```
#### Schaffer
rank   name        med             iqr                                                  10%             30%             50%             70%             90%
=============================================================================================================================================================
   1   DE,             2.0,            0.0   (*                  |                )          2.00,           2.00,           2.00,           2.00,           2.00
   2   SA,    4433871718.52,   9321750053.48   (--    *      --|--------        )   298067342.45,   1617492063.08,   4433871718.52,   8937617497.34,   16104087880.19
   2   GA,    4439275438.94,   8457134042.18   (--    *     ---|--------        )   251424699.05,   1746158724.72,   4439275438.94,   8652605692.45,   16069020415.64
   2   MWS,   4830433200.74,   9059409939.71   (--     *      -|-------        )   226227659.54,   1979652488.66,   4830433200.74,   9448524783.76,   15870004773.89

#### Kursawe
rank   name     med     iqr                                                 10%       30%       50%       70%       90%
=============================================================================================================================
   1   DE,    -13.73,    0.0   (*              |              )   -13.73,   -13.73,   -13.73,   -13.73,   -13.73
   2   MWS,    -2.43,   4.42   (       --   *--|              )    -6.88,    -4.55,    -2.43,    -1.23,     2.16
   3   GA,     -1.24,   6.13   (      ----   * |----          )    -7.82,    -3.50,    -1.24,     1.00,     5.40
   3   SA,     -1.13,   7.02   (      ----   * |----          )    -7.28,    -3.48,    -1.13,     1.75,     5.88

#### Fonseca
rank   name   med    iqr                                          10%      30%      50%      70%      90%
=============================================================================================================
   1   DE,    1.0,    0.0   (*             |              )    1.00,    1.00,    1.00,    1.00,    1.00
   2   GA,    2.0,   0.05   (              |  -----------*)    1.62,    1.97,    2.00,    2.00,    2.00
   2   SA,    2.0,   0.05   (              |  -----------*)    1.63,    1.98,    2.00,    2.00,    2.00
   2   MWS,   2.0,    0.0   (              |             *)    2.00,    2.00,    2.00,    2.00,    2.00

#### ZDT1
rank   name    med     iqr                                           10%      30%      50%      70%      90%
=============================================================================================================
   1   DE,     1.81,   0.03   (*             |             )    1.79,    1.80,    1.81,    1.83,    1.85
   2   GA,     9.85,   1.24   (            -- *-            )    8.60,    9.38,    9.85,   10.27,   10.99
   3   SA,     9.92,   1.33   (             - *-            )    8.77,    9.42,    9.92,   10.51,   11.24
   4   MWS,   15.04,   1.24   (               |      -- *-   )   13.85,   14.74,   15.04,   15.68,   16.17

#### ZDT3
rank   name    med    iqr                                          10%      30%      50%      70%      90%
============================================================================================================
   1   DE,    0.69,   0.01   (*              |             )    0.69,    0.69,    0.69,    0.70,    0.70
   2   GA,    5.02,   4.08   (  ----  *    --|-            )    2.12,    3.96,    5.02,    7.22,    9.25
   3   MWS,   5.26,   3.02   (         *   --|             )    4.73,    5.00,    5.26,    7.58,    9.05
   3   SA,    6.31,   4.19   (    ---    *  -|-            )    2.91,    4.64,    6.31,    8.09,    9.55

#### Viennet3
rank   name    med      iqr                                         10%      30%      50%      70%      90%
=============================================================================================================
   1   DE,    15.97,     0.0   (*              |             )   15.97,   15.97,   15.97,   15.97,   15.97
   2   GA,    23.54,    9.29   ( -  * ------   |             )   17.75,   20.03,   23.54,   28.02,   38.95
   2   SA,    24.43,   11.14   (--  *  ------  |             )   17.50,   20.53,   24.43,   28.47,   39.39
   3   MWS,    30.2,    9.64   (        * -----|             )   24.93,   26.35,   30.20,   34.23,   45.14

#### DTLZ7
rank   name     med        iqr                                          10%        30%        50%        70%        90%
=============================================================================================================================
   1   DE,     237.19,      9.84   (*              |             )    228.76,    233.67,    237.19,    241.28,    248.99
   2   GA,    4503.65,   4464.99   (*              |             )   3090.76,   3677.91,   4503.65,   6900.00,   19353.06
   2   MWS,   4914.42,     507.5   (*              |             )   4575.64,   4710.84,   4914.42,   5132.26,   16185.45
   3   SA,    5537.41,   6896.08   (*              |             )   3075.42,   4013.06,   5537.41,   8910.81,   29922.83

#### Schwefel(10)
rank   name      med          iqr                                          10%          30%          50%          70%          90%
=============================================================================================================================================
   1   DE,     29055.54,    87818.58   (*-             |             )    7533.87,    18668.08,    29055.54,    79429.66,    145635.85
   2   SA,     617412.73,   406020.96   (    --   * ----|             )   304619.26,   464264.54,   617412.73,   773111.08,   1103896.86
   2   GA,     666256.45,   439476.84   (    ---  *  ---|             )   313037.22,   504058.13,   666256.45,   836538.66,   1093304.01
   2   MWS,    683668.89,   489964.68   (  ----    * ---|             )   174833.76,   433619.86,   683668.89,   861319.91,   1036959.64

#### Schwefel(20)
rank   name       med          iqr                                          10%          30%          50%          70%          90%
=============================================================================================================================================
   1   DE,     439178.34,    254034.51   (-*-            |             )    237826.72,    350454.33,    439178.34,    536613.94,    847545.38
   2   MWS,    2389887.37,   1074411.22   (       --  *  -|-            )   1558824.48,   2109879.28,   2389887.37,   2942102.83,   3563112.96
   3   SA,      2784140.3,   1117257.34   (        ---  * |---          )   1749989.79,   2339980.20,   2784140.30,   3250374.39,   3967674.76
   3   GA,     2846787.69,   1114277.84   (        ---  * |---          )   1906111.36,   2453742.73,   2846787.69,   3282535.87,   4001928.99
```

```
#### Schwefel(40)
rank   name       med           iqr                                                  10%           30%           50%           70%           90%
==========================================================================================================================================
   1   DE,     2781053.19,   1028782.23   (  *            |             )   2075053.24,   2429718.90,    2781053.19,    3249668.47,    3688318.03
   2   SA,    10658474.53,   3157534.03   (          ---  * ----         )   7813714.37,   9440213.91,   10658474.53,   11854913.14,   13999155.18
   3   GA,    10829724.73,   3493472.27   (           --  |* ---         )   8160758.83,   9611977.62,   10829724.73,   12360221.38,   14332062.11
   4   MWS,   11548942.23,   3739233.66   (             -- | * --         )   8774079.34,   9917807.52,   11548942.23,   12966544.35,   13924738.49
```

```
from __future__ import division, print_function, unicode_literals

import random
import functools
import math
import itertools
import collections


def pretty_input(t):
    float_format = lambda x: '{:.2f}'.format(x)
    str_tuple = tuple(float_format(x) for x in t)
    return ', '.join(s for s in str_tuple)


def pairs(xs):
    # from https://docs.python.org/2/library/itertools.html
    a, b = itertools.tee(xs)
    next(b, None)
    for p in itertools.izip(a, b):
        yield p


class memo(object):  # noqa -- TODO: rethink this name
    '''adapted from https://github.com/timm/sbse14/wiki/basepy'''

    def __init__(self, **kwargs):
        self.__dict__.update(kwargs)

    def to_str(self, depth=0, indent=4, infix=': ', sep=', ', d=None):
        return '{' + self._to_str(
            depth=depth + 1,
            indent=indent,
            infix=infix,
            sep=sep,
            d=self.__dict__ if d is None else d) + '}'

    def _to_str(self, depth, indent, infix, sep, d):
        after, before = [], []
        rv = ''
        for k in sorted([s for s in d.keys() if s[0] != '_']):
            val = d[k]
            if isinstance(val, memo) or type(val) == dict:
                after.append(k)
            else:
                before.append('{}{}{}'.format(k, infix, repr(val)))
        if before:
            rv += '\n' + ' ' * depth * indent
            rv += sep.join(before)
        rv += '\n'

        for k in after:
            rv += ''.join([' ' * depth * indent, k, infix, '{'])
            k = d[k]
            k = k if type(k) == dict else k.__dict__
            rv += ''.join([self._to_str(depth=depth+1, indent=indent,
                                        infix=infix, sep=sep, d=k),
                           ' ' * depth * indent,
                           '}\n'])

        return rv


def memoize(f):
    'memoizer for single-arg functions'
    d = {}

    @functools.wraps(f)
    def wrapper(x):
        try:
            return d[x]
        except KeyError:
            d[x] = f(x)
            return d[x]

    return wrapper


@memoize
def memo_sqrt(x):
    return math.sqrt(x)


def tuple_replace(t, replace_at, value):
    return tuple(value if i == replace_at else v for i, v in enumerate(t))


def random_index(x):
    if isinstance(x, dict):
        return random.choice(x.keys)
    if isinstance(x, collections.Iterable):
        return random.randint(0, len(x) - 1)
    raise ValueError('{} is not a dict or Iterable'.format(x))


class StringBuilder(object):
    def __init__(self, *args):
        self._s = ''.join(args)
        self._next = []

    def append(self, arg):
        'recurse through iterables in args, adding all strings to _next '
        'raises TypeError if it finds a non-Iterable non-string'
        if isinstance(arg, basestring):
            self._next.append(arg)
        elif isinstance(arg, collections.Iterable):
            map(self.append, arg)
        else:
            raise TypeError('{} not a string or iterable'.format(arg))

    def __iadd__(self, arg):
        self.append(arg)
        return self

    def as_str(self):
        'build and cache _s if necessary, then return it.'
        if self._next:
            self._s += ''.join(self._next)
            self._next = []
        return self._s

    def __repr__(self):
        return "{}('{}')".format(self.__class__.__name__, self.as_str())


class NullObject(object):
    __slots__ = ()

    def __init__(self, *args, **kw):
        return None

    def _return_self(self, *name, **kw):
        return self

    __getattribute__ = _return_self
    __setattr__ = _return_self
    __iadd__ = _return_self
    __call__ = _return_self

    def __bool__(self, *args, **kw):
        return False
    __nonzero__ = __bool__
```

```
from __future__ import division, print_function

import math

import base


def median(xs, is_sorted=False):
    # implementation from http://stackoverflow.com/a/10482734/3408454
    if is_sorted:
        xs = sorted(xs)
    n = len(xs)
    return xs[n // 2] if n % 2 else (xs[n // 2] + xs[n // 2 - 1]) / 2


def mean(xs):
    return sum(xs) / len(xs)


def iqr(xs):
    n = len(xs)
    return xs[int(n * .75)] - xs[int(n * .25)]


def standard_deviation(xs, mean=None):
    if mean is None:
        mean = mean(xs)
    return math.sqrt((sum(x - mean) for x in xs) ** 2)


def norm(x, lo, hi):
    return (x - lo) / (hi - lo)


def xtile(xs, lo=0, hi=0.001,
          width=50,
          chops=[0.1, 0.3, 0.5, 0.7, 0.9],
          marks=["-", " ", " ", " ", "-", " "],
          bar="|", star="*",
          show=" {: >6.2f}",
          as_list=False):
    """The function _xtile_ takes a list of (possibly) unsorted numbers and
    presents them as a horizontal xtile ascii chart. The default is a
    contracted _quintile_ that shows the 10,30,50,70,90 breaks in the data by
    default. These breaks can be customized with the chops parameter.
    """

    xs = sorted(xs)

    lo = min(lo, xs[0])
    hi = max(hi, xs[-1])
    if hi == lo:
        hi += .001  # ugh

    out = [' '] * width

    pos = lambda p: xs[int(len(xs) * p)]
    place = lambda x: min(width-1, int(len(out) * norm(x, lo, hi)))

    what = [pos(p) for p in chops]
    where = [place(n) for n in what]

    for one, two in base.pairs(where):
        for i in range(one, two):
            out[i] = marks[0]
        marks = marks[1:]

    out[int(width / 2)] = bar
    out[place(pos(0.5))] = star

    if as_list:
        rv = ['(' + ''.join(out) + ")"]
        rv.extend(show % x for x in what)
        return rv

    return ''.join(out) + "," + ','.join([show.format(x) for x in what])
```

```
from __future__ import division, print_function

import random
import functools
import collections
import itertools

from sortedcontainers import SortedList

from witschey import base


class Log(object):
    """Keep a random sample of stuff seen so far. Based on Dr. Menzies'
    implementation."""

    MAX_SIZE = 256

    def __init__(self, inits=None, label=None, max_size=MAX_SIZE):
        self._cache = SortedList()
        self._report = None
        self.label = label or ''
        self._n = 0
        self.max_size = max_size
        self._valid_statistics = False
        self._invalidate_statistics()
        if inits:
            map(self.__iadd__, inits)

    def random_index(self):
        return base.random_index(self._cache)

    @classmethod
    def wrap(cls, x, max_size=MAX_SIZE):
        if isinstance(x, cls):
            return x
        return cls(inits=x, max_size=max_size)

    def __len__(self):
        return len(self._cache)

    def extend(self, xs):
        if not isinstance(xs, collections.Iterable):
            raise TypeError()
        map(self.__iadd__, xs)

    def __iadd__(self, x):
        if x is None:
            return x

        self._n += 1

        if isinstance(x, Log):
            map(self.__iadd__, x._cache)

        changed = False

        # if cache has room, add item
        if self.max_size is None or len(self._cache) < self.max_size:
            changed = True
            self._cache.add(x)
        # cache is full: maybe replace an old item
        else:
            # items less likely to be replaced later in the run:
            # leads to uniform sample of entire run
            if random.random() <= self.max_size / len(self):
                changed = True
                self._cache.remove(random.choice(self._cache))
                self._cache.add(x)

        if changed:
            self._invalidate_statistics()
            self._change(x)
```



```
            return self

        def __add__(self, x, max_size=MAX_SIZE):
            inits = itertools.chain(self._cache, x._cache)
            return self.__class__(inits=inits, max_size=max_size)

        def any(self):
            return random.choice(self._cache)

        def report(self):
            if self._report is None:
                self._report = self._generate_report()
            return self._report

        def setup(self):
            raise NotImplementedError()

        def as_list(self):
            return self._cache.as_list()

        def _invalidate_statistics(self):
            '''
            default implementation. if _valid_statistics is something other than
            a boolean, reimplement!
            '''
            self._valid_statistics = False

        def ish(self, *args, **kwargs):
            raise NotImplementedError()

        def _change(self, x):
            '''
            override to add incremental updating functionality
            '''
            pass

        def _prepare_data(self):
            s = '_prepare_data() not implemented for ' + self.__class__.__name__
            raise NotImplementedError(s)

        def contents(self):
            return self._cache.as_list()


    def statistic(f):
        '''
        decorator for log functions that return statistics about contents.
        if _valid_statistics is False, generate valid stats before calling
        the wrapped function.
        '''
        @functools.wraps(f)
        def wrapper(*args, **kwargs):
            self = args[0]
            if not self._valid_statistics:
                self._prepare_data()
            return f(*args, **kwargs)

        return wrapper
```

```
from __future__ import division

from log import Log
from witschey import base
from witschey import basic_stats


class NumberLog(Log):

    def __init__(self, *args, **kwargs):
        super(NumberLog, self).__init__(*args, **kwargs)

        self._invalidate_statistics()

    @property
    def hi(self):
        return self._cache[-1]  # assumes SortedList implementation

    @property
    def lo(self):
        return self._cache[0]  # assumes SortedList implementation

    def _invalidate_statistics(self):
        self._cached_mean, self._cached_median = None, None
        self._cached_sd, self._cached_iqr = None, None

        super(NumberLog, self)._invalidate_statistics()

    def norm(self, x):
        "normalize the argument with respect to maximum and minimum"
        if self.hi == self.lo:
            raise ValueError('hi and lo of {} are equal'.format(self.__name__))
        return basic_stats.norm(x, self.lo, self.hi)

    def _prepare_data(self):
        if not self._valid_statistics:
            pass
        self._valid_statistics = True

    def _generate_report(self):
        return base.memo(median=self.median(), iqr=self.iqr(),
                         lo=self.lo, hi=self.hi)

    def ish(self, f=0.1):
        """return a num likely to be similar to/representative of
        nums in the distribution"""
        return self.any() + f*(self.any() - self.any())

    def median(self):
        if self._cached_median is not None:
            return self._cached_median
        self._cached_median = basic_stats.median(self._cache)
        return self._cached_median

    def mean(self):
        if self._cached_mean is not None:
            return self._cached_mean
        self._cached_mean = basic_stats.mean(self._cache)
        return self._cached_mean

    def standard_deviation(self):
        if self._cached_sd is not None:
            return self._cached_sd
        self._cached_sd = basic_stats.standard_deviation(
            self._cache, mean=self.mean())
        return self._cached_sd

    def iqr(self):
        if self._cached_iqr is not None:
            return self._cached_iqr
        self._cached_iqr = basic_stats.iqr(self._cache)
        return self._cached_iqr
```



```
    def xtile(self, *args, **kw):
        return basic_stats.xtile(self._cache, *args, **kw)

    def better(self, log2):
        if log2 is None:
            return ValueError
        if not self._cache or not log2._cache:
            return False
        if self.median() < log2.median():
            return True
        if self.iqr() < log2.iqr():
            return True
        return False
```

Oct 15, 14 16:45 **"csc710sbse: hw6: Witschey"** Page 1/1

```
from model import Model
from independent_variable import IndependentVariable
from schaffer import Schaffer
from kursawe import Kursawe
from fonseca import Fonseca
from zdt1 import ZDT1
from zdt3 import ZDT3
from viennet3 import Viennet3
from dtlz7 import DTLZ7
from schwefel import Schwefel

__all__ = [Model, IndependentVariable, Schaffer, Kursawe, Fonseca,
           ZDT1, ZDT3, Viennet3, DTLZ7, Schwefel]
```

Oct 18, 14 18:59 **"csc710sbse: hw6: Witschey"** Page 1/1

```
from __future__ import division, print_function

# all adapted from Dr. Tim Menzies' model code:
# https://github.com/timm/sbse14/blob/master/models.py

from abc import ABCMeta
from collections import namedtuple


ModelIO = namedtuple('ModelIO', ('xs', 'ys', 'energy'))


class Model(object):
    # allows us to get all subclasses with __subclasses__()
    __metaclass__ = ABCMeta

    def __init__(self, independents=None, dependents=None,
                 energy_min=None, energy_max=None,
                 enforce_energy_constraints=False):
        if independents is None or dependents is None:
            raise ValueError

        self.xs = independents
        self.ys = dependents
        self.energy_max = energy_max
        self.energy_min = energy_min
        self.enforce_energy_constraints = enforce_energy_constraints

    def normalize(self, x):
        n = x - self.energy_min
        d = self.energy_max - self.energy_min
        try:
            return n / d
        except ZeroDivisionError:
            return 0.5

    def random_input_vector(self):
        return tuple(x() for x in self.xs)

    def __call__(self, xs, io=False):
        ys = tuple(y(xs) for y in self.ys)
        energy = sum(ys)

        if self.enforce_energy_constraints:
            energy_errmsg = 'current energy {} not in range [{}, {}]'.format(
                energy, self.energy_min, self.energy_max)

        if self.energy_min is None or self.energy_min > energy:
            if self.enforce_energy_constraints:
                raise ValueError(energy_errmsg)
            self.energy_min = energy

        if self.energy_max is None or energy > self.energy_max:
            if self.enforce_energy_constraints:
                raise ValueError(energy_errmsg)
            self.energy_max = energy

        if io:
            return ModelIO(xs, ys, energy)

        return ys

    def energy(self, ys, norm=False):
        rv = sum(ys)
        return self.normalize(rv) if norm else rv

    def compute_model_io(self, xs):
        ys = self(xs)
        return ModelIO(xs, ys, self.energy(ys))

    def random_model_io(self):
        return self.compute_model_io(self.random_input_vector())
```

```
from __future__ import division, print_function
# all adapted from Dr. Tim Menzies' model code:
# https://github.com/timm/sbse14/blob/master/models.py

import random


class IndependentVariable(object):
    def __init__(self, lo=None, hi=None, type=float):
        self.lo = lo
        self.hi = hi
        self.type = type

    def __call__(self):
        if self.type == float:
            f = random.uniform
        elif self.type == int:
            f = random.randint

        return f(self.lo, self.hi)

    def clip(self, x):
        return max(self.lo, min(self.hi, x))
```



```
# all adapted from Dr. Tim Menzies' model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division
import math

from model import Model
from independent_variable import IndependentVariable as IV  # noqa


class DTLZ7(Model):
    def __init__(self, ivs=30, dvs=20):

        # dynamically generate these suckers
        # h/t http://stackoverflow.com/a/13184536/3408454
        generated_fs = []
        for x in xrange(1, dvs):
            f = lambda xs: xs[x]
            f.__name__ = 'f{}'.format(x)
            generated_fs.append(f)

        def g(xs):
            # avoid divide by 0 errors
            denom = abs(xs[-1]) or .0001
            return 1 + (9 / denom) * sum(xs)

        def h(xs, fs=generated_fs, g=g):
            s = 0
            for f in fs:
                fxs = f(xs)
                a = fxs / (1 + g(xs))
                b = 1 + math.sin(3 * math.pi * fxs)
                s += a * b

            return dvs - s

        def final_f(xs):
            return (1 + g(xs)) * h(xs)
        final_f.__name__ = 'f{}'.format(dvs)

        fs = tuple(generated_fs + [final_f])

        independents = tuple(IV(lo=0, hi=1) for _ in xrange(ivs))
        super(DTLZ7, self).__init__(independents=independents, dependents=fs)
```

```
# all adapted from Dr. Tim Menzies' model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division
import math

from model import Model
from independent_variable import IndependentVariable as IV  # noqa
from witschey.base import memo_sqrt


class Fonseca(Model):
    def __init__(self, ivs=3):
        ivs = tuple(IV(lo=-4, hi=4) for _ in xrange(ivs - 1))

        def f1(xs):
            e = sum((x - (1 / memo_sqrt(i+1))) ** 2 for i, x in enumerate(xs))
            return 1 - math.exp(-e)

        def f2(xs):
            e = sum((x + (1 / memo_sqrt(i+1))) ** 2 for i, x in enumerate(xs))
            return 1 - math.exp(-e)

        super(Fonseca, self).__init__(independents=ivs, dependents=(f1, f2))
```



```
# all adapted from Dr. Tim Menzies' model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division
import math

from model import Model
from independent_variable import IndependentVariable as IV  # noqa


class Kursawe(Model):
    def __init__(self, ivs=3, a=0.8, b=3):
        ivs = tuple(IV(lo=-5, hi=5) for _ in xrange(ivs - 1))
        self.a = a
        self.b = b

        def f1(xs):
            rv = 0
            for i in xrange(len(xs) - 1):
                exponent = (-0.2) * math.sqrt(xs[i] ** 2 + xs[i+1] ** 2)
                rv += -10 * math.exp(exponent)
            return rv

        def f2(xs):
            f = lambda x: (math.fabs(x)**self.a) + (5 * math.sin(x)**self.b)
            return sum(f(x) for x in xs)

        super(Kursawe, self).__init__(independents=ivs, dependents=(f1, f2))
```

```
# all adapted from Dr. Tim Menzies' model code:
# https://github.com/timm/sbse14/blob/master/models.py

from model import Model
from independent_variable import IndependentVariable as IV  # noqa


class Schaffer(Model):

    def __init__(self, ivs=1):
        independents = tuple(IV(lo=-10 ** 5, hi=10 ** 5) for _ in xrange(ivs))

        # use def instead of lambdas so the functions keep their __name__s
        def f1(xs):
            return sum(x ** 2 for x in xs)

        def f2(xs):
            return sum((x - 2) ** 2 for x in xs)

        super(Schaffer, self).__init__(
            independents=independents, dependents=(f1, f2))
```



```
# all adapted from Dr. Tim Menzies' model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division
import math

from model import Model
from independent_variable import IndependentVariable as IV  # noqa


class Viennet3(Model):

    def __init__(self):

        def f1(xs):
            x_1sq = xs[0] ** 2
            x_2sq = xs[1] ** 2
            a = 0.5 * x_1sq
            b = math.sin(x_1sq + x_2sq)
            return a + x_2sq + b

        def f2(xs):
            x_1 = xs[0]
            x_2 = xs[1]

            a = ((3 * x_1 - 2 * x_2 + 4) ** 2) / 8
            b = ((x_1 + x_2 + 1) ** 2) / 27

            return a + b + 15

        def f3(xs):
            x_1sq = xs[0] ** 2
            x_2sq = xs[1] ** 2

            a = 1 / (x_1sq + x_2sq + 1)
            b = 1.1 * math.exp(-x_1sq - x_2sq)

            return a - b

        ivs = (IV(lo=-3, hi=3), IV(lo=-3, hi=3))

        super(Viennet3, self).__init__(
            independents=ivs, dependents=(f1, f2, f3))
```

Oct 15, 14 20:18 **"csc710sbse: hw6: Witschey"** Page 1/1

```
# all adapted from Dr. Tim Menzies' model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division
import math

from model import Model
from independent_variable import IndependentVariable as IV  # noqa


class ZDT1(Model):
    def __init__(self, ivs=30):

        def g(xs):
            return 1 + 9 * sum(xs[1:]) / (len(xs) - 1)

        def f1(xs):
            return xs[0]

        def f2(xs):
            gxs = g(xs)
            return gxs * (1 - math.sqrt(xs[0] / gxs))

        ivs = tuple(IV(lo=0, hi=1) for _ in xrange(30))
        super(ZDT1, self).__init__(independents=ivs, dependents=(f1, f2, g))
```

Oct 15, 14 20:28 **"csc710sbse: hw6: Witschey"** Page 1/1

```
# all adapted from Dr. Tim Menzies' model code:
# https://github.com/timm/sbse14/blob/master/models.py

from __future__ import division

import math

from model import Model
from independent_variable import IndependentVariable as IV  # noqa
from witschey.base import memo_sqrt


class ZDT3(Model):

    def __init__(self, ivs=30):

        def g(xs):
            return 1 + 9 * sum(xs[1:]) / (len(xs) - 1)

        def f1(xs):
            return xs[0]

        def f2(xs):
            gxs = g(xs)
            a = 1 - memo_sqrt(xs[0] / gxs) - (xs[0] / gxs)
            a *= math.sin(10 * math.pi * xs[0])
            return gxs * a

        ivs = tuple(IV(lo=0, hi=1) for _ in xrange(30))

        super(ZDT3, self).__init__(independents=ivs, dependents=(f1, f2, g))
```

```
from __future__ import division, unicode_literals

from datetime import datetime
import abc
from types import NoneType
from collections import namedtuple, OrderedDict

from witschey.base import memo
from witschey.models import Model
from witschey.config import CONFIG


class Searcher(object):
    # allows us to get all subclasses with __subclasses__()
    __metaclass__ = abc.ABCMeta

    def __new__(cls, *args, **kwargs):
        # construct our object
        future_self = super(Searcher, cls).__new__(cls, *args, **kwargs)

        name = cls.__name__
        # initialize a dict with searcher's name
        # and the initialization time
        d = dict(searcher=name, initialized=datetime.now())

        # if there are global options for this class or its bases in CONFIG
        for k in [name] + [k.__name__ for k in cls.__bases__]:
            if hasattr(CONFIG, k):
                # add them to the dict
                d.update(getattr(CONFIG, k).__dict__)

        # then, add the kwargs to the constructor call to the dict.
        # NB: this happens after adding options from The, so
        #     call-specific options override the globals
        d.update(kwargs)

        # set our spec with the contents of the dict
        future_self.spec = memo(**d)

        return future_self

    def __init__(self, model, *args, **kw):
        self.model = model()

    def run(*args, **kwargs):
        raise NotImplementedError()


class SearcherConfig(object):

    def __init__(self, searcher=None, model=None, **kwargs):
        self.searcher, self.model = searcher, model
        self._kw_dict = kwargs

    def get_searcher(self, searcher=None, model=None, **kwargs):
        s = searcher or self.searcher
        m = model or self.model
        kw = self._kw_dict.copy().update(kwargs) or {}
        return s(m, **kw)

    @property
    def searcher(self):
        return self._searcher

    @searcher.setter
    def searcher(self, value):
        if isinstance(value, NoneType) or issubclass(value, Searcher):
            self._searcher = value
        else:
            raise TypeError('{} is not a Searcher or None'.format(value))

    @property
    def model(self):
```



```
        return self._model

    @model.setter
    def model(self, value):
        if isinstance(value, NoneType) or issubclass(value, Model):
            self._model = value
        else:
            raise TypeError('{} is not a Model or None'.format(value))

    def update(self, searcher=None, model=None, **kwargs):
        if searcher is not None:
            self.searcher = searcher
        if model is not None:
            self.model = model
        self._kw_dict.update(kwargs)

    def as_dict(self):
        "returns a OrderedDict with the searcher and model first"
        return OrderedDict(searcher=self._searcher,
                           model=self._model, **self._kw_dict)

    def __repr__(self):
        kw_string = ', '.join('{0}={1}'.format(k, v)
                              for k, v in self.as_dict().iteritems())
        return '{0}({1})'.format(self.__class__.__name__, kw_string)


SearchReport = namedtuple('SearchReport',
                          ['best', 'best_era', 'evaluations', 'searcher',
                           'spec'])
```

```
from __future__ import division, print_function

import random
import math

from searcher import Searcher, SearchReport
from witschey.base import NullObject, StringBuilder
from witschey.log import NumberLog


class SimulatedAnnealer(Searcher):
    def __init__(self, model, *args, **kw):
        super(SimulatedAnnealer, self).__init__(model=model, *args, **kw)

    def _get_neighbor(self, model_io):
        n_gen = (model_io.xs[i]
                 if random.random() < self.spec.p_mutation else v
                 for i, v in enumerate(self.model.random_input_vector()))
        return self.model(tuple(n_gen), io=True)

    def run(self, text_report=True):
        report = StringBuilder() if text_report else NullObject()
        current = self.model.random_model_io()
        best = current  # assumes current is immutable
        self.lives = 4
        current_era_energies = NumberLog(max_size=None)
        best_era = None
        evals = None

        for k in range(self.spec.iterations):
            if self.lives <= 0 and self.spec.terminate_early:
                evals = k
                break
            prev_era_energies = current_era_energies

            neighbor = self._get_neighbor(current)
            current_era_energies += neighbor.energy

            if neighbor.energy < best.energy:
                best, current = neighbor, neighbor
                report += '!'

            if neighbor.energy < current.energy:
                current = neighbor
                report += '+'
            else:
                cnorm = self.model.normalize(current.energy)
                nnorm = self.model.normalize(neighbor.energy)
                temp = k / self.spec.iterations
                if self._good_idea(cnorm, nnorm, temp) < random.random():
                    current = neighbor
                    report += '?'

            report += '.'

            if k % self.spec.era_length == 0 and k != 0:
                report += ('\n', '{: .2}'.format(best.energy), ' ')
                if not best_era:
                    best_era = current_era_energies

                try:
                    improved = current_era_energies.better(prev_era_energies)
                except ValueError:
                    improved = False
                if improved:
                    best_era = current_era_energies
                else:
                    self.lives -= 1
                current_era_energies = NumberLog()

        if evals is None:
            evals = self.spec.iterations
        rv = SearchReport(best=best.energy, evaluations=evals,
```



```
                          best_era=best_era, spec=self.spec,
                          searcher=self.__class__)
        return rv

    def _good_idea(self, old, new, temp):
        """
        sets the threshold we compare to to decide whether to jump

        returns e^-((new-old)/temp)
        """
        numerator = new - old

        if not 0 <= numerator <= 1:
            numerator = old - new
        try:
            exponent = numerator / temp
        except ZeroDivisionError:
            return 0
        rv = math.exp(-exponent)
        if rv > 1:
            raise ValueError('p returning greater than one',
                             rv, old, new, temp)
        return rv * self.spec.cooling_factor
```

```
from __future__ import division

import random
import numpy as np

from searcher import Searcher, SearchReport
from witschey import base
from witschey.base import tuple_replace, StringBuilder, NullObject
from witschey.log import NumberLog


class MaxWalkSat(Searcher):

    def __init__(self, model, *args, **kw):
        super(MaxWalkSat, self).__init__(model=model, *args, **kw)

    def _local_search_xs(self, bottom, top, n=10):
        '''divide the space from bottom to top into n partitions, then
        randomly sample within each partition'''
        chunk_length = (top - bottom) / n

        for a in np.arange(bottom, top, chunk_length):
            yield random.uniform(a, a + chunk_length)

    def _update(self, improvement_char, dimension=None, value=None):
        '''calculate the next value from the model and update state as
        necessary'''
        # check for invalid input
        if value is not None and dimension is None:
            err = 'cannot call _update with specified value but no dimension'
            raise ValueError(err)

        if dimension is None:
            dimension = base.random_index(self._current.xs)
        if value is None:
            # get random value if no value input
            value = self.model.xs[dimension]()

        # generate and evaluate input vector
        new_xs = tuple_replace(self._current.xs, dimension, value)
        self._current = self.model(new_xs, io=True)
        self._evals += 1
        self._current_era += self._current.energy

        # compare to previous best and update as necessary
        if self._current.energy < self._best.energy:
            self._best = self._current
            self._report += improvement_char
        else:
            self._report += '.'

    def run(self, text_report=True):
        '''run MaxWalkSat on self.model'''

        # current ModelIO to evaluate and mutate
        self._current = self.model.random_model_io()
        self._best = self._current
        # initialize and update log variables to track values by era
        self._current_era = NumberLog()
        self._current_era += self._current.energy
        best_era = None
        # bookkeeping variables
        self._evals = 0
        lives = 4
        self._report = StringBuilder() if text_report else NullObject()
        terminate = False

        while self._evals < self.spec.iterations and not terminate:
            # get the generator for a random independent variable

            if self.spec.p_mutation > random.random():
                # if not searching a dimension, mutate randomly
                self._update('+')
```



```
            else:
                # if doing a local search, choose a dimension
                dimension = base.random_index(self._current.xs)
                search_iv = self.model.xs[dimension]
                # make sure local search ends at era end
                max_search = self.spec.era_length - (self._evals
                                                     % self.spec.era_length)
                n = min(10, max_search)
                # then try points all along the dimension
                lo, hi = search_iv.lo, search_iv.hi
                for j in self._local_search_xs(lo, hi, n):
                    self._update('|', dimension=dimension, value=j)

            # end-of-era bookkeeping
            if self._evals % self.spec.era_length == 0:
                self._report += ('\n{: .2}'.format(self._best.energy), ' ')

                # _prev_era won't exist in era 0, so account for that case
                try:
                    improved = self._current_era.better(self._prev_era)
                except AttributeError:
                    improved = False
                self._prev_era = self._current_era

                # track best_era
                if improved or best_era is None:
                    best_era = self._current_era
                else:
                    lives -= 1

                if lives <= 0:
                    terminate = True
                else:
                    self._current_era = NumberLog()

        return SearchReport(best=self._best.energy,
                            best_era=best_era,
                            evaluations=self._evals,
                            searcher=self.__class__,
                            spec=self.spec)
```

```
from __future__ import division, print_function

import itertools
import random

from witschey import base
from searcher import Searcher, SearchReport
from witschey.log import NumberLog

# adapted from Chris Theisen's code
#     his code provided the shell that I worked in and styled to my liking
# Structure from:
# www.cleveralgorithms.com/nature-inspired/evolution/genetic_algorithm.html


class GeneticAlgorithm(Searcher):

    def __init__(self, model, *args, **kw):
        super(GeneticAlgorithm, self).__init__(model=model, *args, **kw)

    def mutate(self, child):
        i = base.random_index(child)
        return base.tuple_replace(child, i, self.model.xs[i]())

    def crossover(self, parent1, parent2, xovers=1):
        if len(parent1) != len(parent2):
            raise ValueError('parents must be same length to breed')
        if len(parent1) == 1:
            return random.choice((parent1, parent2))

        if xovers < 1:
            raise ValueError('cannot have fewer than 1 crossover')
        xovers = min(len(parent1) - 2, xovers)
        xovers = sorted(random.sample(xrange(1, len(parent1) - 1), xovers))
        x_pts = itertools.chain((0,), xovers, (None,))

        cycle_parents = itertools.cycle((parent1, parent2))
        parent_point_zip = itertools.izip(cycle_parents, base.pairs(x_pts))

        segments = [itertools.islice(parent, p[0], p[1])
                    for parent, p in parent_point_zip]

        return tuple(itertools.chain(*segments))

    def select_parents(self, population, output_size):
        """generates all possible parent pairs from population, clipped to
        output_size
        """
        fore = itertools.combinations(population, 2)
        back = itertools.combinations(reversed(population), 2)
        all_parents = set(fore).union(set(back))
        if len(all_parents) < output_size:
            return all_parents
        return random.sample(all_parents, output_size)

    def run(self, text_report=True):
        rand_vect = lambda: self.model.random_input_vector()
        pop_size = self.spec.population_size
        init_xs = tuple(rand_vect() for _ in xrange(pop_size))
        energy = lambda x: x.energy
        best_era = None

        report = base.StringBuilder() if text_report else base.NullObject()

        population = tuple(self.model.compute_model_io(xs) for xs in init_xs)

        best = min(population, key=energy)

        evals, lives = 0, 4

        for gen in xrange(self.spec.iterations):
            if evals > self.spec.iterations or lives <= 0:
                break
```



```
            children = []
            for parent1, parent2 in self.select_parents(population, pop_size):
                xs = self.crossover(parent1.xs, parent2.xs, 2)
                if random.random() < self.spec.p_mutation:
                    self.mutate(xs)
                child = self.model(xs, io=True)
                children.append(child)

            best_in_pop = min(children, key=energy)

            prev_best_energy = best.energy
            best = min(best, best_in_pop, key=energy)

            report += str(best.energy)
            report += ('+' if x.energy < prev_best_energy else '.'
                       for x in children)
            report += '\n'

            population = children
            evals += len(population)

            energies = NumberLog(inits=(c.energy for c in children))
            try:
                improved = energies.better(prev_energies)
            except NameError:
                improved = False
            prev_energies = energies  # noqa: flake8 doesn't catch use above

            if improved:
                best_era = energies
            else:
                lives -= 1

        if best_era is None:
            best_era = energies

        return SearchReport(best=best.energy,
                            best_era=best_era,
                            evaluations=evals,
                            searcher=self.__class__,
                            spec=self.spec)
```

```
from __future__ import division
import sys
import random
import math

import texttable
from basic_stats import xtile, median
from witschey import base

# flake8: noqa

"""

### Standard Accumulator for Numbers

Note the _lt_ method: this accumulator can be sorted by median values.

Warning: this accumulator keeps _all_ numbers. Might be better to use
a bounded cache.

"""
class Num:
  "An Accumulator for numbers"
  def __init__(i,name,inits=[]):
    i.n = i.m2 = i.mu = 0.0
    i.all=[]
    i._median=None
    i.name = name
    i.rank = 0
    for x in inits: i.add(x)
  def s(i)       : return (i.m2/(i.n - 1))**0.5
  def add(i,x):
    i._median=None
    i.n   += 1
    i.all += [x]
    delta  = x - i.mu
    i.mu  += delta*1.0/i.n
    i.m2  += delta*(x - i.mu)
  def __add__(i,j):
    return Num(i.name + j.name,i.all + j.all)
  def quartiles(i):
    i.all = sorted(i.all)
    n  = int(len(i.all)*0.25)
    return i.all[n] , i.all[n * 2], i.all[n * 3]
  def median(i):
    if not i._median:
      i.all = sorted(i.all)
      i._median=median(i.all)
    return i._median
  def __lt__(i,j):
    return i.median() < j.median()
  def spread(i):
    i.all=sorted(i.all)
    n1=i.n*0.25
    n2=i.n*0.75
    if len(i.all) <= 1:
      return 0
    if len(i.all) == 2:
      return i.all[1] - i.all[0]
    else:
      return i.all[int(n2)] - i.all[int(n1)]


def a12(lst1,lst2):
    "how often is x in lst1 more than y in lst2?"
    def loop(t,t1,t2):
        while t1.j < t1.n and t2.j < t2.n:
            h1 = t1.l[t1.j]
            h2 = t2.l[t2.j]
            h3 = t2.l[t2.j+1] if t2.j+1 < t2.n else None
            if h1>  h2:
                t1.j  += 1; t1.gt += t2.n - t2.j
            elif h1 == h2:
```



```
                if h3 and h1 > h3 :
                    t1.gt += t2.n - t2.j  - 1
                t1.j  += 1; t1.eq += 1; t2.eq += 1
            else:
                t2,t1  = t1,t2
        return t.gt*1.0, t.eq*1.0
    #--------------------------
    lst1 = sorted(lst1, reverse=True)
    lst2 = sorted(lst2, reverse=True)
    n1   = len(lst1)
    n2   = len(lst2)
    t1   = base.memo(l=lst1,j=0,eq=0,gt=0,n=n1)
    t2   = base.memo(l=lst2,j=0,eq=0,gt=0,n=n2)
    gt,eq= loop(t1, t1, t2)
    return gt/(n1*n2) + eq/2/(n1*n2)


"""## Non-Parametric Hypothesis Testing

The following _bootstrap_ method was introduced in
1979 by Bradley Efron at Stanford University. It
was inspired by earlier work on the
jackknife.
Improved estimates of the variance were [developed later][efron01].

[efron01]: http://goo.gl/14n8Wf "Bradley Efron and R.J. Tibshirani. An Introduct
ion to the Bootstrap (Chapman & Hall/CRC Monographs on Statistics & Applied Prob
ability), 1993"


To check if two populations _(y0,z0)_
are different, many times sample with replacement
from both to generate _(y1,z1), (y2,z2), (y3,z3)_.. etc.

"""
def sampleWithReplacement(lst):
  "returns a list same size as list"
  def any(n)  : return random.uniform(0,n)
  def one(lst): return lst[ int(any(len(lst))) ]
  return [one(lst) for _ in lst]
"""



Then, for all those samples,
 check if some *testStatistic* in the original pair
hold for all the other pairs. If it does more than (say) 99%
of the time, then we are 99% confident in that the
populations are the same.

In such a _bootstrap_ hypothesis test, the *some property*
is the difference between the two populations, muted by the
joint standard deviation of the populations.

"""
def testStatistic(y,z):
    """Checks if two means are different, tempered
     by the sample size of 'y' and 'z'"""
    tmp1 = tmp2 = 0
    for y1 in y.all: tmp1 += (y1 - y.mu)**2
    for z1 in z.all: tmp2 += (z1 - z.mu)**2
    s1    = (float(tmp1)/(y.n - 1))**0.5
    s2    = (float(tmp2)/(z.n - 1))**0.5
    delta = z.mu - y.mu
    if s1+s2:
      delta =  delta/((s1/y.n + s2/z.n)**0.5)
    return delta
"""

The rest is just details:

+ Efron advises
  to make the mean of the populations the same (see
  the _yhat,zhat_ stuff shown below).
```

```
+ The class _total_ is a just a quick and dirty accumulation class.
+ For more details see [the Efron text][efron01].

"""
def bootstrap(y0,z0,conf=0.01,b=1000):
  """The bootstrap hypothesis test from
     p220 to 223 of Efron's book 'An
    introduction to the boostrap."""
  class total():
    "quick and dirty data collector"
    def __init__(i,some=[]):
      i.sum = i.n = i.mu = 0 ; i.all=[]
      for one in some: i.put(one)
    def put(i,x):
      i.all.append(x);
      i.sum +=x; i.n += 1; i.mu = float(i.sum)/i.n
    def __add__(i1,i2): return total(i1.all + i2.all)
  y, z   = total(y0), total(z0)
  x      = y + z
  tobs   = testStatistic(y,z)
  yhat   = [y1 - y.mu + x.mu for y1 in y.all]
  zhat   = [z1 - z.mu + x.mu for z1 in z.all]
  bigger = 0.0
  for i in range(b):
    if testStatistic(total(sampleWithReplacement(yhat)),
                     total(sampleWithReplacement(zhat))) > tobs:
      bigger += 1
  return bigger / b < conf

def different(l1,l2):
  #return bootstrap(l1,l2) and a12(l2,l1)
  return a12(l2,l1) and bootstrap(l1,l2)

"""

## Saner Hypothesis Testing

The following code, which you should use verbatim does the following:


+ All treatments are clustered into _ranks_. In practice, dozens
  of treatments end up generating just a handful of ranks.
+ The numbers of calls to the hypothesis tests are minimized:
    + Treatments are sorted by their median value.
    + Treatments are divided into two groups such that the
      expected value of the mean values _after_ the split is minimized;
    + Hypothesis tests are called to test if the two groups are truly difference
.
          + All hypothesis tests are non-parametric and include (1) effect size
tests
            and (2) tests for statistically significant numbers;
          + Slow bootstraps are executed  if the faster _A12_ tests are passed;

In practice, this means that the hypothesis tests (with confidence of say, 95%)
are called on only a logarithmic number of times. So...

+ With this method, 16 treatments can be studied using less than _&sum;<sub>1,2,
4,8,16</sub>log<sub>2</sub>i =15_ hypothesis tests  and confidence _0.99<sup>15<
/sup>=0.86_.
+ But if did this with the 120 all-pairs comparisons of the 16 treatments, we wo
uld have total confidence _0.99<sup>120</sup>=0.30.

For examples on using this code, see _rdivDemo_ (below).

"""
def scottknott(data,cohen=0.3,small=3,epsilon=0.01):
    """Recursively split data, maximizing delta of
    the expected value of the mean before and
    after the splits.
    Reject splits with under 3 items"""
    all  = reduce(lambda x,y:x+y,data)
    same = lambda l, r:   not different(l.all,r.all)
    big  = lambda    n: n > small
    return rdiv(data,all,minMu,big,same,epsilon)

def rdiv(data,  # a list of class Nums
         all,   # all the data combined into one num
         div,   # function: find the best split
         big,   # function: rejects small splits
         same, # function: rejects similar splits
         epsilon): # small enough to split two parts
    """Looks for ways to split sorted data,
    Recurses into each split. Assigns a 'rank' number
    to all the leaf splits found in this way.
    """
    def recurse(parts,all,rank=0):
        "Split, then recurse on each part."
        cut,left,right = maybeIgnore(div(parts,all,big,epsilon),
                                     same,parts)
        if cut:
            # if cut, rank "right" higher than "left"
            rank = recurse(parts[:cut],left,rank) + 1
            rank = recurse(parts[cut:],right,rank)
        else:
            # if no cut, then all get same rank
            for part in parts:
                part.rank = rank
        return rank
    recurse(sorted(data),all)
    return data

def maybeIgnore((cut,left,right), same,parts):
    if cut:
        if same(sum(parts[:cut],Num('upto')), sum(parts[cut:],Num('above'))):
            cut = left = right = None
    return cut,left,right

def minMu(parts,all,big,epsilon):
    """Find a cut in the parts that maximizes
    the expected value of the difference in
    the mean before and after the cut.
    Reject splits that are insignificantly
    different or that generate very small subsets.
    """
    cut,left,right = None,None,None
    before, mu     =  0, all.mu
    for i,l,r in leftRight(parts,epsilon):
        if big(l.n) and big(r.n):
            n   = all.n * 1.0
            now = l.n/n*(mu- l.mu)**2 + r.n/n*(mu- r.mu)**2
            if now > before:
                before,cut,left,right = now,i,l,r
    return cut,left,right

def leftRight(parts,epsilon=0.01):
    """Iterator. For all items in 'parts',
    return everything to the left and everything
    from here to the end. For reasons of
    efficiency, take a first pass over the data
    to pre-compute and cache right-hand-sides
    """
    rights = {}
    n = j = len(parts) - 1
    while j > 0:
        rights[j] = parts[j]
        if j < n: rights[j] += rights[j+1]
        j -=1
    left = parts[0]
    for i,one in enumerate(parts):
        if i> 0:
            if parts[i]._median - parts[i-1]._median > epsilon:
                yield i,left,rights[i]
            left += one
"""

## Putting it All Together
```

```
    Driver for the demos:

    """
290 def rdiv_report(data):
        rows = []
        def z(x):
            return int(100 * (x - lo) / (hi - lo + 0.00001))
        data = map(lambda lst:Num(lst[0],lst[1:]),
295                data)
        ranks=[]
        for x in scottknott(data):
            ranks += [(x.rank,x.median(),x)]
        all=[]
300     for _,__,x in sorted(ranks): all += x.all
        all = sorted(all)
        lo, hi = all[0], all[-1]
        last = None
        rows.append(['rank', 'name', 'med', 'iqr', '',
305                  '10%', '30%', '50%', '70%', '90%'])
        for _,__,x in sorted(ranks):
            q1,q2,q3 = (round(q, 2) for q in x.quartiles())
            xtile_out = xtile(x.all,lo=lo,hi=hi,width=30,show="%5.2f", as_list=True)
            row_xtile = [xtile_out[0]] + map(lambda x: x + ',', xtile_out[1:-1]) +\
310                      [xtile_out[-1]]
            rows.append([x.rank+1] +
              map(lambda y: str(y) + ',', [x.name, q2]) + [q3 - q1] + row_xtile)
            last = x.rank
        table = texttable.Texttable(200)
315     table.set_cols_dtype(['t', 't', 't', 't', 't', 't', 't', 't', 't', 't'])
        table.set_cols_align(['r', 'l', 'r', 'r', 'c', 'r', 'r', 'r', 'r', 'r'])
        table.set_deco(texttable.Texttable.HEADER)
        table.add_rows(rows)
        return table.draw()
```