Contents

1 Description 1

2 Modules 3
2.1 inventory.continuous 3
2.2 inventory.discrete 5
2.3 inventory.solvers 6
2.4 inventory.utils 10

3 Examples 13

Bibliography 17

Python Module Index 19
Description

A Python library for the study of inventory management. Implements cost functions, heuristics, and datasets used in Inventory Management research.

Using any of the following subpackages requires an explicit import. For example, `import inventory.discrete`.

| continuous | --- | Cost functions for the continuous model |
| discrete    | --- | Cost functions for the discrete model   |
| solvers     | --- | Heuristics and optimal algorithms       |
| utils       | --- | Utilities for the datasets and GP       |

The articles addressed (full or partial coverage) are listed below, please refer to these for more information:
2.1 inventory.continuous

Provides the cost functions for the continuous model.

inventory.continuous.ecomStockoutP($dL_1, dL_2, s, Q$)
Calculate the stockout probability for the e-commerce model.

Parameters
- $dL_1$ – distribution of the demand of type 1 (must be continuous)
- $dL_2$ – distribution of the demand of type 2 (must be continuous)
- $s$ – the reorder point
- $Q$ – the reorder quantity

Returns stockout probability
Return type float

inventory.continuous.alphaTCecom($s, Q, dL_1, dL_2, A, B_1, D, r, v, *args, **kwargs$)
Calculate the total cost for the e-commerce model (alpha service level, $B_1$ items).

Parameters
- $dL_1$ – distribution of the demand of type 1 (must be continuous)
- $dL_2$ – distribution of the demand of type 2 (must be continuous)
- $s$ – the reorder point
- $Q$ – the reorder quantity
- $A$ – the setup cost
- $B_1$ – the backorder penalty
- $D$ – the total demand
- $r$ – inventory carrying charge
- $v$ – unit variable cost

Returns total cost
Return type float
inventory.continuous.sfactorTCecom\( (k, Q, dL1, dL2, A, B1, D, r, v, *\text{args}, **\text{kwargs}) \)

Calculate the total cost for the e-commerce model based on the service factor (alpha service level, B1 items).

**Parameters**
- \( dL1 \) – distribution of the demand of type 1 (must be continuous)
- \( dL2 \) – distribution of the demand of type 2 (must be continuous)
- \( k \) – the service factor
- \( Q \) – the reorder quantity
- \( A \) – the setup cost
- \( B1 \) – the backorder penalty
- \( D \) – the total demand
- \( r \) – inventory carrying charge
- \( v \) – unit variable cost

**Returns** total cost

**Return type** float

inventory.continuous.vectorizeTCecom\( (xL1, sigma1, xL2, sigma2, tcOpt=0, **\text{kwargs}) \)

Vectorize the cost function \( \text{alphaTCecom()} \).

**Parameters**
- \( xL1 \) – the mean demand of type 1.
- \( xL2 \) – the mean demand of type 2.
- \( sigma1 \) – the standard deviation for the demand of type 1.
- \( sigma2 \) – the standard deviation for the demand of type 2.

**Returns** vectorized total cost function

**Return type** callable

inventory.continuous.alphaTC\( (s, Q, dL, A, B1, D, h, **\text{kwargs}) \)

Calculate the total cost for an \( (s, Q) \) inventory control policy (alpha service level, B1 items).

**Parameters**
- \( dL \) – distribution of the demand (must be continuous)
- \( s \) – reorder point
- \( Q \) – reorder quantity
- \( A \) – setup cost (K)
- \( B1 \) – backorder penalty (p)
- \( D \) – total demand (lambda)
- \( h \) – on-hold cost

**Returns** total cost

**Return type** float

inventory.continuous.sfactorTC\( (k, Q, dL, A, B1, D, h, **\text{kwargs}) \)

Calculate the total cost for an \( (s, Q) \) inventory control policy based on the service factor (alpha service level, B1 items).
Parameters

- \(dL\) – distribution of the demand (must be continuous)
- \(k\) – service factor
- \(Q\) – reorder quantity
- \(A\) – setup cost (K)
- \(B1\) – backorder penalty (p)
- \(D\) – total demand (lambda)
- \(h\) – on-hold cost

Returns

total cost

Return type float

inventory.continuous.betaTC \((s, Q, dL, A, B1, D, rv, *args, **kwargs)\)

Calculate the total cost for an \((s, Q)\) inventory control policy (beta service level, continuous approximation of the discrete model).

Parameters

- \(dL\) – distribution of the demand (must be continuous)
- \(s\) – reorder point
- \(Q\) – reorder quantity
- \(A\) – setup cost (K)
- \(B1\) – backorder penalty (p)
- \(D\) – total demand (lambda)
- \(rv\) – on-hold cost (h)

Returns

total cost

Return type float

2.2 inventory.discrete

Provides the cost functions for the discrete model.

inventory.discrete.G \((y, ll, p, h, cdf)\)

Aggregates the holding and backorder costs in the discrete model.

Parameters

- \(y\) – the function y parameter
- \(ll\) – demand distribution expected value
- \(p\) – backorder penalty
- \(h\) – on-hold cost
- \(cdf\) – function used to calculate the cumulative distribution of the demand

Returns

the aggregated on-hold and backordered inventory cost

Return type float
inventory.discrete.discreteTC \( s, Q, IL, K, p, lbd, h, \text{xargs}, **\text{kwargs} \)
Calculates the total cost using the discrete model. As described in [FedergruenZheng1992] and [Zheng1992].

**Parameters**
- \( s \) – reorder point
- \( Q \) – reorder quantity
- \( IL \) – demand distribution expected value
- \( K \) – order setup cost
- \( p \) – backorder penalty
- \( lbd \) – total demand
- \( h \) – holding cost

**Returns** total cost
**Return type** float

**Example**
```python
>>> round(discreteTC(50, 7, 50, 1, 25, 50, 10), 2)
95.46
>>> round(discreteTC(56, 7, 50, 1, 100, 50, 10), 2)
142.81
>>> round(discreteTC(46, 6, 50, 1, 25, 50, 25), 2)
153.35
```

inventory.discrete.pdcdf \( x, lbd \)
Returns the pre-calculated value for the cumulative distribution function of a Poisson distribution.

**Parameters**
- \( lbd \) – demand distribution expected value (lambda)
- \( x \) – point to calculate the cdf (integer)

**Returns** \( P(X \leq x) \)
**Return type** float

**Raises** Warning – if the pair \((x, lbd)\) does not exist in the table, and the return value is calculated on-the-fly using scipy.stats

**Examples**
```python
>>> round(pdcdf(1, 25),4) == round(poisson.cdf(1,25))
True
>>> round(pdcdf(1e6, 25),4) == round(poisson.cdf(1e6,25))
True
```

### 2.3 inventory.solvers

Implements the heuristics and optimal algorithms.

inventory.solvers.alphaSolver \( x, std, A, BI, D, rv, \text{xargs}, **\text{kwargs} \)
Simultaneous (iterative) solver for alpha service level in traditional retail.

**Parameters**
• \( x \) – demand distribution expected value
• \( \text{std} \) – demand distribution standard deviation
• \( A \) – setup cost (\( K \))
• \( B_1 \) – backorder penalty (\( p \))
• \( D \) – total demand (\( \lambda \))
• \( r_v \) – on-hold cost (\( h \))

\textbf{Returns}  service factor, reorder quantity

\textbf{Return type}  \textbf{tuple}

\texttt{inventory.solvers.betaSolver}(\( lL, K, p, l, h \))

Calculates (\( r, Q \)) parameters from [Zheng1992] heuristic.

\textbf{Parameters}

• \( lL \) – demand distribution expected value
• \( K \) – order setup cost
• \( p \) – backorder penalty
• \( l \) – total demand
• \( h \) – holding cost

\textbf{Returns}  reorder point, reorder quantity

\textbf{Return type}  \textbf{tuple}

\textbf{Examples}

```python
>>> import functools
>>> map(functools.partial(round, ndigits = 1), betaSolver(50, 1, 25, 50, 10))
[48.9, 3.7]
>>> map(functools.partial(round, ndigits = 1), betaSolver(50, 5, 25, 50, 10))
[47.6, 8.4]
>>> map(functools.partial(round, ndigits = 1), betaSolver(50, 25, 25, 50, 10))
[44.7, 18.7]
>>> map(functools.partial(round, ndigits = 1), betaSolver(50, 100, 25, 50, 10))
[39.3, 37.4]
>>> map(functools.partial(round, ndigits = 1), betaSolver(50, 1000, 25, 50, 10))
[16.2, 118.3]
```

\texttt{inventory.solvers.discreteSolver}(\( lL, K, p, l, h, \text{cdf=<function pdcdf>} \))

Calculates the optimal reorder point and quantity parameters as presented by [FedergruenZheng1992].

\textbf{Parameters}

• \( lL \) – demand distribution expected value
• \( K \) – order setup cost
• \( p \) – backorder penalty
• \( l \) – total demand
• \( h \) – holding cost
• \( \text{cdf} \) – function used to calculate the cumulative distribution function of a Poisson

\textbf{Returns}  reorder point, reorder quantity

2.3. \texttt{inventory.solvers}
Return type  tuple

Example

```python
>>> discreteSolver(50, 1, 25, 50, 10)
(50, 7)
>>> discreteSolver(50, 5, 25, 50, 10)
(48, 12)
>>> discreteSolver(50, 25, 25, 50, 10)
(44, 23)
>>> discreteSolver(50, 100, 25, 50, 10)
(38, 40)
>>> discreteSolver(50, 1000, 25, 50, 10)
(15, 120)
```

inventory.solvers.eoq(K, lbd, h)
Calculates the order quantity according to the Economic Order Quantity (EOQ) model, without planned backorders.

Parameters

- **K** – order setup cost
- **lbd** – total demand
- **h** – holding cost

Returns  reorder quantity

Return type  float

inventory.solvers.gallego(lL, K, p, l, h, L, costfun=<function discreteTC>, minmethod='cobyla')
Calculates the reorder quantity (Q) parameter from [Gallego1998] heuristic, and finds the corresponding reorder point (r) through numerical optimisation (using constrained optimization by linear approximation).

Parameters

- **lL** – demand distribution expected value
- **K** – order setup cost
- **p** – backorder penalty
- **l** – total demand
- **h** – holding cost
- **L** – lead time
- **costfun** – cost function to be minimised when calculating the reorder point
- **minmethod** – optimisation algorithm to use with minimize()
- **roundmethod** – function used to round the floating point reorder quantity

Returns  reorder point, reorder quantity

Return type  tuple

inventory.solvers.kleinauGP(lL, K, p, l, h, L)
Calculates the reorder point and quantity parameters from [KleinauThonemann2004] full Genetic Programming solution.

Parameters

- **lL** – demand distribution expected value
• \(K\) – order setup cost
• \(p\) – backorder penalty
• \(l\) – total demand
• \(h\) – holding cost
• \(L\) – lead time

**Returns**  reorder point, reorder quantity

**Return type**  tuple

`inventory.solvers.kleinauNum(IL, K, p, l, h, L, minmethod='cobyla')`

Calculates the reorder quantity (Q) parameter from [KleinauThonemann2004] hybrid GP approach, and finds the corresponding reorder point (\(r\)) through numerical optimisation (using constrained optimization by linear approximation).

**Parameters**

• \(IL\) – demand distribution expected value
• \(K\) – order setup cost
• \(p\) – backorder penalty
• \(l\) – total demand
• \(h\) – holding cost
• \(L\) – lead time
• \(costfun\) – cost function to be minimised when calculating the reorder point
• \(minmethod\) – optimisation algorithm to use with `minimize()`
• \(roundmethod\) – function used to round the floating point reorder quantity

**Returns**  reorder point, reorder quantity

**Return type**  tuple

`inventory.solvers.zhengNumeric(IL, K, p, l, h, L, costfun=<function discreteTC>, minmethod='cobyla', roundmethod=<built-in function round>)`

Calculates the reorder quantity (Q) parameter from [Zheng1992] heuristic, and finds the corresponding reorder point (\(r\)) through numerical optimisation (using constrained optimization by linear approximation).

**Parameters**

• \(IL\) – demand distribution expected value
• \(K\) – order setup cost
• \(p\) – backorder penalty
• \(l\) – total demand
• \(h\) – holding cost
• \(L\) – lead time
• \(costfun\) – cost function to be minimised when calculating the reorder point
• \(minmethod\) – optimisation algorithm to use with `minimize()`
• \(roundmethod\) – function used to round the floating point reorder quantity

**Returns**  reorder point, reorder quantity
Return type  tuple

## 2.4  inventory.utils

Implements utilities for data generation and plotting, and helpers for the DEAP genetic programming algorithm.

inventory.utils.cartesian(arrays, out=None)
Generates a cartesian product of input arrays.

inventory.utils.computePlot(data)
Computes the cost surface for every entry in data.

inventory.utils.genSample(originData, lowerDev=20, upperDev=50)
Generates random samples from the originData example, with each variable transformed between lowerDev and upperDev.

inventory.utils.getLabels()
Returns the labels for the commerce variables.

inventory.utils.getfitcases(vlist=None, costfun=None, solver=None, backorders=True)
Returns a list of problem instances, based on the cartesian product of vlist

**Parameters**

- **vlist** – the base list to build the cartesian product
- **costfun** – function used to calculate the cost
- **solver** – heuristic used to calculate (r,Q) parameters for each instance
- **backorders** – if False and r < 0, then r = 0

**Returns**  list of instances

Return type  list

inventory.utils.loadcmdargs()
Parses the command line arguments to use with scripts.

inventory.utils.plotETRC(t, show=True)
Plots a wireframe from surface data.

inventory.utils.plotETRCsurface(t, show=True, colormap=None)
Plots a surface for a data record.

inventory.utils.protectedDiv(left, right)
Safe version of the operator div() to use with GP.

inventory.utils.protectedSqrt(value)
Safe version of the operator sqrt() to use with GP.

inventory.utils.runCoEA(toolbox, stats, logbook, numGen=100, rnd=False, cxp=0.6, mutp=1.0, **kwargs)
Initializes and runs the coevolutionary algorithm.

**Returns**  the best individual of the run.

inventory.utils.setupdeap(args, dataset, inputs, evalfun, discrete=True)
Builds the evolutionary algorithm.

**Returns**  Toolbox, Statistics, Logbook

Return type  tuple
inventory.utils.statsfun(distr, statistic, toolbox, discrete=False)

Wrapper (decorator) to use statistical functions in the GP function set.

**Parameters**

- **distr** – the name of the distribution variable
- **statistic** – the name of the statistical function (cdf, pdf, etc)
- **toolbox** – DEAP Toolbox
- **discrete** – used to force the casting of the input

**Returns** parameterized function

**Return type** callable
The following code exemplifies how to use the library, in particular the functions regarding the discrete model. The code can be run using the corresponding iPython notebook `example_discrete.ipynb`.

```python
from __future__ import division

import logging
from math import sqrt, ceil
from operator import itemgetter
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import poisson, norm, rv_continuous
from inventory.discrete import discreteTC
from inventory.solvers import discreteSolver, gallego, kleinauGP, kleinauNum, zhengNumeric
from inventory.utils import KLEINAU_SET as KS, ZHENG_SET as ZS, getfitcases

backorders = True
fitcases = getfitcases(vlist = KS, costfun=discreteTC, solver = discreteSolver, backorders = backorders)

print len(fitcases)

# Heuristics and Gaps to optimal - GALLEGO
gallego_results = []
for l, h, p, K, L, sOpt, qOpt, costOpt in fitcases:
    sg, Qg = gallego(l*L, K, p, l, h, L, minmethod='cobyla')
    # sg = 0 if sg < 0 else int(round(sg))
    sg = int(round(sg))
    if not backorders and sg < 0:
        sg = 0
    Qg = int(round(Qg))
    if Qg == 0: Qg = 1
    gcost = discreteTC(sg, Qg, l*L, K, p, l, h)
    gallego_results.append(((1, h, p, K, L, sOpt, qOpt, sg, Qg, gcost, 
                                   ((gcost - costOpt) / costOpt)*100))

# for g in gallego_results: print g

print sum(abs(fc[-1]) for fc in gallego_results) / len(fitcases)

# Heuristics and Gaps to optimal - ZHENG numeric
zhengnum_results = []
for l, h, p, K, L, sOpt, qOpt, costOpt in fitcases:
```
sk, Qk = zhengNumeric(l*L, K, p, l, h, L, roundmethod=ceil)
rd, Qd = betaSolver(l*L, K, p, l, h)
if not backorders and sk < 0:
    sk = 0
skr = int(round(sk))
# with ceil, error is smaller
# Qk is already rounded
# Qd = int(round(Qd))
gcost = discreteTC(skr, Qk, l*L, K, p, l, h)
zhengnum_results.append((l, h, p, K, L, sOpt, qOpt, costOpt, rd, sk, Qk, gcost, ((gcost - costOpt) / costOpt) * 100))

# for g in zhengnum_results: print g
print sum(abs(fc[-1]) for fc in zhengnum_results) / len(fitcases)

# Heuristics and Gaps to optimal - KLEINAU GP
kleinaugp_results = []
for l, h, p, K, L, sOpt, qOpt, costOpt in fitcases:
    sk, Qk = kleinauGP(l*L, K, p, l, h, L)
    # sg = 0 if sg < 0 else int(round(sg))
    if not backorders and sk < 0:
        sk = 0
    sk = int(round(sk))
    Qk = int(round(Qk))
    gcost = discreteTC(sk, Qk, l*L, K, p, l, h)
    kleinaugp_results.append((l, h, p, K, L, sOpt, qOpt, costOpt, sk, Qk, gcost, ((gcost - costOpt) / costOpt) * 100))

# for g in kleinaugp_results: print g
print sum(fc[-1] for fc in kleinaugp_results) / len(fitcases)

# Heuristics and Gaps to optimal - KLEINAU NUMERIC
kleinaunum_results = []
for l, h, p, K, L, sOpt, qOpt, costOpt in fitcases:
    sk, Qk = kleinauNum(l*L, K, p, l, h, L)
    # sg = 0 if sg < 0 else int(round(sg))
    if not backorders and sk < 0:
        sk = 0
    sk = int(round(sk))
    Qk = int(round(Qk))
    gcost = discreteTC(sk, Qk, l*L, K, p, l, h)
    kleinaunum_results.append((l, h, p, K, L, sOpt, qOpt, costOpt, sk, Qk, gcost, ((gcost - costOpt) / costOpt) * 100))

# for g in kleinaunum_results: print g
print sum(fc[-1] for fc in kleinaunum_results) / len(fitcases)

# Test the best individual obtained with CCGP
import cPickle as pkl
from operator import add, mul, sub, div
from inventory.utils import protectedDiv, protectedSqrt

sfun, qfun = pkl.load(open('data/DTkleinau_n4000s400_cxp2mutp2_35_bestinds.pkl'))
ccgp_results = []
for l, h, p, K, L, sOpt, qOpt, costOpt in fitcases:
    lbd = 1
    k = K
    sgp, Qgp = (eval(sfun, locals()), eval(qfun, locals()))

    sgp = int(round(sgp))
    if not backorders and sgp < 0: sgp = 0

    Qgp = int(round(Qgp))
    if Qgp == 0: Qgp = 1

    gcost = discreteTC(sgp, Qgp, l*L, K, p, l, h)
    ccpp_results.append((l, h, p, K, L, sOpt, qOpt, sgp, Qgp, gcost, ((gcost - costOpt) / costOpt)*100))

# for g in ccpp_results: print g

print sum(fc[-1] for fc in ccpp_results) / len(fitcases)

# test differences in the resulting gap distributions
from functools import partial
from itertools import imap
from scipy.stats import describe, ttest_ind

def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(imap(pred, iterable))

def lt(a = 1, b = 1):
    return a < b

def gt(a = 1, b = 1):
    return a > b

def distribution(data):
    return [quantify(data, partial(lt, b = 1)),
            quantify(data, partial(lt, b = 2)),
            quantify(data, partial(lt, b = 3)),
            quantify(data, partial(gt, b = 5))]

zz = [x[-1] for x in zhengnum_results]
gg = [x[-1] for x in gallego_results]
kk = [x[-1] for x in kleinaunum_results]
ccgp = [x[-1] for x in ccpp_results]

print distribution(zz)
print distribution(gg)
print distribution(kk)
print distribution(ccgp)

# Gallego vs ZhengNumeric
print ttest_ind(zz, gg, equal_var = False)
# ZhengNumeric vs CCGP
print ttest_ind(zz, ccpp, equal_var = False)
# Gallego vs CCGP
print ttest_ind(gg, ccpp, equal_var = False)
# KleinauNumeric vs CCGP
print ttest_ind(kk, ccpp, equal_var = False)


inventory, 1
inventory.continuous, 3
inventory.discrete, 5
inventory.solvers, 6
inventory.utils, 10
Index

A
alphaSolver() (in module inventory.solvers), 6
alphaTC() (in module inventory.continuous), 4
alphaTCEcom() (in module inventory.continuous), 3

B
betaSolver() (in module inventory.solvers), 7
betaTC() (in module inventory.continuous), 5

cartesian() (in module inventory.utils), 10
computePlot() (in module inventory.utils), 10

discreteSolver() (in module inventory.solvers), 7
discreteTC() (in module inventory.discrete), 5

ecomStockoutP() (in module inventory.continuous), 3
eq() (in module inventory.solvers), 8

G
G() (in module inventory.discrete), 5
gallego() (in module inventory.solvers), 8
genSample() (in module inventory.utils), 10
getfitcases() (in module inventory.utils), 10
getLabels() (in module inventory.utils), 10

I
inventory (module), 1
inventory.continuous (module), 3
inventory.discrete (module), 5
inventory.solvers (module), 6
inventory.utils (module), 10

K
tKeinauGP() (in module inventory.solvers), 8
tKeinauNum() (in module inventory.solvers), 9

L
loadcmdargs() (in module inventory.utils), 10

P
pdcdf() (in module inventory.discrete), 6
plotETRC() (in module inventory.utils), 10
plotETRCsurface() (in module inventory.utils), 10
protectedDiv() (in module inventory.utils), 10
protectedSqrt() (in module inventory.utils), 10

R
runCoEA() (in module inventory.utils), 10

S
setupdeap() (in module inventory.utils), 10
sfactorTC() (in module inventory.continuous), 4
sfactorTCEcom() (in module inventory.continuous), 3
Statsfun() (in module inventory.utils), 10

V
vectorizeTCEcom() (in module inventory.continuous), 4

Z
zhengNumeric() (in module inventory.solvers), 9