Parameter Heatmap¶

This tutorial will show how to optimize strategies with multiple parameters and how to examine and reason about optimization results. It is assumed you're already familiar with basic *backtesting.py* usage.

First, let's again import our helper moving average function. In practice, one should use functions from an indicator library, such as TA-Lib or Tulipy.

In [1]:

```
from backtesting.test import SMA
```

Loading BokehJS ...

Our strategy will be a similar moving average cross-over strategy to the one in Quick Start User Guide, but we will use four moving averages in total: two moving averages whose relationship determines a general trend (we only trade long when the shorter MA is above the longer one, and vice versa), and two moving averages whose cross-over with daily *close* prices determine the signal to enter or exit the position.

In [2]:

```
from backtesting import Strategy from backtesting.lib import crossover class Sma4Cross(Strategy): n1 =
50 n2 = 100 n_enter = 20 n_exit = 10 def init(self): self.sma1 = self.I(SMA, self.data.Close, self.n1)
self.sma2 = self.I(SMA, self.data.Close, self.n2) self.sma_enter = self.I(SMA, self.data.Close,
self.n_enter) self.sma_exit = self.I(SMA, self.data.Close, self.n_exit) def next(self): if not
self.position: # On upwards trend, if price closes above # "entry" MA, go long # Here, even though the
operands are arrays, this # works by implicitly comparing the two last values if self.sma1 > self.sma2:
if crossover(self.data.Close, self.sma_enter): self.buy() # On downwards trend, if price closes below #
"entry" MA, go short else: if crossover(self.sma_enter, self.data.Close): self.sell() # But if we
already hold a position and the price # closes back below (above) "exit" MA, close the position else: if
(self.position.is_long and crossover(self.sma_exit, self.data.Close) or self.position.is_short and
crossover(self.data.Close, self.sma_exit)): self.position.close()
```

It's not a robust strategy, but we can optimize it.

Grid search is an exhaustive search through a set of specified sets of values of hyperparameters. One evaluates the performance for each set of parameters and finally selects the combination that performs best.

Let's optimize our strategy on Google stock data using *randomized* grid search over the parameter space, evaluating at most (approximately) 200 randomly chosen combinations:

In [3]:

```
%%time from backtesting import Backtest from backtesting.test import GOOG backtest = Backtest(GOOG,
Sma4Cross, commission=.002) stats, heatmap = backtest.optimize( n1=range(10, 110, 10), n2=range(20, 210,
20), n_enter=range(15, 35, 5), n_exit=range(10, 25, 5), constraint=lambda p: p.n_exit < p.n_enter < p.n1
< p.n2, maximize='Equity Final [$]', max_tries=200, random_state=0, return_heatmap=True)
```

```
CPU times: user 157 ms, sys: 44.4 ms, total: 202 ms Wall time: 5.93 s
```

Notice `return_heatmap=True` parameter passed to `Backtest.optimize()`. It makes the function return a heatmap series along with the usual stats of the best run. `heatmap` is a pandas Series indexed with a MultiIndex, a cartesian product of all permissible (tried) parameter values. The series values are from the `maximize=` argument we provided.

In [4]:

```
heatmap
```

Out[4]:

```
n1 n2 n_enter n_exit 20 60 15 10 8448.64 80 15 10 8348.38 100 15 10 9283.02 30 40 20 15 10331.51 25 15
14441.14 ... 100 200 15 10 10994.65 20 10 9736.19 15 14234.93 25 10 7732.77 30 10 8806.70 Name: Equity
Final [$], Length: 177, dtype: float64
```

This heatmap contains the results of all the runs, making it very easy to obtain parameter combinations for e.g. three best runs:

In [5]:

```
heatmap.sort_values().iloc[-3:]
```

Out[5]:

```
n1 n2 n_enter n_exit 40 60 25 20 15618.15 100 160 20 15 16600.30 50 160 20 15 17149.91 Name: Equity
Final [$], dtype: float64
```

But we use vision to make judgements on larger data sets much faster. Let's plot the whole heatmap by projecting it on two chosen dimensions. Say we're mostly interested in how parameters `n1` and `n2`, on average, affect the outcome.

In [6]:

```
hm = heatmap.groupby(['n1', 'n2']).mean().unstack() hm = hm[::-1] hm
```

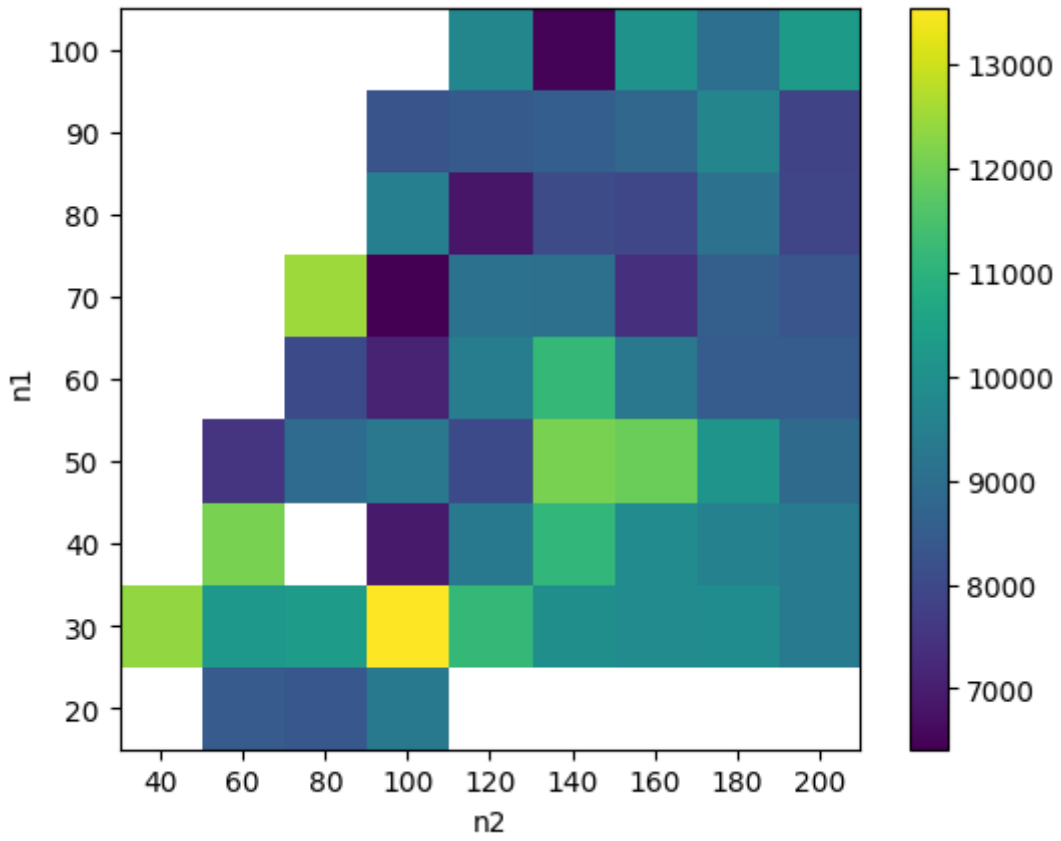
Out[6]:

n2	40	60	80	100	120	140	160	180	200
n1									
100	NaN	NaN	NaN	NaN	9694.92	6494.40	10063.31	9010.02	10301.05
90	NaN	NaN	NaN	8275.00	8427.48	8540.80	8771.71	9646.89	7884.19
80	NaN	NaN	NaN	9477.03	6846.68	8042.73	7919.79	9096.51	7900.53
70	NaN	NaN	12498.79	6417.25	9069.63	9034.70	7384.79	8576.84	8286.39
60	NaN	NaN	8027.70	7129.25	9436.62	11145.88	9273.17	8484.34	8491.34
50	NaN	7501.19	8909.36	9270.71	8017.69	12118.29	11939.28	10091.10	8868.40
40	NaN	12103.34	NaN	6930.31	9295.27	11125.64	9847.44	9538.79	9331.12
30	12386.32	10224.44	10360.19	13537.82	11160.57	9967.15	9841.65	9882.30	9331.49
20	NaN	8448.64	8348.38	9283.02	NaN	NaN	NaN	NaN	NaN

Let's plot this table as a heatmap:

In [7]:

```
%matplotlib inline import matplotlib.pyplot as plt fig, ax = plt.subplots() im = ax.imshow(hm,
cmap='viridis') _ = ( ax.set_xticks(range(len(hm.columns)), labels=hm.columns),
ax.set_yticks(range(len(hm)), labels=hm.index), ax.set_xlabel('n2'), ax.set_ylabel('n1'),
ax.figure.colorbar(im, ax=ax), )
```



We see that, on average, we obtain the highest result using trend-determining parameters `n1=30` and `n2=100` or `n1=70` and `n2=80`, and it's not like other nearby combinations work similarly well — for our particular strategy, these combinations really stand out.

Since our strategy contains several parameters, we might be interested in other relationships between their values. We can use `backtesting.lib.plot_heatmaps()` function to plot interactive heatmaps of all parameter combinations simultaneously.

In [8]:

```
from backtesting.lib import plot_heatmaps plot_heatmaps(heatmap, agg='mean')
```

Out[8]:

GridPlot(id = 'p1292', …)

Model-based optimization¶

Above, we used *randomized grid search* optimization method. Any kind of grid search, however, might be computationally expensive for large data sets. In the follwing example, we will use *SAMBO Optimization* package to guide our optimization better informed using forests of decision trees. The hyperparameter model is sequentially improved by evaluating the expensive function (the backtest) at the next best point, thereby hopefully converging to a set of optimal parameters with **as few evaluations as possible**.

So, with `method="sambo"`:

In [9]:

```
%%capture ! pip install sambo # This is a run-time dependency
```

In [10]:

```
#%%time stats, heatmap, optimize_result = backtest.optimize( n1=[10, 100], # Note: For method="sambo",
we n2=[20, 200], # only need interval end-points n_enter=[10, 40], n_exit=[10, 30], constraint=lambda p:
p.n_exit < p.n_enter < p.n1 < p.n2, maximize='Equity Final [$]', method='sambo', max_tries=40,
random_state=0, return_heatmap=True, return_optimization=True)
```

In [11]:

```
heatmap.sort_values().iloc[-3:]
```

Out[11]:

```
n1 n2 n_enter n_exit 60 92 40 24 29016.24 67 88 40 30 34017.73 54 84 38 25 37542.84 Name: Equity Final
[$], dtype: float64
```

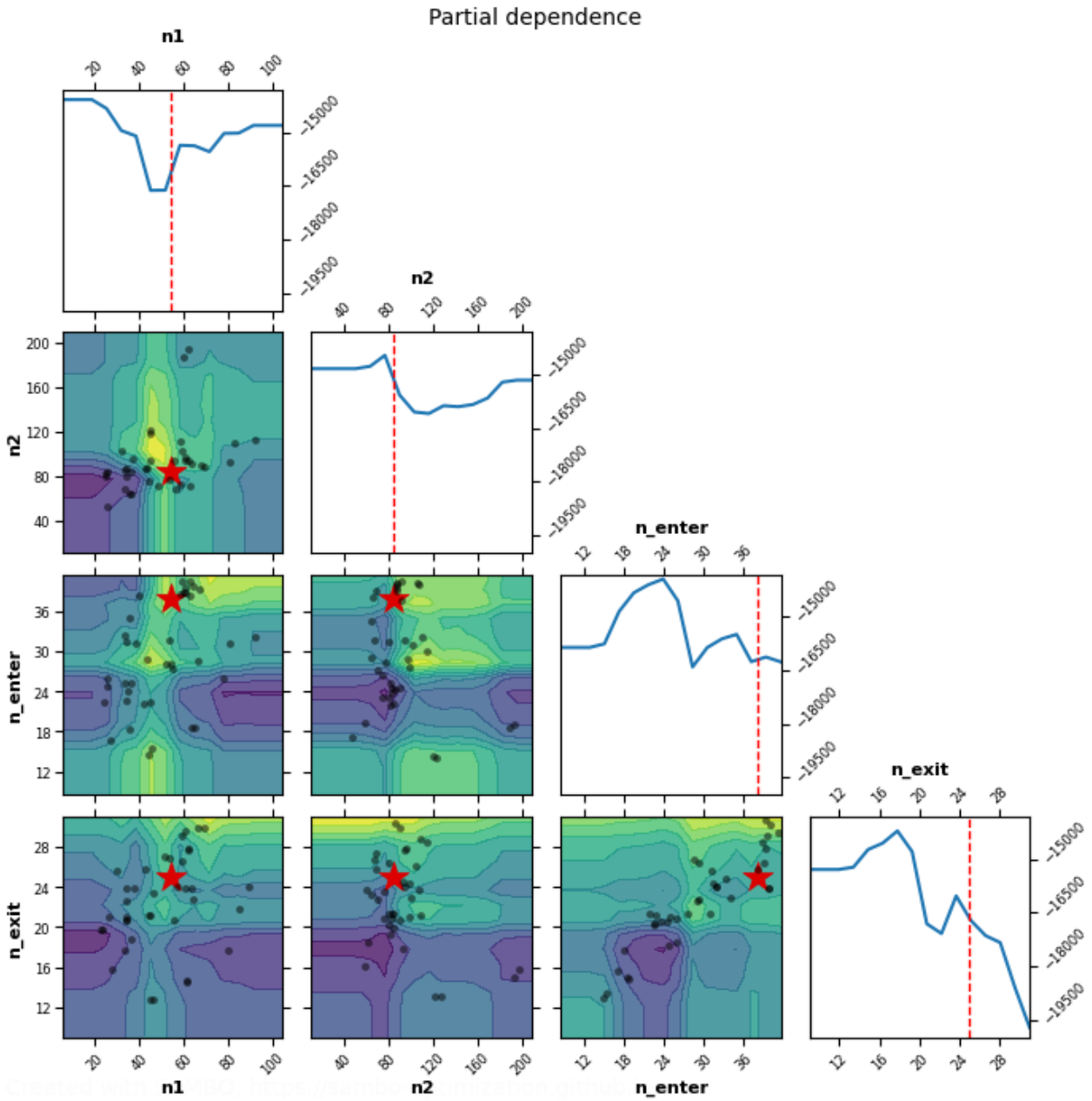
Notice how the optimization runs somewhat slower even though `max_tries=` is lower. This is due to the sequential nature of the algorithm and should actually perform quite comparably even in cases of *much larger parameter spaces* where grid search would effectively blow up, likely reaching a better optimum than a simple randomized search would. A note of warning, again, to take steps to avoid overfitting insofar as possible.

Understanding the impact of each parameter on the computed objective function is easy in two dimensions, but as the number of dimensions grows, partial dependency plots are increasingly useful. Plotting tools from *SAMBO* take care of the more mundane things needed to make good and informative plots of the parameter space.

Note, because SAMBO internally only does *minimization*, the values in `optimize_result` are negated (less is better).

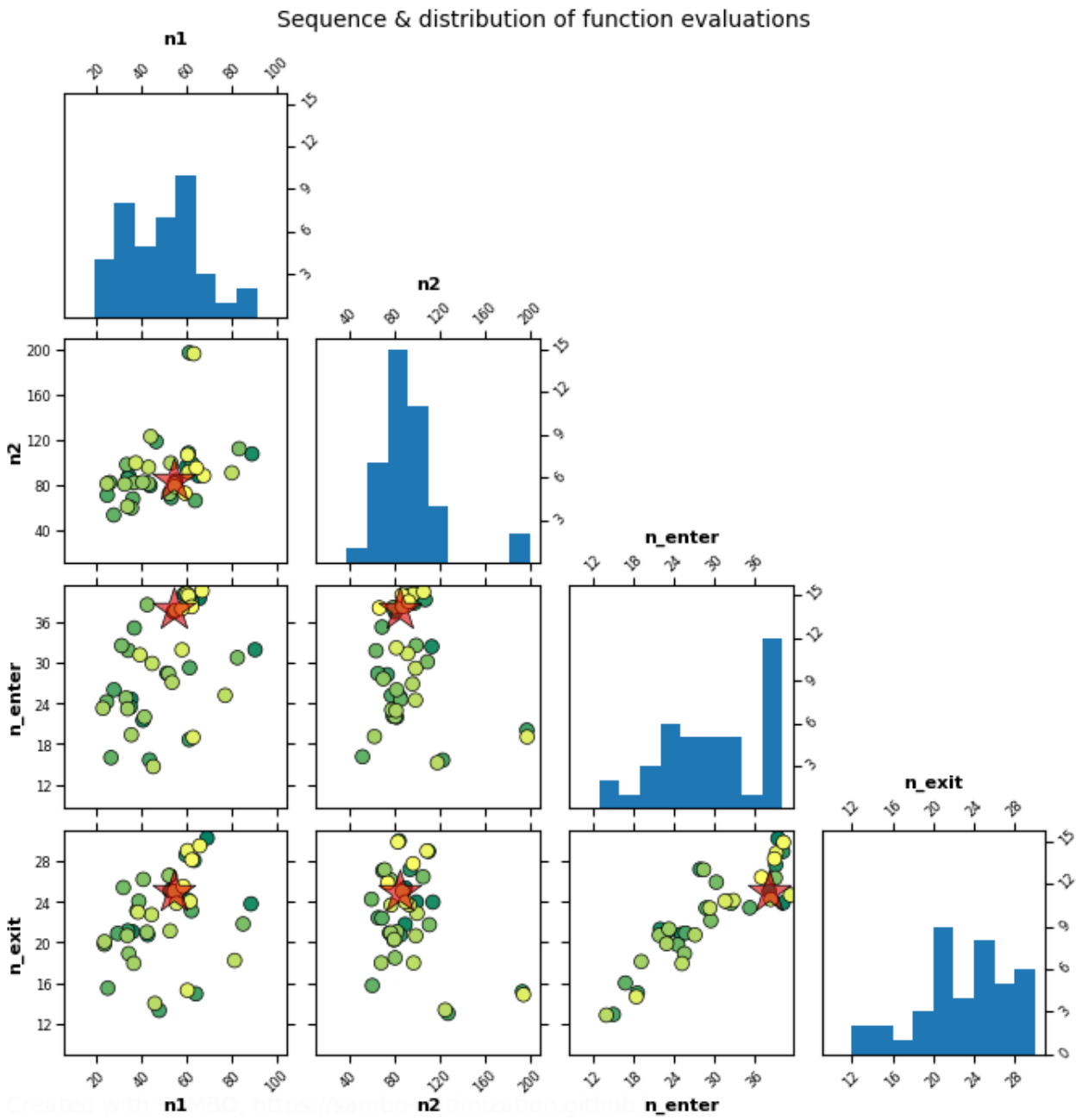
In [12]:

```
from sambo.plot import plot_objective names = ['n1', 'n2', 'n_enter', 'n_exit'] _ =
plot_objective(optimize_result, names=names, estimator='et')
```



In [13]:

```
from sambo.plot import plot_evaluations _ = plot_evaluations(optimize_result, names=names)
```



Learn more by exploring further examples or find more framework options in the full API reference.