Testing statistical properties

John Hughes

QuviQ
\[ 10^{\theta} \]
The Realm of Statistics

Make statements about probabilities

...with a risk of being wrong
100 Coin Tosses

Probability

0,00E+00
1,00E-02
2,00E-02
3,00E-02
4,00E-02
5,00E-02
6,00E-02
7,00E-02
8,00E-02
9,00E-02
0
6
12
18
24
30
36
42
48
54
60
66
72
78
84
90
96
200 Coin Tosses
1000 Coin Tosses
100 Coin Tosses

Confidence level

Probability

p-value
"Test by contradiction"

- To demonstrate $P(\text{tails}) < 50\%$...
  
  - Assume the opposite ($P(\text{tails}) \geq 50\%$)
  
  - Compute the probability of observed results or worse
  
  - If it’s < threshold, *reject the null hypothesis*
  
- Assert $P(\text{tails}) < 50\%$, at *confidence level* $1 -$threshold
What confidence level do we need?

- Particle physicists: 99.99994%
- Psychologists: 95%
- Software developers?
How often is it ok for a test to fail when there is no bug?
My friend Azamat is very good developer, he is always have all unit test green. If unit test is fail, it is remove. Is best practice.

8:35 AM - 5 May 2011

256 Retweets  25 Likes
How often is it ok for a test to fail when there is no bug?

Never in the lifetime of the project!

$10^{-6}$? $10^{-9}$?
Two special characteristics:

• We constantly re-run tests

• We can easily get more data
Testing the Bool generator

import Statistics.Distribution
import Statistics.Distribution.Binomial

cumulative (binomial 100 0.5) 40 < threshold
threshold = 0.000000001

```haskell
rejectAtLeast :: Double -> [Bool] -> Bool
rejectAtLeast p bs =
  cumulative (binomial (length bs) p) k < threshold
  where k = fromIntegral (length (filter id bs))
```

A Bool is True with probability at least p

```haskell
prop_BoolAtLeast p bs =
  not (rejectAtLeast p bs)
```

*Stat*> quickCheck$ prop_BoolAtLeast 0.8
*** Failed! Falsifiable (after 65 tests and 7 shrinks):
[False,False,False,False,False,False,False,False,False,False,False,False,False,False]

Shortest list of False values in the list that enables us to reject prob >= 80%
quickCheck.withMaxSuccess 10000$ prop_BoolAtLeast 0.7
*** Failed! Falsifiable (after 898 tests and 9 shrinks):
[False,False,False,False,False,False,False,False,False,False,False,False,False,False,False,False,False,False,False,False,False]

good

quickCheck.withMaxSuccess 10000$ prop_BoolAtLeast 0.6
+++ OK, passed 10000 tests.
Generate an *infinite* list of samples

```haskell
prop_BoolAtLeastInf p (InfiniteList bs _) = not (rejectAtLeastInf p bs)

rejectAtLeastInf p bs =
  or [rejectAtLeast p pbs
      | pbs <- prefixes bs]

prefixes bs =
  [take n bs | n <- iterate (2*) 100]
```
quickCheck $ prop_BoolAtLeastInf 0.55

*** Failed! Falsifiable (after 1 test and 81 shrinks):
[False,False,False,False,False,False,False,False,False,False,False,Fals
se,False,False,False,False,False,False,False,False,False,False,False,Fa
lse,False,False,False,False,False,False,False,False,False,False,False,
False,False,False,False,False,False,False,False,False,False,False,False
False,False,False,False,False,False,False,False,False,False,False,False] ++ ...

Should terminate
for any p>0.5
Two-sided test

• I **reject** the null hypothesis if there are **too few** True values in the sequence
  • so I know actual probability $< p$

• **When should I accept** the null hypothesis?
  • When I know the actual probability $> p$?

• **What if the actual probability == p?**
We’ll be able to say actual probability is definitely $>p'$, or $<p$ (or possibly both)
checkProbability $ p' \ p \ bs$
  | rejectAtLeast $ p \ bs$ = Just False
  | rejectAtLeast $ (1- p') \ (\map \ not \ bs)$ = Just True
  | otherwise = Nothing

Probability is <\(p\)

checkProbabilityInf $ p' \ p \ bs =$
  fromJust $ \head \ \filter \ (/=Nothing) \ \map \ (checkProbability \ p' \ p) \ \prefixes \ bs$

Probability is >\(p'\)
A property to test booleans

More convenient to pass a tolerance

```
prop_CheckProbability tol p (Blind (Fixed (InfiniteList bs _))) =
checkProbabilityInf (p*tol) p bs
```

Don’t print or shrink the infinite list!
*Main> quickCheck $ prop_CheckProbability 0.9 0.4
+++ OK, passed 100 tests:
65%  800
31%  400
 4%  1600

Instrumented to show how many booleans were needed

*Main> quickCheck $ prop_CheckProbability 0.9 0.6
*** Failed! Falsifiable (after 1 test):
(*)
800

*Main> quickCheck $ prop_CheckProbability 0.9 0.5
+++ OK, passed 100 tests:
64%  6400
33%  3200
 2%  1600
 1%  800
*Main> quickCheck . checkCoverage $ \b -> cover 50 b "True" True
+++ OK, passed 6400 tests (49.11% True).

*Main> quickCheck . checkCoverage $ \b -> cover 50 b "True" True
+++ OK, passed 3200 tests (50.62% True).

*Main> quickCheck . checkCoverage $ \b -> cover 50 b "True" True
+++ OK, passed 6400 tests (50.00% True).

*Main> quickCheck . checkCoverage $ \b -> cover 50 b "True" True
+++ OK, passed 6400 tests (49.88% True).

64%  6400
33%  3200
 2%  1600
 1%   800
checkCoverage =
    checkCoverageWith
        (Confidence{confidence = 1000000000,
                    tolerance = 0.9})
Does it make sense to *repeat* statistical tests?

• Every time there is a risk of a wrong answer

<table>
<thead>
<tr>
<th>10 tests</th>
<th>1 test</th>
</tr>
</thead>
<tbody>
<tr>
<td>X 1000 samples</td>
<td>X 10000 samples</td>
</tr>
</tbody>
</table>

• Worth repeating after a code change
• Worth varying *other* inputs than the samples
Testing *frequency*

• Need to generate *weights* and samples

• There may be a mistake in the interpretation of weights

• Test that each choice is made in proportion to its weight
prop_Frequency :: (NonEmptyList (Positive Int)) -> _
prop_Frequency (NonEmpty ws') =
  forAll (Blind <$>
    infiniteListOf
      (frequency (zip ws (map return [0..])))) $ 
    (Blind ns) ->
      all (\(w,i) ->
        let p = (fromIntegral w/fromIntegral total) in
          checkProbabilityInf (0.9*p) p (map (==i)  ns))
    (zip ws [0..])
  where ws = map getPositive ws'
  total = sum ws
prop_Frequency :: (NonEmptyList (Positive Int)) -> _
prop_Frequency (NonEmpty ws') =
    forAll (Blind <$>)
        infiniteListOf
            (frequency (zip ws (map return [0..]))) $ 
    (Blind ns) ->
    all (\(w,i) ->
        let p = (fromIntegral w/fromIntegral total) in
            checkProbabilityInf (0.9*p) p (map (==i)  ns))
        (zip ws [0..])

where ws = map getPositive ws'
    total = sum ws
prop_Frequency :: (NonEmptyList (Positive Int)) -> _
prop_Frequency (NonEmpty ws') =
  forAll (Blind <$>)
    infiniteListOf
      (frequency (zip ws (map return [0..])))$ 
    (Blind ns) ->
      all (\(w,i) ->
        let p = (fromIntegral w/fromIntegral total) in
          checkProbabilityInf (0.9*p) p (map (==i) ns))
    (zip ws [0..])
where ws = map getPositive ws'
  total = sum ws
prop_Frequency :: (NonEmptyList (Positive Int)) -> _
prop_Frequency (NonEmpty ws') =
  forAll (Blind <$>
          infiniteListOf
            (frequency (zip ws (map return [0..])))) $
  (Blind ns) ->
    all (\(w,i) ->
        let p = (fromIntegral w/fromIntegral total) in
        checkProbabilityInf (0.9*p) p (map (==i)  ns))
  (zip ws [0..])
where ws = map getPositive ws'
    total = sum ws
prop_Frequency :: (NonEmptyList (Positive Int)) -> _
prop_Frequency (NonEmpty ws') =
  forAll (Blind <$>
    infiniteListOf
      (frequency (zip ws (map return [0..])))) $ 
  \(Blind ns) ->
    all (\(w,i) ->
      let p = (fromIntegral w/fromIntegral total) in
        checkProbabilityInf (0.9*p) p (map (==i)  ns))
  (zip ws [0..])
where ws = map getPositive ws'
  total = sum ws
Failed: NonEmpty \{getNonEmpty = ... [6,4,4,5] (*)

Failed: ... [6,4] (*)

Failed: ... [6,3] (*)

*** Failed! Falsifiable (after 7 tests and 2 shrinks):
NonEmpty \{getNonEmpty = ... [6,3] (*)

Counterexample: contains 6 and another value
prop_Frequency :: (NonEmptyList (Positive Int)) -> _

prop_Frequency (NonEmpty ws') =
  forall (Blind <$>)
    infiniteListOf
      (frequency (zip ws (map return [0..])))))) $ 
  (Blind ns) ->
    all (\(w,i) ->
      let p = (fromIntegral w/fromIntegral total) in
        checkProbabilityInf (0.9*p) p (map (==i)  ns))
    (zip ws [0..])
  where ws = map getPositive ws'
    total = sum ws

Sloppy tolerance ➔
non-determinism
What can we do?

• Change tolerance to 0.99

  ➔ **Much** slower tests

  ➔ **Much** less non-determinism
Failed:
[2,5,6,4,5,7]
(*)

Failed:
[4,5,7]
(*)

Failed:
[5,7]
(*)

Failed:
[3,7]
(*)

Failed:
[2,7]
(*)

Failed:
[1,7]
(*)

Failed:
[1,6]
(*)

*** Failed! Falsifiable (after 8 tests and 6shrinks):
[1,6]
(*)
Another planted bug: map (+1)

Failed:
[1,3]
(*)

Failed:
[1,2]
(*)

*** Failed! Falsifiable (after 1 test and 1 shrink):
[1,2]
(*)
Lessons

• Statistical properties need a *tolerance* for error, and a *certainty threshold* (e.g. $10^{-9}$ probability of error)

• Use *infinite* lists of samples; keep sampling until certainty is attained

• Avoid *too many* statistical tests—each may be wrong

• Use a *tight* tolerance to get good shrinking
  • (maybe only during shrinking?)
Heads up!

• There are many more statistical tests, suitable for different problems

• Pearson’s Chi\(^2\) test
  • rejects the hypothesis ”samples were drawn from this particular finite distribution”
  • i.e. perfect for testing frequency, FTS, etc
  • (but when do we accept the samples?)

Andrea Arcuri\(^1\) and Lionel Briand\(^2\)

\(^1\) Simula Research Laboratory, P.O. Box 134, Lysaker, Norway. Email: arcuri@simula.no
\(^2\) SnT Centre, University of Luxembourg, 6 rue Richard Coudenhove-Kalergi, L-1359, Luxembourg. Email: lionel.briand@uni.lu

Abstract

Randomized algorithms are widely used to address many types of software engineering problems, especially in the area of software verification and validation with a strong emphasis on test automation. However, randomized algorithms are affected by chance, and so require the use of appropriate statistical tests to be asserted or adapted in a correct manner. This paper focuses on summarizing and providing guidance on statistical testing relevant to randomized algorithms.
Conclusion

• Use *sound* statistical tests, ...
• ...to test the *actual property of interest*

• Statistical tests are expensive and a bit specialised, but *can work well* in combination with QuickCheck and shrinking