution), test 4 of [1]
5. Z-test for the equality of two proportions (binomial distribution),
   test 5 of [1]
6. Z-test for comparing two counts (Poisson distribution),
   test 6 of [1]
7. Z-test of a correlation coefficient, test 13 of [1]
8. Z-test for two correlation coefficients, test 14 of [1]
9. Z-test for correlated proportions, test 23 of [1]
10. Spearman rank correlation test (paired observations), test 58 of [1]

These codes are licensed under Lesser General Public Licence version 3.

[1] Kanji, Gopal K. 2006. 100 Statistical Tests, 3rd Edition. Sage
Publications.
[2] Ling, MHT. 2009. Ten Z-Test Routines from Gopal Kanji's 100 Statistical
Tests. The Python Papers Source Codes 1:5
"""

from math import exp, sqrt, log

SQRT2 = 1.4142135623730950488016887242096980785696718753769
PI2 = 6.2831853071795864769252867665590057683943387987502


def erfcc(x):
    """
    Complementary error function similar to erfc(x) but with
    fractional error lesser than 1.2e-7.
    @see: Numerical Recipes in Pascal (Recipe 6.2)

    @param x: float number
    @return: float number
    """
    z = abs(x)
    t = 1.0 / (1.0 + 0.5*z)
    r = t * exp(-z*z-1.26551223+t*(1.00002368+t*(0.37409196+
        t*(0.09678418+t*(-0.18628806+t*(0.27886807+
        t*(-1.13520398+t*(1.48851587+t*(-0.82215223+
        t*0.17087277)))))))))
    if (x >= 0.0):
        return r
    else:
        return 2.0 - r


class NormalDistribution:
    """
    Class for standardized Normal distribution (area under the curve = 1)
    """
    def __init__(self):
        self.mean = 0.0
        self.stdev = 1.0
    def CDF(self, x):
        """
        Calculates the cumulative probability from -infinity to x.
        """
        return 1.0 - 0.5 * erfcc(x/SQRT2)
    def PDF(self, x):
        """
        Calculates the density (probability) at x by the formula:
        f(x) = 1/(sqrt(2 pi) sigma) e^-((x^2/(2 sigma^2))
        where mu is the mean of the distribution and sigma the standard
        deviation.

        @param x: probability at x
        """
        return (1/(sqrt(PI2) * self.stdev)) * \
            exp(-(x ** 2/(2 * self.stdev**2)))
    def inverseCDF(self, probability, start = -10.0,
                   end = 10.0, error = 10e-8):
        """
        It does the reverse of CDF() method, it takes a probability value
        and returns the corresponding value on the x-axis, together with
        the cumulative probability.

        @param probability: probability under the curve from -infinity
        @param start: lower boundary of calculation (default = -10)
        @param end: upper boundary of calculation (default = 10)
        @param error: error between the given and calculated probabilities
        (default = 10e-8)
        @return (start, cprob): 'start' is the standard deviation for the
        area under the curve from -infinity to the given 'probability' (+/-
        step). 'cprob' is the calculated area under the curve from
        -infinity to the returned 'start'.
        """
        # check for tolerance
        if abs(self.CDF(start)-probability) < error:
            return (start, self.CDF(start))
        # case 1: lower than -10 standard deviations
        if probability < self.CDF(start):
            return self.inverseCDF(probability, start-5, start, error)
        # case 2: between -10 to 10 standard deviations (bisection method)
        if probability > self.CDF(start) and \
        probability < self.CDF((start+end)/2):
            return self.inverseCDF(probability, start, (start+end)/2,
                                   error)
        if probability > self.CDF((start+end)/2) and \
        probability < self.CDF(end):
            return self.inverseCDF(probability, (start+end)/2, end, error)
        # case 3: higher than 10 standard deviations
        if probability > self.CDF(end):
            return self.inverseCDF(probability, end, end+5, error)


def test(statistic, distribution, confidence):
    """
    Generates the critical value from distribution and confidence value
    using the distribution's inverseCDF method and performs 1-tailed and 2-
    tailed test by comparing the calculated statistic with the critical
    value.

    Returns a 5-element list:
    [left result, left critical, statistic, right critical, right result]

    where
    left result = True (statistic in lower critical region) or
        False (statistic not in lower critical region)
    left critical = lower critical value generated from 1 - confidence
    statistic = calculated statistic value
    right critical = upper critical value generated from confidence
    right result = True (statistic in upper critical region) or
        False (statistic not in upper critical region)

    Therefore, null hypothesis is accepted if left result and right result
    are both False in a 2-tailed test.

    @param statistic: calculated statistic (float)
    @param distribution: distribution to calculate critical value
    @type distribution: instance of a statistics distribution
    @param confidence: confidence level of a one-tail
        test (usually 0.95 or 0.99), use 0.975 or 0.995 for 2-tail test
    @type confidence: float of less than 1.0
    """
    data = [None, None, statistic, None, None]
    data[1] = distribution.inverseCDF(1.0 - confidence)[0]
    if data[1] < statistic:
        data[0] = False
    else:
        data[0] = True
    data[3] = distribution.inverseCDF(confidence)[0]
    if statistic < data[3]:
        data[4] = False
    else:
        data[4] = True
    return data

def Z1Mean1Variance(**kwargs):
    """
    Test 1: Z-test for a population mean (variance known)

    To investigate the significance of the difference between an assumed
    population mean and sample mean when the population variance is
    known.

    Limitations
    - Requires population variance (use Test 7 if population variance
      unknown)

    @param smean: sample mean
    @param pmean: population mean
    @param pvar: population variance
    @param ssize: sample size
    @param confidence: confidence level
    """
    smean = kwargs['smean']
    pmean = kwargs['pmean']
    pvar = kwargs['pvar']
    ssize = kwargs['ssize']
    statistic = abs(smean - pmean)/ \
                (pvar / sqrt(ssize))
    return test(statistic, NormalDistribution(), kwargs['confidence'])

def Z2Mean1Variance(**kwargs):
    """
    Test 2: Z-test for two population means (variances known and equal)

    To investigate the significance of the difference between the means of
    Two samples when the variances are known and equal.

    Limitations:
    1. Population variances must be known and equal (use Test 8 if
       Population variances unknown

    @param smean1: sample mean of sample #1
    @param smean2: sample mean of sample #2
    @param pvar: variances of both populations (variances are equal)
    @param ssize1: sample size of sample #1
    @param ssize2: sample size of sample #2
    @param confidence: confidence level
    @param pmean1: population mean of population #1 (optional)
    @param pmean2: population mean of population #2 (optional)
    """
    if not kwargs.has_key('pmean1'):
        pmean1 = 0.0
    else: pmean1 = kwargs['pmean1']
    if not kwargs.has_key('pmean2'):
        pmean2 = 0.0
    else: pmean2 = kwargs['pmean2']
    smean1 = kwargs['smean1']
    smean2 = kwargs['smean2']
    pvar = kwargs['pvar']
    ssize1 = kwargs['ssize1']
    ssize2 = kwargs['ssize2']
    statistic = ((smean1 - smean2) - (pmean1 - pmean2))/ \
                (pvar * sqrt((1.0 / ssize1) + (1.0 / ssize2)))
    return test(statistic, NormalDistribution(), kwargs['confidence'])

def Z2Mean2Variance(**kwargs):
    """
    Test 3: Z-test for two population means (variances known and unequal)

    To investigate the significance of the difference between the means of
    Two samples when the variances are known and unequal.

    Limitations:
    1. Population variances must be known(use Test 9 if population
       variances unknown

    @param smean1: sample mean of sample #1
    @param smean2: sample mean of sample #2
    @param pvar1: variance of population #1
    @param pvar2: variance of population #2
    @param ssize1: sample size of sample #1
    @param ssize2: sample size of sample #2
    @param confidence: confidence level
    @param pmean1: population mean of population #1 (optional)
    @param pmean2: population mean of population #2 (optional)
    """
    if not kwargs.has_key('pmean1'):
        pmean1 = 0.0
    else: pmean1 = kwargs['pmean1']
    if not kwargs.has_key('pmean2'):
        pmean2 = 0.0
    else: pmean2 = kwargs['pmean2']
    smean1 = kwargs['smean1']
    smean2 = kwargs['smean2']
    pvar1 = kwargs['pvar1']
    pvar2 = kwargs['pvar2']
    ssize1 = kwargs['ssize1']
    ssize2 = kwargs['ssize2']
    statistic = ((smean1 - smean2) - (pmean1 - pmean2))/ \
                sqrt((pvar1 / ssize1) + (pvar2 / ssize2))
    return test(statistic, NormalDistribution(), kwargs['confidence'])

def Z1Proportion(**kwargs):
    """
    Test 4: Z-test for a proportion (binomial distribution)

    To investigate the significance of the difference between an assumed
    proportion and an observed proportion.

    Limitations:
    1. Requires sufficiently large sample size to use Normal approximation
       To binomial

    @param spro: sample proportion
    @param ppro: population proportion
    @param ssize: sample size
    @param confidence: confidence level
    """
    spro = kwargs['spro']
    ppro = kwargs['ppro']
    ssize = kwargs['ssize']
    statistic = (abs(ppro - spro) - (1 / (2 * ssize)))/ \
                sqrt((ppro * (1 - spro)) / ssize)
    return test(statistic, NormalDistribution(), kwargs['confidence'])

def Z2Proportion(**kwargs):
    """
    Test 5: Z-test for the equality of two proportions (binomial
    distribution)
    To investigate the assumption that the proportions of elements from two
    populations are equal, based on two samples, one from each population.

    Limitations:
    1. Requires sufficiently large sample size to use Normal approximation
       To binomial

    @param spro1: sample proportion #1
    @param spro2: sample proportion #2
    @param ssize1: sample size #1
    @param ssize2: sample size #2
    @param confidence: confidence level
    """
    spro1 = kwargs['spro1']
    spro2 = kwargs['spro2']
    ssize1 = kwargs['ssize1']
    ssize2 = kwargs['ssize2']
    P = ((spro1 * ssize1) + (spro2 * ssize2)) / (ssize1 + ssize2)
    statistic = (spro1 - spro2) / \
                (P * (1.0 - P) * ((1.0 / ssize1) + (1.0 / ssize2))) ** 0.5
    return test(statistic, NormalDistribution(), kwargs['confidence'])

def Z2Count(**kwargs):
    """
    Test 6: Z-test for comparing two counts (Poisson distribution)

    To investigate the significance of the differences between two counts.

    Limitations:
    1. Requires sufficiently large sample size to use Normal approximation
       To binomial

    @param time1: first measurement time
    @param time2: second measurement time
    @param count1: counts at first measurement time
    @param count2: counts at second measurement time
    @param confidence: confidence level
    """
    time1 = kwargs['time1']
    time2 = kwargs['time2']
    R1 = kwargs['count1'] / float(time1)
    R2 = kwargs['count2'] / float(time2)
    statistic = (R1 - R2) / sqrt((R1 / time1) + (R2 / time2))
    return test(statistic, NormalDistribution(), kwargs['confidence'])

def ZPearsonCorrelation(**kwargs):
    """
    Test 13: Z-test of a correlation coefficient

    To investigate the significance of the difference between a correlation
    coefficient and a specified value.

    Limitations:
    1. Assumes a linear relationship (regression line as Y = MX + C)
    2. Independence of x-values and y-values
    Use Test 59 when these conditions cannot be met

    @param sr: calculated sample Pearson's product-moment correlation
    coefficient
    @param pr: specified Pearson's product-moment correlation coefficient
    to test
    @param ssize: sample size
    @param confidence: confidence level
    """
    ssize = kwargs['ssize']
    sr = kwargs['sr']
    pr = kwargs['pr']
    Z1 = 0.5 * log((1 + sr) / (1 - sr))
    meanZ1 = 0.5 * log((1 + pr) / (1 - pr))
    sigmaZ1 = 1.0 / sqrt(ssize - 3)
    statistic = (Z1 - meanZ1) / sigmaZ1
    return test(statistic, NormalDistribution(), kwargs['confidence'])

def Z2PearsonCorrelation(**kwargs):
    """
    Test 14: Z-test for two correlation coefficients

    To investigate the significance of the difference between the
    Correlation coefficients for a pair variables occurring from two
    difference populations.

    @param r1: Pearson correlation coefficient of sample #1
    @param r2: Pearson correlation coefficient of sample #2
    @param ssize1: Sample size #1
    @param ssize2: Sample size #2
    @param confidence: confidence level
    """
    z1 = 0.5 * log((1.0 + kwargs['r1']) /(1.0 - kwargs['r1']))
    z2 = 0.5 * log((1.0 + kwargs['r2']) /(1.0 - kwargs['r2']))
    sigma1 = 1.0 / sqrt(kwargs['ssize1'] - 3)
    sigma2 = 1.0 / sqrt(kwargs['ssize2'] - 3)
    sigma = sqrt((sigma1 ** 2) + (sigma2 ** 2))
    statistic = abs(z1 - z2) / sigma
    return test(statistic, NormalDistribution(), kwargs['confidence'])

def ZCorrProportion(**kwargs):
    """
    Test 23: Z-test for correlated proportions

    To investigate the significance of the difference between two
    correlated proportions in opinion surveys. It can also be used for more
    general applications.

    Limitations:
    1. The same people are questioned both times (correlated property).
    2. Sample size must be quite large.

    @param ssize: sample size
    @param ny: number answered 'no' in first poll and 'yes' in second poll
    @param yn: number answered 'yes' in first poll and 'no' in second poll
    @param confidence: confidence level
    """
    ssize = kwargs['ssize']
    ny = kwargs['ny']
    yn = kwargs['yn']
    sigma = (ny + yn) - (((ny - yn) ** 2.0) / ssize)
    sigma = sqrt(sigma / (ssize * (ssize - 1.0)))
    statistic = (ny - yn) / (sigma * ssize)
    return test(statistic, NormalDistribution(), kwargs['confidence'])

def SpearmanCorrelation(**kwargs):
    """
    Test 58: Spearman rank correlation test (paired observations)
    To investigate the significance of the correlation between two series
    of observations obtained in pairs.

    Limitations:
    1. Assumes the two population distributions to be continuous
    2. Sample size must be more than 10

    @param R: sum of squared ranks differences
    @param ssize: sample size
    @param series1: ranks of series #1 (not used if R is given)
    @param series2: ranks of series #2 (not used if R is given)
    @param confidence: confidence level
    """
    ssize = kwargs['ssize']
    if not kwargs.has_key('R'):
        series1 = kwargs['series1']
        series2 = kwargs['series2']
        R = [((series1[i] - series2[i]) ** 2) for i in range(len(series1))]
        R = sum(R)
    else:
        R = kwargs['R']
    statistic = (6.0 * R) - (ssize * ((ssize ** 2) - 1.0))
    statistic = statistic / (ssize * (ssize + 1.0) * sqrt(ssize - 1.0))
    return test(statistic, NormalDistribution(), kwargs['confidence'])
```

File: testfile.py

```
"""
This file contains the test codes for the implementation of 10 Z-test
routines from Gopal Kanji's book[1] and published under The Python Papers
Source Codes [2].

These codes are licensed under Lesser General Public Licence version 3.

[1] Kanji, Gopal K. 2006. 100 Statistical Tests, 3rd Edition. Sage
Publications.
[2] Ling, MHT. 2009. Ten Z-Test Routines from Gopal Kanji's 100 Statistical
Tests. The Python Papers Source Codes 1:5
"""

import unittest
import ZTests as N


class testNormalDistribution(unittest.TestCase):

    def testZ1Mean1Variance(self):
        """
        Test 1: Z-test for a population mean (variance known)
        """
        self.assertAlmostEqual(N.Z1Mean1Variance(pmean = 4.0, smean = 4.6,
            pvar = 1.0, ssize = 9, confidence = 0.975)[2], 1.8)
        self.assertFalse(N.Z1Mean1Variance(pmean = 4.0, smean = 4.6,
            pvar = 1.0, ssize = 9, confidence = 0.975)[4])

    def testZ2Mean1Variance(self):
        """
        Test 2: Z-test for two population means (variances known and equal)
        """
        self.assertAlmostEqual(N.Z2Mean1Variance(smean1 = 1.2,
            smean2 = 1.7, pvar = 1.4405, ssize1 = 9, ssize2 = 16,
            confidence = 0.975)[2],
            -0.83304408)
        self.assertFalse(N.Z2Mean1Variance(smean1 = 1.2, smean2 = 1.7,
            pvar = 1.4405, ssize1 = 9, ssize2 = 16, confidence = 0.975)[4])

    def testZ2Mean2Variance(self):
        """
        Test 3: Z-test for two population means (variances known and
        unequal)
        """
        self.assertAlmostEqual(N.Z2Mean2Variance(smean1 = 80.02,
            smean2 = 79.98, pvar1 = 0.000576, pvar2 = 0.001089,
            ssize1 = 13, ssize2 = 8, confidence = 0.975)[2],
            2.977847)
        self.assertTrue(N.Z2Mean2Variance(smean1 = 80.02,
            smean2 = 79.98, pvar1 = 0.000576, pvar2 = 0.001089,
            ssize1 = 13, ssize2 = 8, confidence = 0.975)[4])

    def testZ1Proportion(self):
        """
        Test 4: Z-test for a proportion (binomial distribution)
        """
        self.assertAlmostEqual(N.Z1Proportion(spro = 0.5, ppro = 0.4,
            ssize = 100, confidence = 0.975)[2], 2.23606798)
        self.assertTrue(N.Z1Proportion(spro = 0.5, ppro = 0.4,
            ssize = 100, confidence = 0.975)[4])

    def testZ2Proportion(self):
        """
        Test 5: Z-test for the equality of two proportions (binomial
        distribution)
        """
        self.assertAlmostEqual(N.Z2Proportion(spro1 = 0.00325,
            spro2 = 0.0573, ssize1 = 952, ssize2 = 1168,
            confidence = 0.025)[2], -6.9265418)
        self.assertFalse(N.Z2Proportion(spro1 = 0.00325, spro2 = 0.0573,
            ssize1 = 952, ssize2 = 1168, confidence = 0.025)[4])

    def testZ2Count(self):
        """
        Test 6: Z-test for comparing two counts (Poisson
        distribution)
        """
        self.assertAlmostEqual(N.Z2Count(time1 = 22, time2 = 30,
            count1 = 952, count2 = 1168, confidence = 0.975)[2],
            2.401630072)
        self.assertTrue(N.Z2Count(time1 = 22, time2 = 30, count1 = 952,
            count2 = 1168, confidence = 0.975)[4])

    def testZPearsonCorrelation(self):
        """
        Test 13: Z-test of a correlation coefficient
        """
        self.assertAlmostEqual(N.ZPearsonCorrelation(sr = 0.75, pr = 0.5,
            ssize = 24, confidence = 0.95)[2], 1.94140329)
        self.assertTrue(N.ZPearsonCorrelation(sr = 0.75, pr = 0.5,
            ssize = 24, confidence = 0.95)[4])

    def testZ2PearsonCorrelation(self):
        """
        Test 14: Z-test for two correlation coefficients
        """
        self.assertAlmostEqual(N.Z2PearsonCorrelation(r1 = 0.5, r2 = 0.3,
            ssize1 = 28, ssize2 = 35, confidence = 0.975)[2], 0.89832268)
        self.assertFalse(N.Z2PearsonCorrelation(r1 = 0.5, r2 = 0.3,
            ssize1 = 28, ssize2 = 35, confidence = 0.975)[4])

    def testZCorrProportion(self):
        """
        Test 23: Z-test for correlated proportions
        """
        self.assertAlmostEqual(N.ZCorrProportion(ny = 15, yn = 9,
            ssize = 105, confidence = 0.975)[2], 1.22769962)
        self.assertFalse(N.ZCorrProportion(ny = 15, yn = 9,
            ssize = 105, confidence = 0.975)[4])
    def testSpearmanCorrelation(self):
        """
        Test 58: Spearman rank correlation test (paired observations)
        """
        self.assertAlmostEqual(N.SpearmanCorrelation(R = 24, ssize = 11,
            confidence = 0.975)[2], -2.8173019)
        self.assertFalse(N.SpearmanCorrelation(R = 24, ssize = 11,
            confidence = 0.975)[4])

class testNormal(unittest.TestCase):
    def testCDF1(self):
        self.assertAlmostEqual(N.NormalDistribution().CDF(0), 0.5)
    def testPDF1(self):
        self.assertAlmostEqual(N.NormalDistribution().PDF(0), 0.3989423)
    def testinverseCDF1(self):
        self.assertAlmostEqual(N.NormalDistribution().inverseCDF(0.5)[0],
                               0)
class testerfcc(unittest.TestCase):
    """
    Test data from Press, William H., Flannery, Brian P., Teukolsky, Saul
    A., and Vetterling, William T. 1992. Numerical Recipes Example Book C,
    2nd edition. Cambridge University Press.
    """
    def test1(self):
        self.assertAlmostEqual(N.erfcc(0.0), 1-0.000000)
    def test2(self):
        self.assertAlmostEqual(N.erfcc(0.5), 1-0.5204999)
    def test3(self):
        self.assertAlmostEqual(N.erfcc(1.0), 1-0.8427008)
    def test4(self):
        self.assertAlmostEqual(N.erfcc(1.5), 1-0.9661051)

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

3. References

Kanji, Gopal K. 2006. 100 Statistical Tests, 3rd edition. Sage Publications.

Press, William H., Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T. 1989. Numerical Recipes in Pascal. Cambridge University Press, Cambridge.

Press, William H., Flannery, Brian P., Teukolsky, Saul A., and Vetterling, William T. 1992. Numerical Recipes Example Book C. Cambridge University Press, Cambridge.

Salmoni, A. 2008. Embedding a Python interpreter into your program. The Python Papers, 3(2): 3.