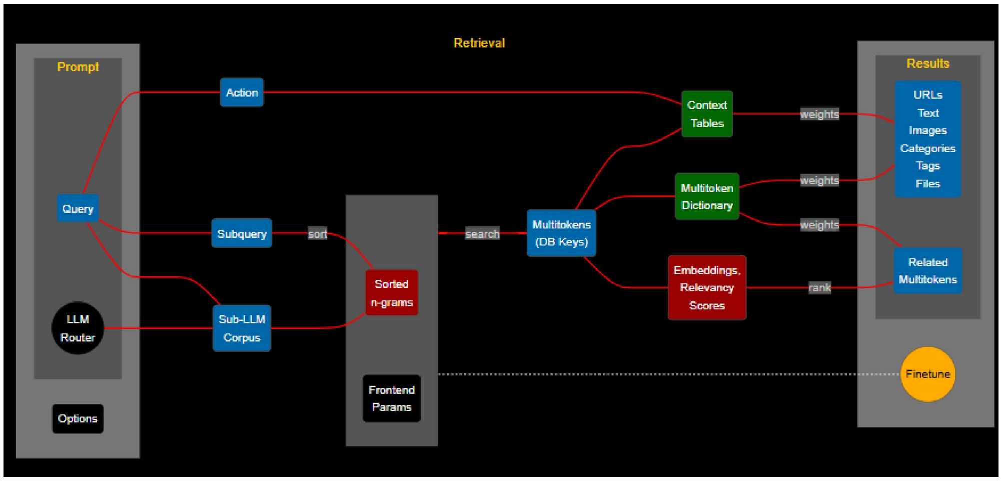
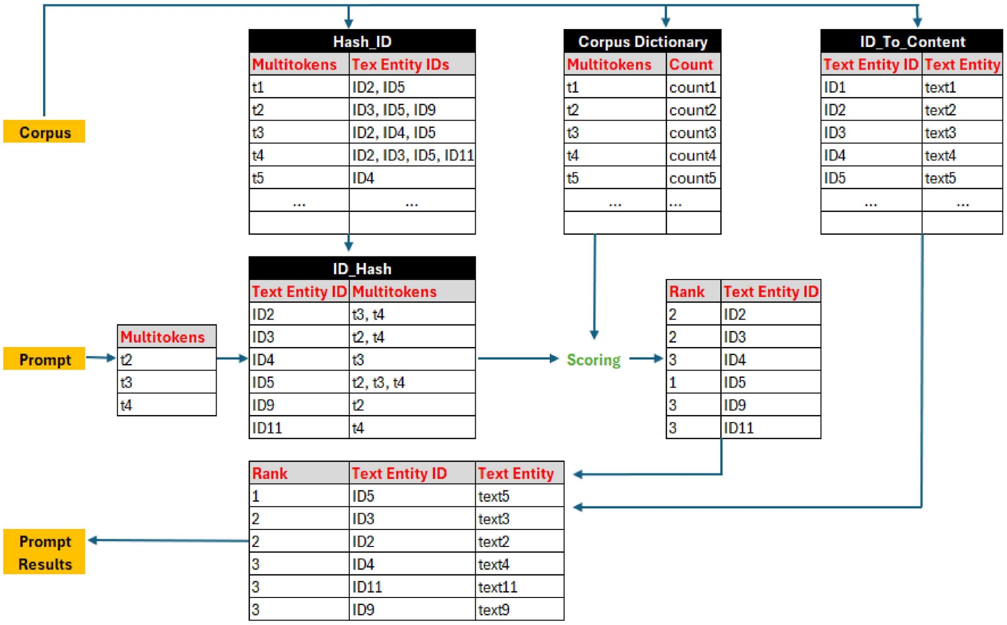
Building Disruptive AI & LLM Technology From Scratch





Hash_ID

Multitokens	Tex Entity IDs
t1	ID2, ID5
t2	ID3, ID5, ID9
t3	ID2, ID4, ID5
t4	ID2, ID3, ID5, ID11
t5	ID4
...	...

Corpus Dictionary

Multitokens	Count
t1	count1
t2	count2
t3	count3
t4	count4
t5	count5
...	...

ID_To_Content

Text Entity ID	Text Entity
ID1	text1
ID2	text2
ID3	text3
ID4	text4
ID5	text5
...	...

Multitokens
t2
t3
t4

ID_Hash

Text Entity ID	Multitokens
ID2	t3, t4
ID3	t2, t4
ID4	t3
ID5	t2, t3, t4
ID9	t2
ID11	t4

Rank	Text Entity ID
2	ID2
2	ID3
3	ID4
1	ID5
3	ID9
3	ID11

Rank	Text Entity ID	Text Entity
1	ID5	text5
2	ID3	text3
2	ID2	text2
3	ID4	text4
3	ID11	text11
3	ID9	text9

Vincent Granville, Ph.D. | www.MLTechniques.com | Version 2.0, October 2024

Contents

Introduction

This book features new advances in game-changing AI and LLM technologies built by GenAItechLab.com. Written in simple English, it is best suited for engineers, developers, data scientists, analysts, consultants and anyone with an analytic background interested in starting a career in AI. The emphasis is on scalable enterprise solutions, easy to implement, yet outperforming vendors both in terms of speed and quality, by several orders of magnitude.

Each topic comes with GitHub links, full Python code, datasets, illustrations, and real life case studies, including from Fortune 100 company. Some of the material is presented as enterprise projects with solution, to help you build robust applications and boost your career. You don't need expensive GPU and cloud bandwidth to implement them: a standard laptop works.

Part I: Hallucination-Free LLM with Real-Time Fine-Tuning focuses on high performance in-memory agentic multi-LLMs for professional users and enterprise, with real-time fine-tuning, self-tuning, no weight, no training, no latency, no hallucinations, no GPU. Made from scratch, leading to replicable results, leveraging explainable AI, adopted by Fortune 100. With a focus on delivering concise, exhaustive, relevant, and in-depth search results, references, and links. See also the section on 31 features to substantially boost RAG/LLM performance.

Part II: Outperforming Neural Nets and Classic AI discusses related large-scale systems also benefiting from a light-weight but more efficient architecture. It features LLMs for clustering, classification, and taxonomy creation, leveraging the knowledge graphs embedded in and retrieved from the input corpus when crawling. Then, in chapters 7 and 8, I focus on tabular data synthetization, presenting techniques such as NoGAN, that significantly outperform neural networks, along with the best evaluation metric. The methodology in chapter 9 applies to most AI problems. It offers a generic tool to improve any existing architecture relying on gradient descent, such as deep neural networks.

Part III: Innovations in Statistical AI features a collection of methods that you can integrate in any AI system to boost performance. Based on a modern approach to statistical AI, they cover probabilistic vector search, sampling outside the observation range, strong random number generators, math-free gradient descent, beating the slow statistical convergence of parameter estimates dictated by the Central Limit Theorem, exact geospatial interpolation for non-smooth systems, and more. Efficient chunking and indexing for LLMs is the topic of chapter 10. Finally, chapter 15 shows how to optimize trading strategies to consistently outperform the stock market.

About the author

Vincent Granville is a pioneering GenAI scientist and machine learning expert, co-founder of Data Science Central (acquired by a publicly traded company in 2020), Chief AI Scientist at MLTechniques.com, former VC-funded executive, author and patent owner – one related to LLM. Vincent's past corporate experience includes Visa, Wells Fargo, eBay, NBC, Microsoft, and CNET.

Vincent is also a former post-doc at Cambridge University, and the National Institute of Statistical Sciences (NISS). He published in *Journal of Number Theory*, *Journal of the Royal Statistical Society* (Series B), and *IEEE Transactions on Pattern Analysis and Machine Intelligence*. He is the author of multiple books, available here, including "Synthetic Data and Generative AI" (Elsevier, 2024). Vincent lives in Washington state, and enjoys doing research on stochastic processes, dynamical systems, experimental math and probabilistic number theory.

Chapter 15

Trading the S&P 500 Index

Before telling what this chapter is about, I want to discuss Figure 15.1. It features problems found in almost all AI systems. The solutions are relevant to many contexts well beyond Fintech. Each dot in the scatterplots represents the performance of an algorithm based on two parameters: β (X-axis) and σ (Y-axis). There is an extra parameter ω, with two values tested here: $\omega = 40$ on the left, and $\omega = 60$ on the right. The framework is highly non-linear, and indeed quite chaotic.

The color indicates the performance level: blue for great, green for good, gray for similar to baseline, orange for poor, and red for bad. The goal is to find decent parameter vectors (β, σ, ω) in the parameter space, based on the 128 tested configurations: 64 on the left, and another 64 on the right. Each test consists of running an algorithm on 12 different large portions of the input dataset, itself consisting of 40 years' worth or daily stock market data. In short, each performance measurement (color) is averaged over 12 different input datasets. The approach is similar to using grid search to optimize hyperparameters in a neural network. But there is more to it, applicable to the core of almost all neural networks and machine learning techniques: gradient descent.

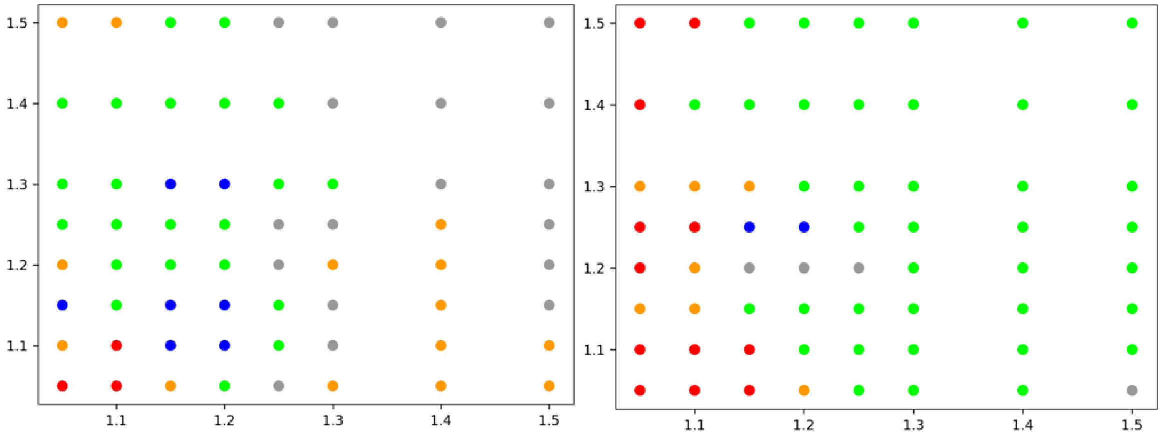


Figure 15.1: Blue/green outperform baseline, orange/red underperform (left: $\omega = 40$, right: $\omega = 60$)

Interesting conclusions:

- There are areas with relatively stable and good performance (green) in the parameter space. There may be many of them, separated by rather large regions of medium or low performance. For gradient descent, as long as it lands in one of them, the results will be good. However, progress within a green zone can be very slow, as the gradient approaches zero. You can stop the descent once deep enough inside a green zone. See how gradient descent works on datasets like this one, without loss function, in section 14.4. An alternative method to find the optimum is smart grid search.
- You want to find a large green region with good or medium performance, rather than a small one with good or great performance. Also, the boundary of these regions can be very unstable, thus the reason to avoid small ones. Note that large green regions may have red zones inside. Do not get too close, these red zones are like black holes. Some of the best parameters (the blue dots) are very close to some of the worst (red dots). Avoid these blue dots to reduce overfitting. This is typical in many problems, where

green zones are basins of attraction in the underlying chaotic dynamical system acting behind the scenes, surrounded by basins of repulsion. For more details, see my book [9] on this topic.

- The somewhat chaotic boundaries of the various regions can be detected using the interpolation algorithm described in section 14.3. This algorithm generalizes to higher dimensions.

Now I focus on the problem addressed in this chapter: designing S&P 500 trading strategies that outperform the baseline (staying long the entire period). The idea consists of exiting (selling) and re-entering (buying) at the right times, including for the first entry, over a long time period, ranging from 3 to 10 years. Depending on the parameters β, σ, ω to be explained later, the number of exits ranges from one or two, to a dozen. At the end of the pre-specified time period, two scenarios are possible: either you have already exited, or you still have an open position. The latter is called *holding*. The right times to exit or re-enter are determined by price variations in the index, your accumulated ROI, the price at your last exit, the time elapsed since your last exit, and the new price. It is well-known that this index, while going up on average, experiences sharp drops – sometimes prolonged – and rebounds that can be leveraged.

15.1 Non-standard strategies

It is difficult to successfully arbitrage the stock market due to the large number of participants competing against you. Staying long on the S&P 500 index is one of the most effective strategies, even outperforming many if not most professional traders, in the long run. In order to do better while minimizing competition, you need to use strategies that Wall Street professionals must avoid. For instance, keeping cash for extended periods of time (sometimes for years in a row) to be able to jump in at the right time with no advance notice – after a massive crash – then buy and sell during a short window following the crash to leverage the resulting volatility, to finally have a stable long position acquired at a steeply discounted price. You then sell the position in question years or months later when its value has massively increased following the slow or fast rebound. Then you repeat the cycle.

The strategies discussed here are inspired by the above principle. Each strategy has a specific parameter vector (β, σ, ω) and its own time frame, starting at t_0 and ending at t_f. The first buy takes place at a time t'_0 in that window, typically with $t'_0 > t_0$, and the last sell at a time t'_f either within that window, or later on if you must still hold your position at time t_f. The annualized compound return is compared to the baseline: buying at t_0, selling at t_f. If you still hold your position at t_f, the return on your strategy is based on the value of your position at t_f. The starting point to compute your return is t_0, not t'_0.

If you are all cash, you buy when the price reaches $\beta \cdot p_{\text{min}}(\omega)$, where $p_{\text{min}}(\omega)$ is the bottom price observed in the last w days preceding the purchase. If holding (active position), you sell when the price hits the $\sigma \cdot p_{\text{buy}}$ threshold, where p_{buy} is the price you paid per unit in your last purchase. Much of the work consists of testing various (β, σ, ω) that works well across various time frames $[t_0, t_f]$, and that are stable. That is, not sensitive to small variations. Now that everything is in place, I make the link to the names used in the Python code in section 15.3.

- In the code, $t_0, t_f, \beta, \sigma, \omega$ are respectively named `init`, `end`, `buy_param`, `sell_param`, `windowLow`, and stored in the `params` dictionary. See lines 121–125 in the code. One extra parameter `rnd` is discussed later. For t'_0 and t'_f, the corresponding notations are `entry_idate` and `exit_idate`. The prices p_{buy} and p_{min} are respectively denoted as `buy_price` and `bottom[params]`. Finally, `params` is the key to many key-value tables.

- I tested 4 values for t_0 and 3 for t_f. Thus, for each strategy (β, σ, ω), the summary statistics are based on $4 \times 3 = 12$ observations. An observation is the result of trading the entire time period $[t_0, t_f]$. See lines 47–48 in the code. Note that instead of specifying t_f, I use `duration`, with $t_f = t_0 + \texttt{duration}$.

- I tested 8 values for β, 8 for σ, and 2 for ω, leading to $8 \times 8 \times 2 = 128$ strategies, each with 12 observations based on the 12 combinations of t_0, t_f. See lines 49–51 in the code. Thus, the output dataset (see here) has 128 rows: one per strategy, each featuring averages based on 12 measurements.

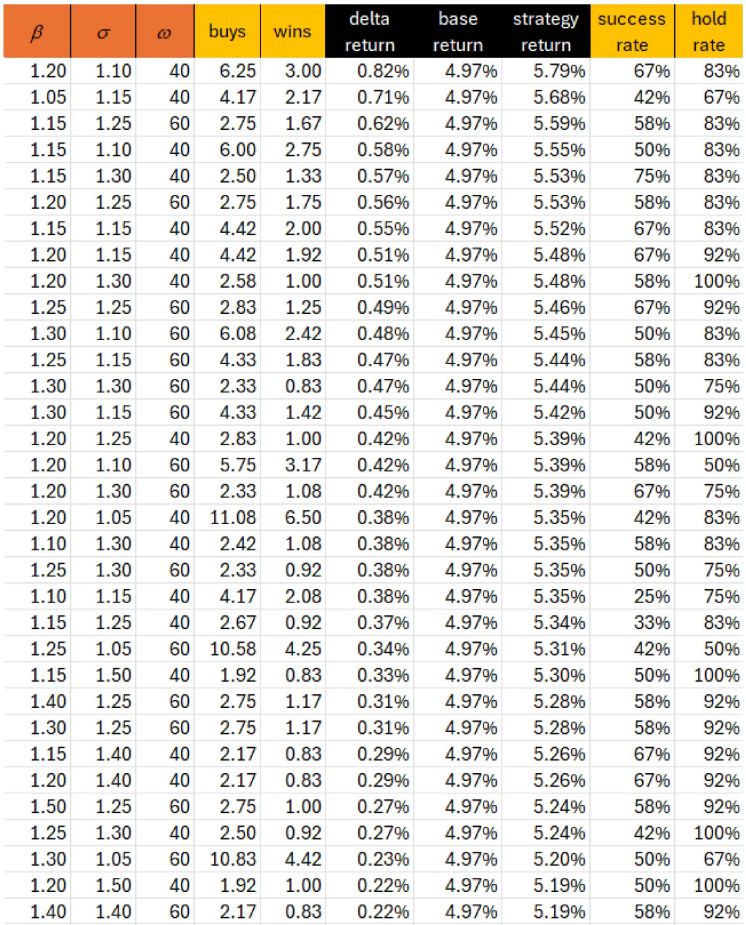
- Each daily price is a value between the low and the high of the day, specified by the fixed weight `rnd=0.50`. See line 128 in the code. You should try with different values of `rnd`, to rule out strategies too sensitive to little variations.

The code is somewhat complicated because of the pre-processing step to accelerate many computations. The bottleneck is calculating the minimum price in the moving window consisting of the last w days. It is done once in lines 58–61, rather than for each (β, σ, ω), speeding up the computations by a factor 128.

Other important metrics attached to a strategy are the number `nbuys` of "buy" trades, and among those trades, the `wins`, that is, when buying at a lower price than the previous sell. See lines 81 and 84 in the code. The hold rate is the chance to still be in a holding position at the end of the time period. In the results, all these numbers are averaged per strategy (β, σ, ω) with the strategy average computed over the 12 time series of trading activity (one for each $[t_0, t_f]$). For a strategy, the success rate is defined as the chance to outperform the "buy and hold" baseline.

15.2 Selecting a strategy based on its features

Figure 15.2 shows the summary results for the top 30 strategies, out of 128. The annualized ROI gain over the baseline ranges from 0.82% to -1.71%. Here the baseline is a 4.97% ROI. The average number of "buys" per strategy ranges from barely above 1 to nearly 11. Among those buys, usually fewer than 50% are "wins", meaning buying at a lower price than the previous sell. Despite this seemingly low performance, the overall success rate (outperforming baseline) per strategy may be over 50% due to the fact that a few of the wins are spectacular, taking place after a massive price drop.



β	σ	ω	buys	wins	delta return	base return	strategy return	success rate	hold rate
1.20	1.10	40	6.25	3.00	0.82%	4.97%	5.79%	67%	83%
1.05	1.15	40	4.17	2.17	0.71%	4.97%	5.68%	42%	67%
1.15	1.25	60	2.75	1.67	0.62%	4.97%	5.59%	58%	83%
1.15	1.10	40	6.00	2.75	0.58%	4.97%	5.55%	50%	83%
1.15	1.30	40	2.50	1.33	0.57%	4.97%	5.53%	75%	83%
1.20	1.25	60	2.75	1.75	0.56%	4.97%	5.53%	58%	83%
1.15	1.15	40	4.42	2.00	0.55%	4.97%	5.52%	67%	83%
1.20	1.15	40	4.42	1.92	0.51%	4.97%	5.48%	67%	92%
1.20	1.30	40	2.58	1.00	0.51%	4.97%	5.48%	58%	100%
1.25	1.25	60	2.83	1.25	0.49%	4.97%	5.46%	67%	92%
1.30	1.10	60	6.08	2.42	0.48%	4.97%	5.45%	50%	83%
1.25	1.15	60	4.33	1.83	0.47%	4.97%	5.44%	58%	83%
1.30	1.30	60	2.33	0.83	0.47%	4.97%	5.44%	50%	75%
1.30	1.15	60	4.33	1.42	0.45%	4.97%	5.42%	50%	92%
1.20	1.25	40	2.83	1.00	0.42%	4.97%	5.39%	42%	100%
1.20	1.10	60	5.75	3.17	0.42%	4.97%	5.39%	58%	50%
1.20	1.30	60	2.33	1.08	0.42%	4.97%	5.39%	67%	75%
1.20	1.05	40	11.08	6.50	0.38%	4.97%	5.35%	42%	83%
1.10	1.30	40	2.42	1.08	0.38%	4.97%	5.35%	58%	83%
1.25	1.30	60	2.33	0.92	0.38%	4.97%	5.35%	50%	75%
1.10	1.15	40	4.17	2.08	0.38%	4.97%	5.35%	25%	75%
1.15	1.25	40	2.67	0.92	0.37%	4.97%	5.34%	33%	83%
1.25	1.05	60	10.58	4.25	0.34%	4.97%	5.31%	42%	50%
1.15	1.50	40	1.92	0.83	0.33%	4.97%	5.30%	50%	100%
1.40	1.25	60	2.75	1.17	0.31%	4.97%	5.28%	58%	92%
1.30	1.25	60	2.75	1.17	0.31%	4.97%	5.28%	58%	92%
1.15	1.40	40	2.17	0.83	0.29%	4.97%	5.26%	67%	92%
1.20	1.40	40	2.17	0.83	0.29%	4.97%	5.26%	67%	92%
1.50	1.25	60	2.75	1.00	0.27%	4.97%	5.24%	58%	92%
1.25	1.30	40	2.50	0.92	0.27%	4.97%	5.24%	42%	100%
1.30	1.05	60	10.83	4.42	0.23%	4.97%	5.20%	50%	67%
1.20	1.50	40	1.92	1.00	0.22%	4.97%	5.19%	50%	100%
1.40	1.40	60	2.17	0.83	0.22%	4.97%	5.19%	58%	92%

Figure 15.2: Average stats for each strategy (β, σ, ω)

Not surprisingly, the chances to finish with an active position (holding) is rather high due to the small volume of trading activity, favoring long over short-term holding positions. See rightmost column in Figure 15.2. It would be interesting to see what proportion of time is spent on holding a position, versus being all cash.

How to choose a strategy, that is, β, σ and ω? A large number of "buys" with 40% of more that are "wins" is an indication of robustness. Another benefit is that you are more frequently in a cash position. This can be useful if you face some sudden emergencies. However, frequent trades may be subject to short term capital gains. You also want to avoid good strategies (green or blue dot) with parameters β, σ, ω too close to a red dot in Figure 15.1.

Finally, most of the tests were performed on the last 20 years of historical data. Earlier patterns are slightly different and more erratic. Also, you can play with multiple strategies in parallel. The strategies can be used to identify entry and exit points in the S&P 500 index. It would be interesting the break down performance based on the length $t_f - t_0$ of the time window

15.3 Python code and dataset: 40 years’ worth of historical data

The Python code `spx500.py` is on GitHub in my “Statistical Optimization” repository: click here to access it. The input dataset `spx500.txt` and output results `spx500-results.xlsx` are in the same repository, respectively here and here. The input data comes from Yahoo Finance: you can also download it from here, and also access data for individual stocks.

```python
import numpy as np
import pandas as pd
from datetime import datetime
import matplotlib.pyplot as plt
import matplotlib as mpl


#--- [1] Read data

data = pd.read_csv("spx500.txt")

def date_to_int(date, start_date):
   xstart_date = datetime.strptime(start_date, ’%b %d %Y’)
   xdate = datetime.strptime(date, ’%b %d %Y’)
   date_int = str(xdate - xstart_date)
   date_int = date_int.split(’ ’)[0]
   if date_int == ’0:00:00’:
      date_int = 0
   else:
      date_int = int(date_int)
   return(date_int)

arr_date = data[’Date’]
arr_low = data[’Low’]
arr_high = data[’High’]
arr_open = data[’Open’]
arr_idate = []
arr_tdate = []
nobs = len(arr_date)

for k in range(nobs):
   arr_idate.append(date_to_int(arr_date[k], arr_date[0]))
   tdate = datetime.strptime(arr_date[k], ’%b %d %Y’)
   arr_tdate.append(tdate)

data.insert(1, ’iDate’, arr_idate, True)

print(data.head())
print(data.columns.values)
print(data.shape)
# features: ’Date’,’iDate’,’Open’,’High’,’Low’,’Close’,’Adj_Close’,’Volume’

#--- [2] Initializations for main loop

h_params = {}

l_init = (4000, 5000, 6000, 7000)
l_duration = (1000, 2000, 3000)
l_windowLow = (40, 60)
l_buy_param = (1.05, 1.1, 1.15, 1.2, 1.25, 1.3, 1.4, 1.5)
l_sell_param = (1.05, 1.1, 1.15, 1.2, 1.25, 1.3, 1.4, 1.5)

h_max = {}
h_min = {}

#- [2.1] preprocessing to speed up computations

for k in range(nobs):
   for windowLow in l_windowLow:
      if k >= windowLow:
         h_min[(k, windowLow)] = min(arr_low[k-windowLow:k-1])

#- [2.2] create table of parameter sets (stored as key in h_params)
```

```python

for buy_param in l_buy_param:
   for sell_param in l_sell_param:
      for windowLow in l_windowLow:
         for init in l_init:
            for duration in l_duration:
               end = init + duration
               key = (init,end,buy_param,sell_param,windowLow)
               h_params[key] = 1

#- [2.3] Create main hash tables
#      The index (same in all tables) is the parameter set

bottom   = {}
hold     = {} # true if we have open position
value    = {}
max_hold = {}
nbuys    = {} # number buys done during whole period
entry_price = {} # buy price on first buy
sell_price = {} # sell price on last sell, before current buy
wins     = {} # a win is buying a lower price than last sell

init_price = {} # price when trading period starts
buy_price = {} # buy price
end_price = {} # price when trading period ends
entry_idate = {} # date of first buy
exit_idate = {} # date of last sell if out, otherwise end_idate
init_idate = {} # date when trading period starts
end_idate = {} # date when trading period ends
start    = {}

for params in h_params:

   bottom[params] = arr_high[0]
   hold[params]   = False
   value[params]  = 0
   max_hold[params] = 0
   nbuys[params]  = 0
   entry_price[params] = -1
   sell_price[params] = -1
   wins[params]   = 0


#--- [3] Main loop

for k in range(nobs):
   # loop over all trading days

   idate = arr_idate[k]
   price_high = arr_high[k]
   price_low = arr_low[k]
   if k % 100 == 0:
      print(arr_date[k])

   for params in h_params:
      # update trading stats for each param set in parallel

      init     = params[0]
      end      = params[1]
      buy_param = params[2]
      sell_param = params[3]
      windowLow = params[4]
      rnd      = 0.5

      price = rnd * price_high + (1-rnd) * price_low

      if k == init:
         init_price[params] = price
         init_idate[params] = idate
      elif k == end:
         end_price[params] = price
         end_idate[params] = idate

      if k >= windowLow and price < h_min[(k, windowLow)]:
         bottom[params] = price

      if k >= init and not hold[params] and k < end:
         if price < buy_param * bottom[params]:
            # buy
            buy_price[params] = price
            if value[params] == 0:
               # first purchase
               value[params] = price
               entry_price[params] = price
               entry_idate[params] = idate
               if price < init_price[params]:
                  # first buy at lower price than init price
                  wins[params] += 1
            elif buy_price[params] < sell_price[params]:
               wins[params] += 1
            start[params] = idate
            hold[params] = True
            nbuys[params] += 1

      elif hold[params] and k < end:
         span = idate-start[params]
         if span > max_hold[params]:
            max_hold[params] = span
         if price > sell_param * buy_price[params]:
            # sell
            sell_price[params] = price
            value[params] *= price/buy_price[params]
            hold[params]= False
            exit_idate[params] = idate


#--- [4] Summary stats for each param set

# group params in h_params by (init, end)
# stored average stats by grouped params, in arr_local

hash_performance = {}
hash_count = {}

for params in h_params:
   if hold[params]:
      value[params] *= end_price[params]/buy_price[params]
      exit_idate[params] = end_idate[params]
   else:
      if end_price[params] < sell_price[params]:
         wins[params] += 1
   duration1 = end_idate[params] - init_idate[params] + 1
   duration2 = exit_idate[params] - entry_idate[params] + 1
   R_market = end_price[params] / init_price[params]
   R_strategy = value[params] / entry_price[params]
   adj_R_market = 100*(R_market**(365/duration1) - 1)
   adj_R_strategy = 100*(R_strategy**(365/duration1) - 1)
   ratio = R_strategy / R_market # reinvest all to compound return
   performance = adj_R_strategy - adj_R_market
   if performance > 0:
      success = 1
   else:
      success = 0

   key = (params[2], params[3], params[4])
   arr_local = [nbuys[params], wins[params],
             performance, adj_R_market, adj_R_strategy,
             duration1, duration2, success, hold[params]]

   if key in hash_performance:
      arr_local2 = hash_performance[key]
      for idx in range(len(arr_local2)):
         arr_local2[idx] += arr_local[idx]
      hash_performance[key] = arr_local2
      hash_count[key] += 1
   else:
      hash_performance[key] = arr_local
      hash_count[key] = 1

#- [4.1] to get averages for each param set, divide sums by sample size cnt

cnt = hash_count[key]
```

```

for key in hash_performance:
   arr_avg = hash_performance[key]
   for idx in range(len(arr_avg)):
      arr_avg[idx] /= cnt


#--- [5] Print results

OUT = open("spx500-results2.txt", "w")

labels="beta\tsigma\tomega\tsample size\tbuys\tbuys below last sell\tDelta return"
labels+="base return\ttrading return\tperiod (days)\tactive peiod\tsuccess rate\thold rate"
OUT.write(labels + "\n")

for key in hash_count:
   strout = ""
   for param in key:
      strout += str(param) + "\t"
   cnt = hash_count[key]
   strout += str(cnt) + "\t"
   arr_avg = hash_performance[key]
   for idx in range(len(arr_avg)):
      strout += str(arr_avg[idx]) + "\t"
   strout += "\n"
   OUT.write(strout)

OUT.close()


#--- [6] Visualizations

hash_xy = {}
hash_color = {}
for value in l_windowLow:
   hash_xy[value] = []
   hash_color[value] = []

for key in hash_count:
   cnt = hash_count[key]
   arr_avg = hash_performance[key]
   for params in key:
      buy_param = key[0]
      sell_param = key[1]
      window_low = key[2]
      performance = arr_avg[2] # performance
      if performance > 0.5:
         color = [0, 0, 1] # blue
      elif performance > 0.1:
         color = [0, 1, 0] # lightgreen
      elif performance > -0.1:
         color = [0.6, 0.6, 0.6] # lightgray
      elif performance > -0.5:
         color = [1, 0.6, 0] # orange
      else:
         color = [1, 0, 0] # red
      local_arr = hash_xy[window_low]
      local_arr.append([buy_param, sell_param])
      local_arr = hash_color[window_low]
      local_arr.append(color)

mpl.rcParams['axes.linewidth'] = 0.5
plt.rcParams['xtick.labelsize'] = 9
plt.rcParams['ytick.labelsize'] = 9

for window_low in hash_xy:
   z = np.array(hash_xy[window_low])
   z = np.transpose(z)
   color = hash_color[window_low]
   plt.scatter(z[0], z[1], c = color)
   plt.show()
```