Package ‘SPC’

June 16, 2013

Type Package

Title What the package does (short line)

Version 1.0

Date 2013-06-16

Author Who wrote it

Maintainer Who to complain to `<yourfault@somewhere.net>`

Description More about what it does (maybe more than one line)

License What license is it under?

R topics documented:

`SPC-package` *SPC - Implements a Statistical Process Control (SPC) chart ~~ package title ~~*

Description

Analyzes data for Western Electric SPC (industry standard since 1956) rule violation(s). Plots data to reveal issues or lack thereof.

Details

Package:	SPC
Type:	Package
Version:	1.0
Date:	2013-06-15
License:	Fishing license

Given a process in which results are produced in batchs (i.e., drive shafts, stored procedure invocations, eggs layed), measure the average value and the range of values produced by the batch. A process owner will want to monitor trends in those data over time to determine if the process is in control. Excessive variation will be identified by SPC. Any violation of SPC rules should trigger investigation and possible remedial action.

Author(s)

Rod Doe

Maintainer: <doerodney@gmail.com>

References

'Statistical Process Control' by Donald Wheeler

Examples

```
batchAverages = c(0, 1.1,1.2,1.1,  1.3,1.4,1.3, 1.4,1.5,1.4,   0,  1.1,-1,1.1,1.2,1.1,   0,  2.2,1.1,2.3,
x.bar.batchAverages = 0
sigma.batchAverages = 1
batchRanges = numeric(length(batchAverages))
x.bar.batchRanges = 10
sigma.batchRanges = 1.0
for (i in 1:length(batchRanges)) { batchRanges[i] = x.bar.batchRanges + rnorm(1, 0, 0.5) }
result = plotSpcChart(batchAverages, batchRanges, x.bar.batchAverages, sigma.batchAverages, x.bar.batchRang
(result)
```

`classifyByWesternElectricRuleZones`

classifyByWesternElectricRuleZones

Description

Given a vector of data, creates and returns a vector of Western Electric Rule zones for each point. Western Electric Rules classify data into zones of A (2-3 sigma), B (1-2 sigma), and C (0-1) sigma.

Usage

```
classifyByWesternElectricRuleZones(x, x.bar = NULL, sigma = NULL)
```

Arguments

`x`	a vector of numeric values that are to be classified into Western Electric Rule zones
`x.bar`	the mean value to be used for classification. Is NULL by default. Calculated as necessary.
`sigma`	standard deviation to be used for classification. Is NULL by default. Calculated as necessary.

Details

Western Electric Rules classify data into zones of A (2-3 sigma), B (1-2 sigma), and C (0-1) sigma.

Value

a vector with these values: 'O' = outlier, 'A' = zone A, 'B' = zone B, 'C' = zone C

Note

None

Author(s)

Rod Doe

References

None

See Also

None

Examples

```
## x = c(3.1, 2.1, 1.1, 0.1, 0, -0.1, -1.1, -2.1, -3.1)
## x.bar = 0
## sigma = 1
## xclass = classifyByWesternElectricRuleZones.numeric(x, x.bar, sigma)
## (xclass)
## #[1] "O" "A" "B" "C" "C" "C" "B" "A" "O"

## The function is currently defined as
function (x, x.bar = NULL, sigma = NULL)
{
    zones.numeric = classifyByWesternElectricRuleZones.numeric(x,
        x.bar, sigma)
    zone.A = union(which(zones.numeric == 3), which(zones.numeric ==
        -3))
    zone.B = union(which(zones.numeric == 2), which(zones.numeric ==
        -2))
    zone.C = union(which(zones.numeric == 1), which(zones.numeric ==
        -1))
    zone.C = union(zone.C, which(zones.numeric == 0))
    zone.Outlier = union(which(zones.numeric == 4), which(zones.numeric ==
        -4))
```

```
    zones = character(length(x))
    zones[zone.A] = "A"
    zones[zone.B] = "B"
    zones[zone.C] = "C"
    zones[zone.Outlier] = "O"
    return(zones)
  }
```

`classifyByWesternElectricRuleZones.numeric`
classifyByWesternElectricRuleZones.numeric

Description

Given a vector of data, creates and returns a vector of Western Electric Rule numeric zones for each point. Western Electric Rules classify data into zones of A (2-3 sigma), B (1-2 sigma), and C (0-1) sigma. This function returns numeric classifications 4 (> 3 sigma), 3 (positive class A), 2 (positive class B), 1 (positive class C), and 0 (equals mean). It also returns negative numeric classifications -4 (< 3 sigma), -3 (negative class A), -2 (negative class B), and -1 (negative class C).

Usage

```
classifyByWesternElectricRuleZones.numeric(x, x.bar = NULL, sigma = NULL)
```

Arguments

`x`	a vector of numeric values that are to be classified into Western Electric Rule zones
`x.bar`	the mean value to be used for classification. Is NULL by default. Calculated as necessary.
`sigma`	the standard deviation to be used for classification. Is NULL by default. Calculated as necessary.

Details

Western Electric Rules classify data into zones of A (2-3 sigma), B (1-2 sigma), and C (0-1) sigma. This function returns numeric classifications 4 (> 3 sigma), 3 (positive class A), 2 (positive class B), 1 (positive class C), and 0 (equals mean). It also returns negative numeric classifications -4 (< 3 sigma), -3 (negative class A), -2 (negative class B), and -1 (negative class C).

Value

a vector with these positive/negative values: |4| = outlier, |3| = zone A, |2| = zone B, |1| = zone C, 0 = x.bar

Note

None

Author(s)

Rod Doe

References

None

See Also

classifyByWesternElectricRuleZones

Examples

```
## x = c(3.1, 2.1, 1.1, 0.1, 0, -0.1, -1.1, -2.1, -3.1)
## x.bar = 0
## sigma = 1
## xclass = classifyByWesternElectricRuleZones.numeric(x, x.bar, sigma)
## (xclass)
## # [1]  4  3  2  1  0 -1 -2 -3 -4


## The function is currently defined as
function (x, x.bar = NULL, sigma = NULL)
{
    if (is.null(x.bar) || is.null(sigma)) {
        x.bar = mean(x)
        sigma = sd(x)
    }
    gt.pos.three.sigma = which(x > (x.bar + (3 * sigma)))
    gt.pos.two.sigma = which(x > (x.bar + (2 * sigma)))
    gt.pos.one.sigma = which(x > (x.bar + sigma))
    gt.xbar = which(x > x.bar)
    eq.xbar = which(x == x.bar)
    lt.x.bar = which(x < x.bar)
    lt.neg.one.sigma = which(x < (x.bar - sigma))
    lt.neg.two.sigma = which(x < (x.bar - (2 * sigma)))
    lt.neg.three.sigma = which(x < (x.bar - (3 * sigma)))
    outliers = union(gt.pos.three.sigma, lt.neg.three.sigma)
    pos.A = setdiff(gt.pos.two.sigma, gt.pos.three.sigma)
    pos.B = setdiff(gt.pos.one.sigma, gt.pos.two.sigma)
    pos.C = setdiff(gt.xbar, gt.pos.one.sigma)
    neg.C = setdiff(lt.x.bar, lt.neg.one.sigma)
    neg.B = setdiff(lt.neg.one.sigma, lt.neg.two.sigma)
    neg.A = setdiff(lt.neg.two.sigma, lt.neg.three.sigma)
    zones = numeric(length(x))
    zones[gt.pos.three.sigma] = 4
    zones[pos.A] = 3
    zones[pos.B] = 2
    zones[pos.C] = 1
    zones[eq.xbar] = 0
    zones[neg.C] = -1
    zones[neg.B] = -2
    zones[neg.A] = -3
    zones[lt.neg.three.sigma] = -4
    return(zones)
  }
```

findWesternElectricRuleViolations
findWesternElectricRuleViolations

Description

Given a vector of numeric zones, this function looks for sequences of data that constitute Western Electric Rule violations.

Usage

```
findWesternElectricRuleViolations(x, x.bar = NULL, sigma = NULL)
```

Arguments

`x`	a vector of numeric values that are to be classified into Western Electric Rule zones
`x.bar`	the mean value to be used for classification. Is NULL by default. Calculated as necessary.
`sigma`	the standard deviation to be used for classification. Is NULL by default. Calculated as necessary.

Details

Rule 4 violation: This is 9 consecutive points that fall on the same side of the centerline (Zone C or beyond). Rule 3 violation: This is 4 out of 5 points that fall beyond the 1 sigma limit (Zone B or beyond) on the same side of the centerline. Rule 2 violation: This is 2 out of 3 consecutive points that fall beyond the 2 sigma limit (Zone B or beyond) on the same side of the centerline. Rule 1 violation: This is any point that falls beyond the 3 sigma limit (Zone C or beyond).

Value

a (possibly empty) dataframe with these numeric columns: idxInitial, idxFinal, rule

Note

None

Author(s)

Rod Doe

References

Western Electric Rules wikipedia site

See Also

classifyByWesternElectricRuleZones

Examples

```
x.bar = 0
## sigma = 1
## x = c(0, 1.1,1.2,1.1,  1.3,1.4,1.3, 1.4,1.5,1.4, 0,  1.1,-1.0,1.1,1.2,1.1, 0.0, 2.2,1.1,2.3, 0, 4.1, -4

## results =  findWesternElectricRuleViolations(x, x.bar, sigma)
## (results)

## The function is currently defined as
function (x, x.bar = NULL, sigma = NULL)
{
    if (is.null(x.bar) || is.null(sigma)) {
        x.bar = mean(x)
        sigma = sd(x)
    }
    results <- data.frame(idxInitial = numeric(), idxFinal = numeric(),
        rule = numeric())
    nItems = length(x)
    zones.numeric = classifyByWesternElectricRuleZones.numeric(x,
        x.bar, sigma)
    minRunLen = 9
    for (idxInitial in 1:(nItems - minRunLen + 1)) {
        idxFinal = (idxInitial + minRunLen - 1)
        if ((all(zones.numeric[idxInitial:idxFinal] > 0)) ||
            (all(zones.numeric[idxInitial:idxFinal] < 0))) {
            results = logWesternElectricRuleViolation(results,
                idxInitial, idxFinal, 4)
        }
    }
    minRunLen = 5
    for (idxInitial in 1:(nItems - minRunLen + 1)) {
        idxFinal = (idxInitial + minRunLen - 1)
        if ((any(zones.numeric[idxInitial:idxFinal] > 1)) ||
            (any(zones.numeric[idxInitial:idxFinal] < -1))) {
            countZoneB = 0
            countOpposite = 0
            matched = 0
            for (i in idxInitial:idxFinal) {
                if (zones.numeric[i] > 1) {
                  countZoneB = countZoneB + 1
                }
                else if (zones.numeric[i] < 0) {
                  countOpposite = countOpposite + 1
                }
            }
            if ((countZoneB == 4) && (countOpposite == 1)) {
                matched = 1
            }
            if (matched == 0) {
                countZoneB = 0
                countOpposite = 0
                for (i in idxInitial:idxFinal) {
                  if (zones.numeric[i] < -1) {
                    countZoneB = countZoneB + 1
                  }
                  else if (zones.numeric[i] > 0) {
```

```
                countOpposite = countOpposite + 1
              }
              if ((countZoneB == 4) && (countOpposite ==
                1)) {
                matched = 1
              }
            }
        }
        if (matched == 1) {
            results = logWesternElectricRuleViolation(results,
              idxInitial, idxFinal, 3)
        }
    }
}
minRunLen = 3
for (idxInitial in 1:(nItems - minRunLen + 1)) {
    idxFinal = (idxInitial + minRunLen - 1)
    if ((any(zones.numeric[idxInitial:idxFinal] > 2)) ||
        (any(zones.numeric[idxInitial:idxFinal] < -2))) {
        countZoneA = 0
        countSameSide = 0
        matched = 0
        for (i in idxInitial:idxFinal) {
            if (zones.numeric[i] > 2) {
              countZoneA = countZoneA + 1
            }
            else if (zones.numeric[i] > 0) {
              countSameSide = countSameSide + 1
            }
        }
        if ((countZoneA == 2) && (countSameSide == 1)) {
            matched = 1
        }
        if (matched == 0) {
            countZoneA = 0
            countSameSide = 0
            for (i in idxInitial:idxFinal) {
              if (zones.numeric[i] < -2) {
                countZoneA = countZoneA + 1
              }
              else if (zones.numeric[i] < 0) {
                countSameSide = countSameSide + 1
              }
              if ((countZoneB == 2) && (countSameSide ==
                1)) {
                matched = 1
              }
            }
        }
        if (matched == 1) {
            results = logWesternElectricRuleViolation(results,
              idxInitial, idxFinal, 2)
        }
    }
}
violations = sort(union(which(zones.numeric == 4), which(zones.numeric ==
    -4)))
```

```
    if (length(violations) > 0) {
        for (i in 1:length(violations)) {
            results = logWesternElectricRuleViolation(results,
                violations[i], violations[i], 1)
        }
    }
    return(results)
  }
```

`logWesternElectricRuleViolation`
logWesternElectricRuleViolation

Description

appends a new row onto an existing dataframe of rule violations

Usage

```
logWesternElectricRuleViolation(results, idxInitial, idxFinal, rule)
```

Arguments

`results`	a (possibly empty) dataframe with these numeric columns: idxInitial, idxFinal, rule
`idxInitial`	the index of the value in the source vector where a violation was initially detected
`idxFinal`	the index of the value in the source vector where the violation was last detected
`rule`	the Western Electric Rule number that was violated in the range idxInitial:idxFinal (inclusive). 0 if no violation.

Details

Simply appends a new row in a standard fashion so that THAT wheel need not be reinvented.

Value

the original results dataframe with one additional row added to it

Note

You are the man with six fingers. You killed my father. Prepare to die.

Author(s)

Inigo Montoya

References

None

See Also

None

Examples

```
## results <- data.frame(idxInitial = numeric(), idxFinal = numeric(), rule = numeric())
## results <- logWesternElectricRuleViolation(results, 5, 5, 1)
## (results)

## The function is currently defined as
function (results, idxInitial, idxFinal, rule)
{
    result <- data.frame(idxInitial = idxInitial, idxFinal = idxFinal,
        rule = rule)
    results <- rbind(results, result)
    return(results)
  }
```

---

plotSpcChart *plotSpcChart*

Description

Plots a standard Statistical Process Control (SPC) of a series of data average values and ranges.

Usage

```
plotSpcChart(batchAverages, batchRanges, x.bar.batchAverages = NULL, sigma.batchAverages = NULL,
```

Arguments

`batchAverages` a vector of numeric values that contains the average value of a series of batches

`batchRanges` a vector of numeric values that contains the range of value of a series of batches. Must be of the same length as batchAverages.

`x.bar.batchAverages` the mean value to be used for classification of batchAverages. Is NULL by default. Calculated as necessary.

`sigma.batchAverages` the standard deviation value to be used for classification of batchAverages. Is NULL by default. Calculated as necessary.

`x.bar.batchRanges` the mean value to be used for classification of batchAverages. Is NULL by default. Calculated as necessary.

`sigma.batchRanges` the standard deviation value to be used for classification of batchRanges. Is NULL by default. Calculated as necessary.

Details

Given a process in which batch average values and ranges of values are recorded, this function applies the standard Western Electric Rules to the data to determine SPC compliance. The batch averages and ranges are plotted in colors associated with their SPC status. A green datum indicates that the process is in control. A non-green datum indicates a rule violation and warrants a response by the process owner. For specific information about the Western Electric Rules, see the associated wikipedia article.

Value

a dataframe with these numeric columns: batchAverage, state.batchAverage, batchRange, state.batchRange

Note

Not only is travel at the speed of light impossible, it would be highly inconvenient because one's hat would frequently blow off.

Author(s)

Rod Doe

References

google wikipedia Western Electric Rules. For a deeper dive, see 'Statistical Process Control' by Donald Wheeler.

See Also

None

Examples

```
## batchAverages = c(0, 1.1,1.2,1.1,  1.3,1.4,1.3, 1.4,1.5,1.4,   0,  1.1,-1,1.1,1.2,1.1,   0,  2.2,1.1,2.3
## x.bar.batchAverages = 0
## sigma.batchAverages = 1
## batchRanges = numeric(length(batchAverages))
## x.bar.batchRanges = 10
## sigma.batchRanges = 1.0
## for (i in 1:length(batchRanges)) { batchRanges[i] = x.bar.batchRanges + rnorm(1, 0, 0.5) }
## result = plotSpcChart(batchAverages, batchRanges, x.bar.batchAverages, sigma.batchAverages, x.bar.batchR
## (result)


## The function is currently defined as
function (batchAverages, batchRanges, x.bar.batchAverages = NULL,
    sigma.batchAverages = NULL, x.bar.batchRanges = NULL, sigma.batchRanges = NULL)
{
    if ((is.null(x.bar.batchAverages)) || (is.null(sigma.batchAverages))) {
        x.bar.batchAverages = mean(batchAverages)
        sigma.batchAverages = sd(batchAverages)
    }
    if ((is.null(x.bar.batchRanges)) || (is.null(sigma.batchRanges))) {
        x.bar.batchRanges = mean(batchRanges)
        sigma.batchRanges = sd(batchRanges)
    }
```

```
spc.violations.batchAverages = findWesternElectricRuleViolations(batchAverages,
    x.bar.batchAverages, sigma.batchAverages)
spc.violations.batchRanges = findWesternElectricRuleViolations(batchRanges,
    x.bar.batchRanges, sigma.batchRanges)
spc.state.batchAverages = rep(0, length(batchAverages))
if (length(spc.violations.batchAverages[, 1]) > 0) {
    for (i in 1:length(spc.violations.batchAverages[, 1])) {
        spc.state.batchAverages[spc.violations.batchAverages$idxInitial[i]:spc.violations.batchAverages
    }
}
spc.state.batchRanges = rep(0, length(batchRanges))
if (length(spc.violations.batchRanges[, 1]) > 0) {
    for (i in 1:length(spc.violations.batchRanges[, 1])) {
        spc.state.batchRanges[spc.violations.batchRanges$idxInitial[i]:spc.violations.batchRanges$idxF
    }
}
spcData = data.frame(batchAverage = numeric(), state.batchAverage = numeric())
for (i in 1:length(batchAverages)) {
    nextRow = data.frame(batchAverage = batchAverages[i],
        state.batchAverage = spc.state.batchAverages[i],
        batchRange = batchRanges[i], state.batchRange = spc.state.batchRanges[i])
    spcData <- rbind(spcData, nextRow)
}
cols = c(rgb(0, 1, 0, 0.6), rgb(1, 1, 0, 0.6), rgb(1, 0.6,
    0, 0.6), rgb(0, 0, 1, 0.6), rgb(1, 0, 1, 0.6))
prevOp <- par(fig = c(0, 1, 0.5, 1))
par(mar = c(0, 5, 2, 2))
par(cex = 0.64)
plot(spcData$batchAverage, col = cols[spcData$state.batchAverage +
    1], xlab = "", xaxt = "n", type = "b", main = "SPC Chart",
    ylab = "Batch Averages", pch = 19)
legend("topleft", horiz = TRUE, legend = c("In Control",
    "Rule 1", "Rule 2", "Rule 3", "Rule 4"), col = cols,
    pch = 19)
abline(h = x.bar.batchAverages)
abline(h = (x.bar.batchAverages + (1 * sigma.batchAverages)),
    lty = "dashed", col = rgb(0, 0, 0, 0.4))
abline(h = (x.bar.batchAverages + (2 * sigma.batchAverages)),
    lty = "dashed", col = rgb(0, 0, 0, 0.4))
abline(h = (x.bar.batchAverages + (3 * sigma.batchAverages)),
    lty = "dashed", col = rgb(0, 0, 0, 0.4))
abline(h = (x.bar.batchAverages - (1 * sigma.batchAverages)),
    lty = "dashed", col = rgb(0, 0, 0, 0.4))
abline(h = (x.bar.batchAverages - (2 * sigma.batchAverages)),
    lty = "dashed", col = rgb(0, 0, 0, 0.4))
abline(h = (x.bar.batchAverages - (3 * sigma.batchAverages)),
    lty = "dashed", col = rgb(0, 0, 0, 0.4))
par(fig = c(0, 1, 0, 0.5), new = TRUE)
par(mar = c(5, 5, 0, 2))
plot(spcData$batchRange, col = cols[spcData$state.batchRange +
    1], xlab = "", ylab = "Batch Ranges", type = "b", pch = 19)
abline(h = x.bar.batchRanges)
abline(h = (x.bar.batchRanges + (1 * sigma.batchRanges)),
    lty = "dashed", col = rgb(0, 0, 0, 0.4))
abline(h = (x.bar.batchRanges + (2 * sigma.batchRanges)),
    lty = "dashed", col = rgb(0, 0, 0, 0.4))
abline(h = (x.bar.batchRanges + (3 * sigma.batchRanges)),
```

```
        lty = "dashed", col = rgb(0, 0, 0, 0.4))
    abline(h = (x.bar.batchRanges - (1 * sigma.batchRanges)),
        lty = "dashed", col = rgb(0, 0, 0, 0.4))
    abline(h = (x.bar.batchRanges - (2 * sigma.batchRanges)),
        lty = "dashed", col = rgb(0, 0, 0, 0.4))
    abline(h = (x.bar.batchRanges - (3 * sigma.batchRanges)),
        lty = "dashed", col = rgb(0, 0, 0, 0.4))
    par(prevOp)
    return(spcData)
  }
```

`spcSampleData` *spcSampleData - example of data produced by plotSpcData*

Description

Produced by invocation of plotSpcChart(batchAverages, batchRanges). Contains the initial data in the batchAverage and batchRange columns, with the applicable SPC state in the state.* columns.

Usage

```
data(spcSampleData)
```

Format

A data frame with 23 observations on the following 4 variables.

`batchAverage` a numeric vector

`state.batchAverage` a numeric vector

`batchRange` a numeric vector

`state.batchRange` a numeric vector

Details

The data in the batchAverages are specifically designed to violate Western Electric rules 4,3,2,1. The data in the batchRange are designed to be more typical.

Source

Low pressure atmospheric extraction (pulled from thin air)

References

None

Examples

```
data(spcSampleData)
head(spcSampleData)
## maybe str(spcSampleData) ; plot(spcSampleData) ...
```

Index