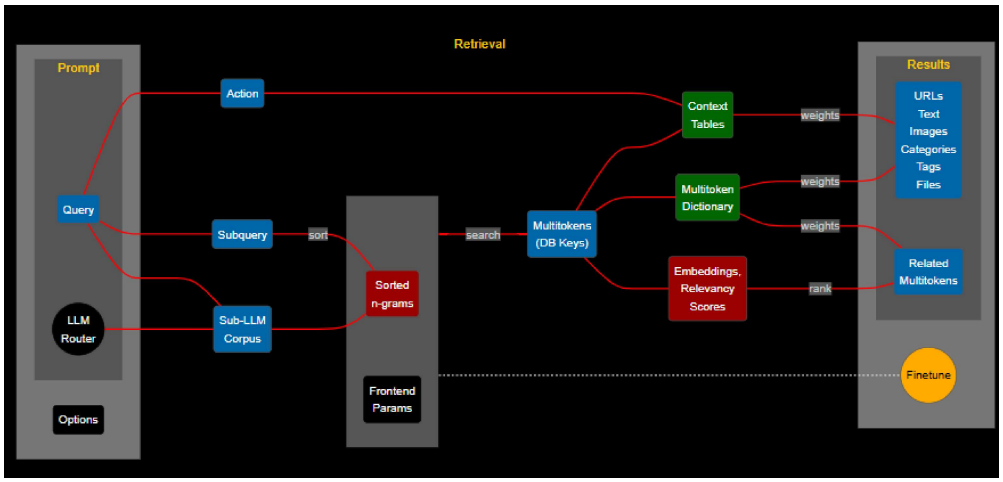
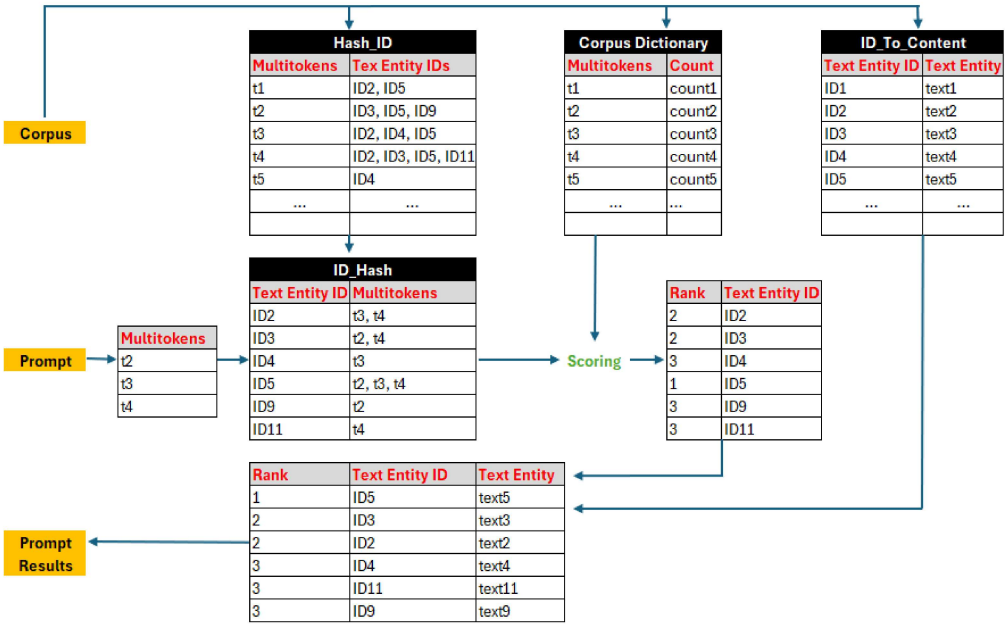

Retrieval
Prompt
Action
Query
Subquery
sort
Sub-LLM
Corpus
LLM
Router
Options
Sorted
n-grams
Frontend
Params
search
Multitokens
(DB Keys)
Context
Tables
Multitoken
Dictionary
Embeddings,
Relevancy
Scores
weights
weights
weights
rank
Results
URLs
Text
Images
Categories
Tags
Files
Related
Multitokens
Finetune


Corpus
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
ID_Hash
Text Entity ID
Multitokens
ID2
t3, t4
ID3
t2, t4
ID4
t3
ID5
t2, t3, t4
ID9
t2
ID11
t4
Prompt
Multitokens
t2
t3
t4
Scoring
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
Prompt
Results

Contents

I Hallucination-Free LLM with Real-Time Fine-Tuning 6

1 Enterprise xLLM for Search & Retrieval 7
- 1.1 Efficient database architecture: nested hashes . . . 7
- 1.2 From frontend prompts to backend tables . . . 8
- 1.3 What is not covered here . . . 9

2 Parameters, features, and fine-tuning 11
- 2.1 Backend parameters . . . 11
- 2.2 Frontend parameters . . . 12
- 2.3 Agents . . . 12
- 2.4 Reproducibility . . . 13
- 2.5 Singularization, stemming, auto-correct . . . 14
- 2.6 Augmentation, distillation, and frontend tables . . . 14
- 2.7 In-memory database, latency, and scalability . . . 15

3 Case study 17
- 3.1 Real-time fine-tuning, prompts, and command menu . . . 18
- 3.2 Sample session . . . 19
- 3.3 Web API for enterprise xLLM . . . 29
  - 3.3.1 Left panel: command menu and prompt box . . . 29
  - 3.3.2 Right panel: prompt results . . . 29
  - 3.3.3 Next steps . . . 31
- 3.4 Conclusions and references . . . 31

4 Appendix 32
- 4.1 Python code . . . 32
- 4.2 Thirty features to boost LLM performance . . . 44
  - 4.2.1 Fast search and caching . . . 44
  - 4.2.2 Leveraging sparse databases . . . 45
  - 4.2.3 Contextual tokens . . . 45
  - 4.2.4 Adaptive loss function . . . 45
  - 4.2.5 Contextual tables . . . 45
  - 4.2.6 Smart crawling . . . 45
  - 4.2.7 LLM router, sub-LLMs, and distributed architecture . . . 45
  - 4.2.8 From one trillion parameters down to two . . . 46
  - 4.2.9 Agentic LLMs . . . 46
  - 4.2.10 Data augmentation via dictionaries . . . 46
  - 4.2.11 Distillation done smartly . . . 46
  - 4.2.12 Reproducibility . . . 47
  - 4.2.13 Explainable AI . . . 47
  - 4.2.14 No training, in-memory LLM . . . 47
  - 4.2.15 No neural network . . . 47
  - 4.2.16 Show URLs and references . . . 48
  - 4.2.17 Taxonomy-based evaluation . . . 48
  - 4.2.18 Augmentation via prompt data . . . 48
  - 4.2.19 Variable-length embeddings, indexing, and database optimization . . . 48
  - 4.2.20 Favor backend over frontend engineering . . . 48
  - 4.2.21 Use NLP and Python libraries with caution . . . 49
  - 4.2.22 Self-tuning and customization . . . 49

4.2.23 Local, global parameters, and debugging . . . 49
4.2.24 Displaying relevancy scores, and customizing scores . . . 50
4.2.25 Intuitive hyperparameters . . . 50
4.2.26 Sorted n-grams and token order preservation . . . 50
4.2.27 Blending standard tokens with tokens from the knowledge graph . . . 50
4.2.28 Boosted weights for knowledge-graph tokens . . . 51
4.2.29 Versatile command prompt . . . 51
4.2.30 Boost long multitokens and rare single tokens . . . 51
4.2.31 Disambiguation . . . 51
4.3 LLM glossary . . . 51

II Outperforming Neural Nets and Classic AI 55

5 Building and evaluating a taxonomy-enriched LLM 57
5.1 Project and solution . . . 59
5.2 Python code . . . 60

6 LLM for Clustering and Predictions 66
6.1 Project and solution . . . 67
6.2 Visualizations and discussion . . . 69
6.3 Python code . . . 70

7 Fast, High-Quality NoGAN for Data Synthetization 78
7.1 Project description . . . 78
7.2 Solution . . . 80
7.3 Python implementation . . . 81

8 Hierarchical Bayesian NoGAN for Data Synthetization 87
8.1 Methodology . . . 87
8.1.1 Base algorithm . . . 88
8.1.2 Loss function . . . 88
8.1.3 Hyperparemeters and convergence . . . 89
8.1.4 Acknowledgments . . . 90
8.2 Case studies . . . 90
8.2.1 Synthesizing the student dataset . . . 91
8.2.2 Synthesizing the Telecom dataset . . . 93
8.2.3 Other case studies . . . 94
8.2.4 Auto-tuning the hyperparameters . . . 95
8.2.5 Evaluation with multivariate ECDF and KS distance . . . 96
8.3 Conclusion . . . 97
8.4 Python implementation . . . 98

9 Boosting Model Evaluation with Smart Adaptive Loss Functions 107
9.1 Project and solution . . . 108
9.2 Python code . . . 112

III Innovations in Statistical AI 117

10 Building a Ranking System to Enhance Prompt Results 119
10.1 Relevancy scores and rankings . . . 120
10.2 Case study . . . 121
10.2.1 xLLM for auto-indexing, cataloging and glossary generation . . . 121
10.2.2 xLLM for scientific research . . . 122
10.3 Python code . . . 123
10.4 Appendix: smart ranking in a nutshell . . . 129

11 Probabilistic Search: Alternative to Vector Search 131
11.1 Motivation and architecture . . . 132
11.2 Applications . . . 133
11.2.1 Embeddings and large language models . . . 133

11.2.2 Generating and evaluating synthetic data . . . 134
11.2.3 Clustering, dataset comparisons, outlier detection . . . 135
11.3 Project and solution . . . 136
11.4 Python code . . . 137

12 Strong Random Generators for Reproducible AI **141**
12.1 Strong Randomness and reproducibility: two key components . . . 141
12.2 Computing the digits of special math constants . . . 142
12.2.1 P-adic valuations . . . 142
12.2.2 Digit blocks, speed of convergence . . . 143
12.2.3 A plethora of interesting pseudo-random sequences . . . 143
12.3 Testing random number generators . . . 144
12.3.1 Theoretical properties of the digits of $\sqrt{2}$ . . . 144
12.3.2 Fast recursion and congruential equidistribution . . . 145
12.3.3 Exponential system: predicting the next block . . . 147
12.4 Python code . . . 149
12.4.1 Fast recursion . . . 149
12.4.2 Main code . . . 150

13 Sampling Outside the Observation Range **155**
13.1 Quantile convolution . . . 155
13.2 Truncated Gaussian mixtures and bias detection . . . 157
13.3 Case studies . . . 157
13.3.1 Conclusion . . . 158
13.4 Python code . . . 159

14 Additional Resources **162**
14.1 Accelerating convergence of parameter estimates . . . 162
14.1.1 First case study . . . 162
14.1.2 Second case study . . . 162
14.2 Generic, all-in-one curve fitting, regression and clustering . . . 164
14.2.1 Solution, R-squared and backward compatibility . . . 165
14.2.2 Model upgrades . . . 166
14.2.3 Case studies . . . 166
14.2.4 Python code . . . 170
14.3 A simple geospatial interpolation method . . . 171
14.3.1 Problem in two dimensions . . . 171
14.3.2 Spatial interpolation of the temperature dataset . . . 173
14.3.3 Python code . . . 174
14.4 Math-free, parameter-free gradient descent . . . 174
14.4.1 Introduction . . . 175
14.4.2 Implementation details . . . 175
14.4.3 General comments about the methodology and parameters . . . 178
14.4.4 Mathematical version of gradient descent and orthogonal trajectories . . . 179
14.4.5 Python code . . . 180

15 Trading the S&P 500 Index **181**
15.1 Non-standard strategies . . . 182
15.2 Selecting a strategy based on its features . . . 183
15.3 Python code and dataset: 40 years' worth of historical data . . . 184

Bibliography **188**

Index **190**

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 No-GAN, 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.

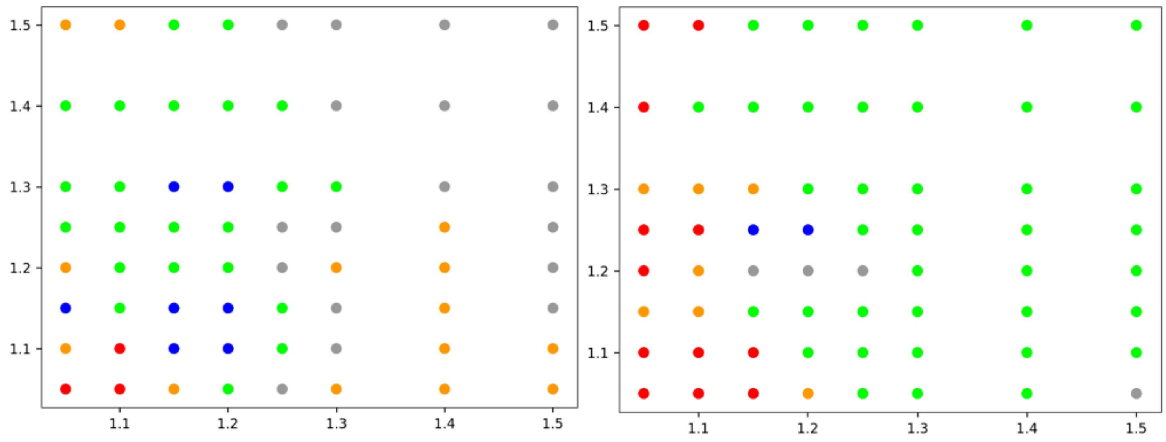


Figure 15.1: Blue/green outperform baseline, orange/red underperform (left: $\omega = 40$, right: $\omega = 60$)

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 $\beta \cdot p_{\min}(\omega)$, where $p_{\min}(\omega)$ 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 $t_f = t_0 +$ `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.

```
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)
```

```
64
65  for buy_param in l_buy_param:
66     for sell_param in l_sell_param:
67        for windowLow in l_windowLow:
68           for init in l_init:
69              for duration in l_duration:
70                 end = init + duration
71                 key = (init,end,buy_param,sell_param,windowLow)
72                 h_params[key] = 1
73
74  #- [2.3] Create main hash tables
75  #      The index (same in all tables) is the parameter set
76
77  bottom   = {}
78  hold     = {} # true if we have open position
79  value    = {}
80  max_hold = {}
81  nbuys    = {} # number buys done during whole period
82  entry_price = {} # buy price on first buy
83  sell_price = {} # sell price on last sell, before current buy
84  wins     = {} # a win is buying a lower price than last sell
85
86  init_price = {} # price when trading period starts
87  buy_price = {} # buy price
88  end_price = {} # price when trading period ends
89  entry_idate = {} # date of first buy
90  exit_idate = {} # date of last sell if out, otherwise end_idate
91  init_idate = {} # date when trading period starts
92  end_idate = {} # date when trading period ends
93  start    = {}
94
95  for params in h_params:
96
97     bottom[params] = arr_high[0]
98     hold[params]   = False
99     value[params]  = 0
100    max_hold[params] = 0
101    nbuys[params]  = 0
102    entry_price[params] = -1
103    sell_price[params] = -1
104    wins[params]   = 0
105
106
107 #--- [3] Main loop
108
109 for k in range(nobs):
110    # loop over all trading days
111
112    idate = arr_idate[k]
113    price_high = arr_high[k]
114    price_low = arr_low[k]
115    if k % 100 == 0:
116       print(arr_date[k])
117
118    for params in h_params:
119       # update trading stats for each param set in parallel
120
121       init     = params[0]
122       end      = params[1]
123       buy_param = params[2]
124       sell_param = params[3]
125       windowLow = params[4]
126       rnd      = 0.5
127
128       price = rnd * price_high + (1-rnd) * price_low
129
130       if k == init:
131          init_price[params] = price
132          init_idate[params] = idate
133       elif k == end:
134          end_price[params] = price
135          end_idate[params] = idate
136
137       if k >= windowLow and price < h_min[(k, windowLow)]:
138          bottom[params] = price
139
```

```
140       if k >= init and not hold[params] and k < end:
141          if price < buy_param * bottom[params]:
142             # buy
143             buy_price[params] = price
144             if value[params] == 0:
145                # first purchase
146                value[params] = price
147                entry_price[params] = price
148                entry_idate[params] = idate
149                if price < init_price[params]:
150                   # first buy at lower price than init price
151                   wins[params] += 1
152             elif buy_price[params] < sell_price[params]:
153                wins[params] += 1
154             start[params] = idate
155             hold[params] = True
156             nbuys[params] += 1
157
158       elif hold[params] and k < end:
159          span = idate-start[params]
160          if span > max_hold[params]:
161             max_hold[params] = span
162          if price > sell_param * buy_price[params]:
163             # sell
164             sell_price[params] = price
165             value[params] *= price/buy_price[params]
166             hold[params]= False
167             exit_idate[params] = idate
168
169
170 #--- [4] Summary stats for each param set
171
172 # group params in h_params by (init, end)
173 # stored average stats by grouped params, in arr_local
174
175 hash_performance = {}
176 hash_count = {}
177
178 for params in h_params:
179    if hold[params]:
180       value[params] *= end_price[params]/buy_price[params]
181       exit_idate[params] = end_idate[params]
182    else:
183       if end_price[params] < sell_price[params]:
184          wins[params] += 1
185    duration1 = end_idate[params] - init_idate[params] + 1
186    duration2 = exit_idate[params] - entry_idate[params] + 1
187    R_market = end_price[params] / init_price[params]
188    R_strategy = value[params] / entry_price[params]
189    adj_R_market = 100*(R_market**(365/duration1) - 1)
190    adj_R_strategy = 100*(R_strategy**(365/duration1) - 1)
191    ratio = R_strategy / R_market # reinvest all to compound return
192    performance = adj_R_strategy - adj_R_market
193    if performance > 0:
194       success = 1
195    else:
196       success = 0
197
198    key = (params[2], params[3], params[4])
199    arr_local = [nbuys[params], wins[params],
200              performance, adj_R_market, adj_R_strategy,
201              duration1, duration2, success, hold[params]]
202
203    if key in hash_performance:
204       arr_local2 = hash_performance[key]
205       for idx in range(len(arr_local2)):
206          arr_local2[idx] += arr_local[idx]
207       hash_performance[key] = arr_local2
208       hash_count[key] += 1
209    else:
210       hash_performance[key] = arr_local
211       hash_count[key] = 1
212
213 #- [4.1] to get averages for each param set, divide sums by sample size cnt
214
215 cnt = hash_count[key]
```

```
216
217 for key in hash_performance:
218    arr_avg = hash_performance[key]
219    for idx in range(len(arr_avg)):
220       arr_avg[idx] /= cnt
221
222
223 #--- [5] Print results
224
225 OUT = open("spx500-results2.txt", "w")
226
227 labels="beta\tsigma\tomega\tsample size\tbuys\tbuys below last sell\tDelta return"
228 labels+="base return\ttrading return\tperiod (days)\tactive peiod\tsuccess rate\thold rate"
229 OUT.write(labels + "\n")
230
231 for key in hash_count:
232    strout = ""
233    for param in key:
234       strout += str(param) + "\t"
235    cnt = hash_count[key]
236    strout += str(cnt) + "\t"
237    arr_avg = hash_performance[key]
238    for idx in range(len(arr_avg)):
239       strout += str(arr_avg[idx]) + "\t"
240    strout += "\n"
241    OUT.write(strout)
242
243 OUT.close()
244
245
246 #--- [6] Visualizations
247
248 hash_xy = {}
249 hash_color = {}
250 for value in l_windowLow:
251    hash_xy[value] = []
252    hash_color[value] = []
253
254 for key in hash_count:
255    cnt = hash_count[key]
256    arr_avg = hash_performance[key]
257    for params in key:
258       buy_param = key[0]
259       sell_param = key[1]
260       window_low = key[2]
261       performance = arr_avg[2] # performance
262       if performance > 0.5:
263          color = [0, 0, 1] # blue
264       elif performance > 0.1:
265          color = [0, 1, 0] # lightgreen
266       elif performance > -0.1:
267          color = [0.6, 0.6, 0.6] # lightgray
268       elif performance > -0.5:
269          color = [1, 0.6, 0] # orange
270       else:
271          color = [1, 0, 0] # red
272       local_arr = hash_xy[window_low]
273       local_arr.append([buy_param, sell_param])
274       local_arr = hash_color[window_low]
275       local_arr.append(color)
276
277 mpl.rcParams['axes.linewidth'] = 0.5
278 plt.rcParams['xtick.labelsize'] = 9
279 plt.rcParams['ytick.labelsize'] = 9
280
281 for window_low in hash_xy:
282    z = np.array(hash_xy[window_low])
283    z = np.transpose(z)
284    color = hash_color[window_low]
285    plt.scatter(z[0], z[1], c = color)
286    plt.show()
```