Master of Science in Data Science

Dissertation

A Reproducible Approach to Equity Backtesting

Riaz Arbi

Supervisor: A/Prof. T. Gebbie

Mathematical Statistics,

Department of Statistical Sciences,

University of Cape Town, Rondebosch

2019

Abstract

Research findings relating to anomalous equity returns should ideally be repeatable by others. Usually, only a small subset of the decisions made in a particular backtest workflow are released, which limits reproducibility. Data collection and cleaning, parameter setting, algorithm development and report generation are often done with manual point-and-click tools which do not log user actions. This problem is compounded by the fact that the trial-and-error approach of researchers increases the probability of backtest overfitting. Borrowing practices from the reproducible research community, we introduce a set of scripts that completely automate a portfolio-based, event-driven backtest. Based on free, open source tools, these scripts can completely capture the decisions made by a researcher, resulting in a distributable code package that allows easy reproduction of results.

Keywords: Equity Backtesting, Reproducible Research, Event-based Backtesting, R, RStudio

Acknowledgements

This work would not have been possible without the ongoing emotional and practical support of my wife Simone Lilienfeld. I would like to thank Tim Gebbie for his technical guidance and advice on how to ensure that this technical implementation is sufficiently statistically robust. Finally, I would like to extend my gratitude to the statistical computing community at large for the generosity with which knowledge has been shared with me during the course of my research.

Table of Contents

1. Introduction

The question of how to identify stocks with abnormal returns is a large area of research, both in academia and in industry. This is understandable, given the profit potential of a good answer. A central tool in this area is backtesting, which is the testing of a stock-selection strategy on historical data. Constructing a backtest is difficult, and there are many design considerations that impact the final result.

Researchers must obtain data that is free of biases and as close to the actual data that would have been available to an investor at the time of a trade. They must also make decisions around how closely they want to model real-world trading, and how their trading behaviour might impact the market. They need to guard against overfitting the historical dataset. Finally, they need to ensure that their implementation is free of errors. The complexity of the undertaking makes replication very difficult, which often casts serious doubt on reported results.

We introduce a simple event-based backtester that is structured along the lines of a research compendium. Borrowing concepts and practices from the reproducible science movement, it aims to be totally transparent in its operation and fully amenable to customization. Our objective is to provide an open backtesting environment that can be distributed along with results to facilitate auditing and enhance credibility. In order to be fully auditable the entire backtest process from data acquisition to report generation is scripted using the R programming language. These scripts can be run again to replicate the result, and they can be inspected to verify the method.

In terms of functionality, it allows a researcher to address survivorship bias and look-ahead bias, it models market liquidity and transaction costs and provides tools for detecting backtest overfitting. It keeps data structures in simple tabular formats that can be readily consumed by other packages to facilitate extension.

This paper is organised as follows. Section 2 briefly introduces the practice of backtesting investment strategies in academia and industry, enumerates common methodological issues that compromise the validity of research, and discusses the replication crisis that exists in both domains. Section 3 describes the goals and philosophy of the reproducible research movement and its links to the open-source software movement. We focus especially on the R statistical programming language, the tidy approach to data organization and the use of scripting to facilitate reproducibility and transparency in scientific research. Section 4 introduces an opinionated backtesting template that is roughly compatible with the structure of a research compendium - that is, a template which constitutes a basic minimum viable unit of reproducible research. Section 5 provides detailed documentation of the codebase, along with design justifications.

2. Backtesting in Academia and Industry

2.1. History and distinction between academics and practitioners

Stock selection strategies have a long history. Professional bodies of knowledge, often captured in heuristics or rules of thumb, have existed since at least the early 20th century. For instance Graham, Dodd, and Dodd (1934) released the first edition of their *Security Analysis* textbook in 1934. As time has progressed, professionals and academics have sought to test these rules in a rigorous manner, and to implement automated trading strategies based on those rules.

Academics often frame these investigations as an attempt to identify whether a population of stocks subsetted along the lines of some attribute exhibit statistically significant differences in returns. Fama and French (1992), for instance, split the US stock universe by market capitalization and market beta (β), and then regress the cohorts against size, book to market, leverage and earnings yield to investigate whether any of these variables have explanatory power for outsize returns.

Practitioners, in contrast, tend to frame the question in terms of raw performance. Common measures include annualized return, excess return over some benchmark, risk-adjusted return, Sharpe Ratio or information ratio. There is a rich ecosystem of software packages, tutorials, online communities, books and blog posts that attempts to tackle this question (see Quantpedia (2018) for a list of software packages).

2.2. Differences in backtesting methods

These subtly different objectives result in different statistical approaches. Academics tend to focus on using statistical techniques to explain abnormal returns in terms of independent variables. Fama and French (1992) uses regression techniques to test a null hypothesis that one population is different from another.

Traders tend to focus on prediction rather than inference; trading rules range from being extremely simple, such as moving averages, to exceedingly complex implementations that use machine learning to automatically select features (Peterson (2017)). There are two main approaches to practitioner-led backtesting. Vectorized backtesting applies mathematical functions to *datexticker* arrays to quickly arrive at a solution, and event-driven backtesting 'steps through' time, presenting a trading engine with data in a spooling fashion. Each of these steps is a heartbeat. At every heartbeat the trading engine computes whether an action is required and acts accordingly. At the end of each heartbeat the engine moves a fixed increment forward in time and the next heartbeat commences; this repeats until the backtest date range ends.

Vectorized backtesting tends to be much faster, because computations can be done in parallel. But it is easier to allow look-ahead bias into the backtest because the computation has access to all the data at once - a simple cell offset in a function can completely nullify the result. Migrating a vectorized backtest to real-time trading is also very difficult, because new code needs to be written for the function to compute on streaming, rather than array, data. Event-driven backtesters are less likely to leak future information into computations because the engine is only presented with data that existed prior to the heartbeat period. It is also much easier to move into real trading, since all that needs to be done is the redirection of the data stream from a simulated source to a live source. The trade-off is that these backtesters are much slower, and much more complex to build.

Confusingly, all of the methods mentioned above fall under the term 'backtesting'. To avoid confusion, we adopt a broad definition from Bailey et al. (2014) when discussing the term :

> A *backtest* is a historical simulation of an algorithmic investment strategy.

2.3. Common areas between academics and practitioners

Both groups of researchers are concerned with whether their result is robust. For both groups, a well-designed backtest should yield results *out of sample* (OOS) which are similar to results derived *in sample* (IS). Unfortunately, backtesting is a very difficult operation to achieve correctly, and as a result it is quite easy to design strategies which perform well IS but poorly OOS (Bailey et al. (2014)).

Because of this common concern, there are common opinions on how to appropriately construct datasets and to assess whether a result is statistically robust.

2.3.1. Data quality requirements

Every backtest needs a historical dataset upon which to simulate a trading strategy. This dataset needs, at a minimum, to contain a time series of every member of the stock universe with price returns and any other attributes that may serve as inputs to the trading rule. This minimal dataset is necessary, but not sufficient, for a realistic backtest.

A realistic backtest also needs to address the following data quality issues:

1) Does the dataset contain *survivorship bias*? At any point in time, constituents of a stock universe are survivors. They are, by definition, the stocks which have not disappeared from the universe because of bankruptcy, delisting or mergers (Peterson (2017)). To base our data only on the history of this select group will mean that we apply the rule only to a very particular subset of a stock universe: those destined to remain listed. Since this implicit filter cannot be applied

going forward (i.e we cannot filter our existing universe to include only those that will be listed at some arbitrary point in the future), our IS result is likely to be different from our OOS result.

2) Does the dataset contain *look-ahead bias*? There are many ways that we can implicitly 'peek' into the future with historical datasets. A common pitfall is to use the 'close' price as representative of a day's average price. This, of course, uses future data because the close price cannot be known before the end of day. Another example is the inappropriate joining of fundamental and market data. Fundamental data is usually timestamped to the time at which the fact was true in the real world. However, these facts only become known at a (typically much later) future date. The Current Assets of a company at 31 December 2003, for instance, are only known at release of the annual financial statements, which might have been on the 27th February 2004. Joining this data point to market data on the date 31 December 2003 will introduce look-ahead bias to dataset (Peterson (2017)).

3) Is the dataset *complete*? Some datasets contain more information than others. Trice (2017) compares data for the S&P 500 Index across the free data vendors Google.com and Yahoo.com and finds discrepancies between them - the salient chart is reproduced below. In this instance, the discrepancy derives from the fact that Google contains a significant number of N/A values, which skew the summary statistics.

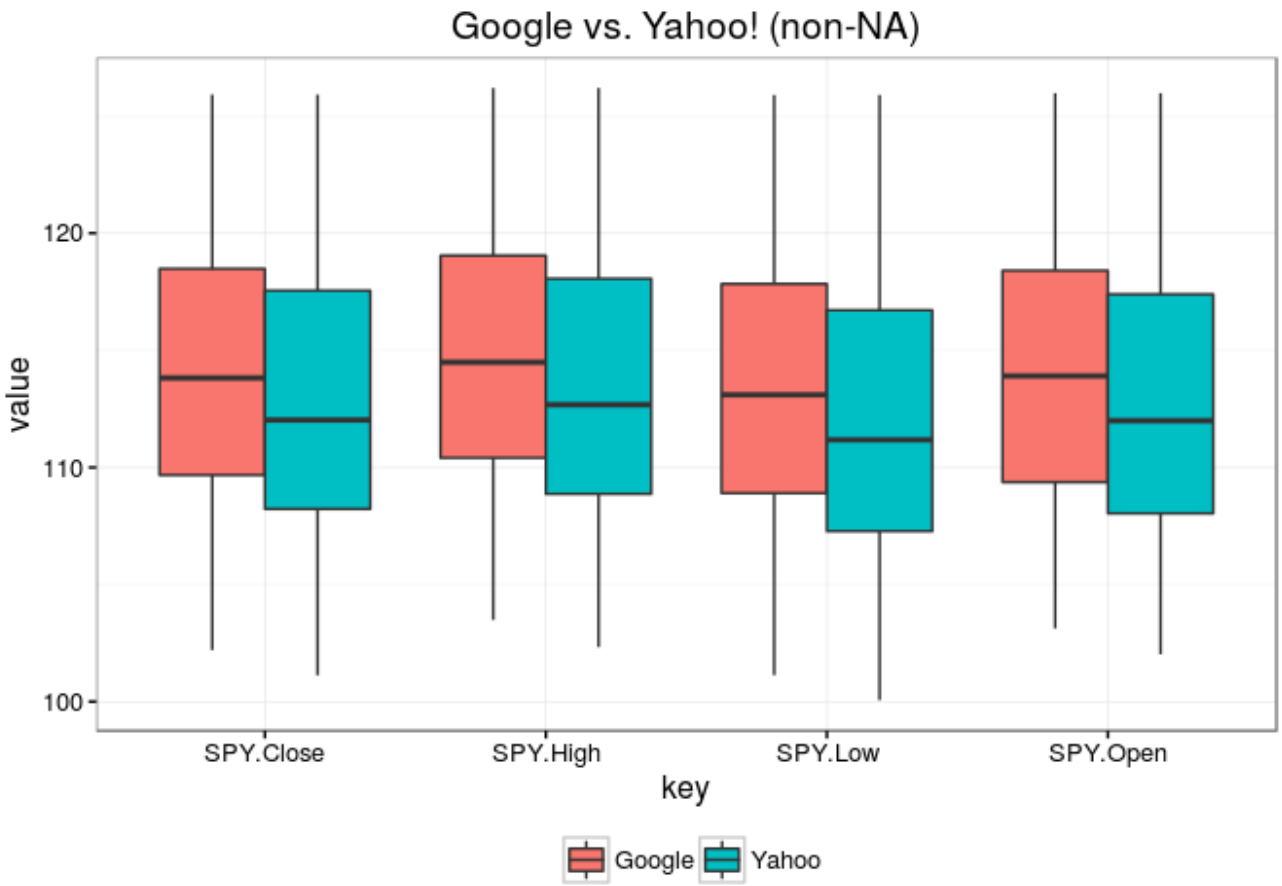

Figure 2.1: Data Discrepancies between Google and Yahoo! (Trice (2017))

4) Does the dataset contain *retroactive corrections*? It is common for companies to restate financial information after the fact. This can be because of changes in accounting rules, new information that comes to light, or to aid in comparison across major changes to the company structure. If these restatements are retroactively applied to a dataset, they can introduce look-ahead bias.

2.3.2. Modelling issues

Regardless of the backtest approach, the following issues need to be taken into account to assess the realism of the model:

1) Has *market impact* been taken into account? Return anomalies may be related to illiquidity. That is to say, the returns of a stock (or class of stocks) may appear to be abnormal, but there exists no liquidity in the market to make a trade at the modelled price (Harvey et al. (2017)). A realistic model will adjust for this by assuming minimal impact for only a small fraction of daily volume traded, increment the price if the trade size exceeds this volume, and respect the

bid/ask spread when deciding that a trade matches.

2) Have *transaction costs* been taken into account? Similar to market impact, transaction costs are associated with every real trade but often ignored in research. The transaction costs depend on the commission structure, which varies by broker and exchange. A class of stocks may appear to exhibit abnormal returns but all suffer from high transaction costs because they are, for instance, penny stocks that all require an exorbitant minimum commission for small lots (Loonat and Gebbie (2018)).

3) Failing to account for *short positions or portfolio bankruptcy*. Portfolios can be long only or allow for short positions. Long-only portfolios can either hold stocks or cash. Portfolios that can be short can, in addition to long-only portfolios, lend stocks that they do not own. This is done by borrowing a stock and selling it in the market, with an expectation that the share price will decline, at which point a trader can buy it back and return it to the lender. Short positions involve interest charges and margin requirements that need to be taken into account. Long-only portfolios cannot have negative positions. Similarly, portfolio cash balances cannot be negative if no leverage (which attract interest charges) is employed. The psychological impact of portfolio behaviour needs to be accounted for as well - if a strategy has an average excess return of 2%, but there are several periods in which the return is -50%, then the researcher needs to interrogate whether a reasonable trader would maintain those positions.

4) Data mining and *backtest overfitting*. When a researcher iterates over many strategies while recycling the same dataset, it is easy to overfit a model. Overfitting a model is when one selects rules which perform well IS but poorly OOS. That is, they model the noise in the training set, not the signal (Bailey et al. (2014)). This possibility is especially acute in the age of hyperparameter tuning and feature selection, where a computer can modify large numbers of parameters at once to select ones that perform optimally. This can be dealt with by keeping track of the number of trials and adjusting the hurdle for statistical significance accordingly, or testing for path-dependency, which is an indicator that a model has been tuned for a particular path-dependent series, as opposed to an underlying signal (Lopez de Prado (2013)).

2.3.3. Backtest overfitting

Data mining, in particular, deserves some elaboration. It is standard practice to divide a dataset into a training and test set, and to train models on the training set (Peterson (2017)). Once a profitable strategy has been found, it can be tested on the test set to determine how well it performs OOS. The rationale is that, since the test set was not used to train the model, good OOS performance implies that the strategy has legitimately detected some real signal in the data. Unfortunately, it is extremely easy to programatically tune backtest parameters to obtain a strategy that performs well IS. Bailey et al. (2014) show how easy it is to derive high Sharpe ratios over a random series by hyperparameter tuning. They claim that, with five years of data, only 45 trials can be tried before the expected IS Sharpe ratio is 1, whereas the expected OOS Sharpe ratio is 0. A fundamental issue

is that the likelihood of uncovering a spuriously high Sharpe ratio increases with the number of trials. The limited amount of data available (there is only one history) inevitably results in knowledge of the test set characteristics bleeding into IS training. In the presence of memory effects IS performance is *negatively* correlated with out-of-sample performance.

Lopez de Prado (2013) present several mathematical tools to test for backtest overfitting.

All of the tools rely on a method that Lopez de Prado (2013) named Combinatorially Symmetric Cross-Validation (CSCV). This method requires that the following steps be performed -

1) Divide a matrix M of the returns of a cohort of trials into an even number S row-wise partitions, and then form all possible combinations of the partitions.
2) Select all those combinations that contain $S/2$ partitions. That is, use those combinations which contain half of the returns, divided into segments. Call these combinations the set C.
3) Order each combination set of partitions C chronologically and join them to form a training set C_i^{train}. The remaining partitions in the matrix M are joined to form the testing set C_i^{test}, for this particular combination. C_i^{train} and C_i^{train} are the same dimensions, by design.
4) For each C, performance statistics are computed for each trial on the training set C_i^{train} and the testing set C_i^{test}. Rank the trials accordingly. At the conclusion of this step, for each combination, the tester has a set of performance statistics in sample and out of sample, and the corresponding in and out of sample trials rankings.
5) For each of these combinations C, determine the best in-sample strategy. Determine the relative rank of these best in sample strategies out of sample. Relative rank is defined as

$$Relative\ Rank = \frac{OOS\ Rank\ of\ IS\ Winner}{(Number\ of\ Trials + 1)} \tag{2.1}$$

The possible range of relative rank is therefore $[0 - 1]$. 5) For each of these combinations, compute the logit λ_i, where

$$\lambda = ln \frac{Relative\ Rank}{(1 - RelativeRank)} \tag{2.2}$$

Rankings which persist across in sample and out of sample performance will have a high logit value. 6) Collect all the logits from all the combinations, and determine the relative frequency of logit occurrence

across the C combinations. That is, determine a density function F where

$$\int dF = 1 \tag{2.3}$$

F can be visualised as a histogram of the logits along with a density function.

The first of the tools presented is the *Probability of Backtest Overfitting* (PBO), which is the probability that a strategy with optimal performance IS performs below the median OOS. Simply put, the PBO is the proportion of F along the domain $[-\infty, 0]$. The more of the density function that is below 0, the more likely it is best IS performer will do worse than the median OOS performer.

The second is the *Performance Degradation*, which determines the correlation between performance IS and OOS. This is useful for determining if memory effects are present, because it implies a relationship between the two ranks. It is simply the regression of OOS performance against IS performance, and is typically shown as a scatter plot with a regression line. A negative regression slope indicates the presence of memory effects.

The third is the *Probability of Loss*, which is the probability that the optimal IS strategy will deliver an OOS loss. It is the probability that the OOS Sharpe ratio will be less than zero.

The final one is *Stochastic dominance*, which determines whether the algorithm selection procedure is preferable to random selection of a strategy from the cohort. This test determines whether the distribution of ranks OOS stochastically dominates over the distribution of all OOS returns. If it does not, then the selection criteria is no better than random.

2.4. The Replication Crisis

While total data fidelity and perfect model specification are good goals in theory, the reality is that compromises often need to be made. It is often simply impossible to model the exact market impact a trade will have; data may not be available in sufficient detail; information on when fundamental data was released may simply not exist.

Inevitably, a researcher needs to make judgment calls. A typical backtest involves many judgment calls; these permeate the entire analysis chain, from deciding what source data to use all the way through to deciding which metrics to use to evaluate strategy success, and are difficult or impossible to document.

2.4.1. The crisis in academia

It is a standard aspiration for researchers to disclose all the judgment calls made during the backtest process, but these disclosures are invariably incomplete. This makes validation by other researchers very difficult. At the extreme, independent validators may arrive at a completely different conclusion (Hou, Xue, and Zhang (2017)). With ambiguity on both sides it is unclear whether the discrepancy is due to differences in underlying data, preparation methodologies, mathematical errors or some other factor unrelated to the question at hand.

The academic discipline of anomalies research has grown into a substantial body of contradictory claims. These contradictions are mirrored in the business of investment management by competing investment philosophies, each with its own library of peer-reviewed papers (Damodaran (2012)). While markets may very well accommodate contradictory drivers of returns, it is often difficult to discern whether differing claims are the result of genuine facts or the result of methodological differences.

Hou, Xue, and Zhang (2017) attempt to replicate the entire anomalies literature to identify which results can actually be confirmed. They attempt to replicate 447 anomalies and find that between 64% and 85% of them are insignificant. In other words, they suspect widespread misuse of statistical analysis to make claims that are not, in fact, true. This is possible, in part, because practitioners have a large degree of discretion in determining every aspect of the backtesting analysis chain. Hou, Xue, and Zhang (2017) attempt to set out a common set of replication procedures to standardise research output, including (but not limited to):

1) Specifying datasets: Compustat Annual and Quarterly Fundamental Files; Center for Research and Security Prices (CRSP).
2) Specifying breakpoints: Use NYSE breakpoints.
3) Use value-weighted, not equal-weighted portfolios.
4) No sample screening (i.e. don't exclude stocks because of some arbitrary cutoff).
5) For annually-composed portfolios, re-sort at the end of June.
6) Incorporate fundamental data four months after valid date to impose a lag and guard against look-ahead bias.

Many of the above guidelines are designed to avoid outsize effects from micro-capitalization shares, which form 3% of market value but comprise 60% of the total number of stocks. These stocks often suffer from high transaction costs and low liquidity, which muddy the waters when testing for a particular factor.

2.4.2. The crisis in industry

Unfortunately, these guidelines are not directly applicable to backtesting for trading. Traders are primarily interested in real-world excess returns; the use of CRSP prices to compute returns, for instance, does not translate into real cash balances because CRSP are not available on a real-time basis.

The approach taken by developers of backtesting software has been to build event-based engines which are capable of easily switching between simulated and real-life trading. The rationale is that, as long as the data is from the same source, signal generation, trade matching, transaction costs, account balances and portfolio composition can operate unmodified. This somewhat mitigates concerns around replicability - if it works on historical data, simply switch the data source to live data and don't ask too many questions.

The caveats are as follow. Firstly, many of these engines are proprietary (Quantpedia (2018)) and the actual mathematics of what is going on is not available, which means that the researcher has to take it on faith that the implementation (and source data) is correct. If IS results differ from OOS results it is difficult to debug where the issue is: it could be data quality, poor implementation, or overfitting. One cannot be certain that one's returns are being driven by the reasons one thinks.

There do exist many open-source backtesting libraries, most notably the `zipline` python package (Zipline (2018)), the Quantopian trading platform (Quantopian (2018)), which is built on top of `zipline`, and the R package `quantstrat` (Carl et al. (2018)).

Although these packages make it possible to inspect the entire analytical chain they do not provide historical data. Historical data is difficult (and expensive) to procure, and the same concerns raised in 2.3.1 still exist. In many ways, these issues are worse because these packages default to free data sources such as Yahoo! and Google Finance, which do not document index constituent membership changes, retroactive data modification or provide any guarantees of veracity. Bias-free data must be procured, cleaned and put in an appropriate format beforehand; documentation of how to conduct these operations is left as an exercise to the researcher. Even if a researcher can procure well-formed, comprehensive data, there exists no toolset to transform that data into an appropriate format for ingestion to any particular software package, and no tools to investigate the health of the data with reference to the biases mentioned in 2.3.1.

These shortcomings in free data sources drive areas of research among enthusiasts: the documentation on `zipline`, Quantopian and `quantstrat` focus overwhelmingly on price signals (algorithms relying on non-market data such as fundamental, macroeconomic or alternative data are scarce) and single-stock or fixed-basket portolios. This makes them much less suitable for research into general strategies that can be applied across an entire universe or research which relies on more than just market data.

There is something of an irony here. It is difficult for industry practitioners to translate findings in anomalies research into actionable strategies because of incomplete documentation and it is difficult for academics to use production-level backtesters to search for new anomalies. Part of this is because they have differing objectives. But another hurdle is the steep learning curve required to understand what is going on 'under the hood' of a production trading engine. Both `zipline` and `quantstrat` implement the Object Oriented Programming (OOP) paradigm, which focuses on objects rather than procedures (Van Roy (2009)). This is in contrast to a procedural approach, which focuses on operations (i.e taking an input, performing an operation, and producing an output). OOP programs scale well, and enable encapsulation: knowledge of the implementation of an object is not necessary for its use. This provides convenience and lowers barriers to entry for new users, but comes at a cost: linear mapping of the sequence of procedures that constitute an operation can be difficult.

Procedural programs, on the other hand, can typically be read from top to bottom, and the procedural flow is often quite easy to follow. Unfortunately, as a program's size increases, the complexity and brittleness of a procedural program increase super-linearly, as the procedure has to incorporate more and more subroutines to manage the various inputs and outputs of a particular part of the program.

3. Reproducible research and the R programming language

Problems with replication are not unique to backtesting. Science as a whole is experiencing a replication crisis.

Despite scientific research being an accretive process, involving verification and building upon earlier datasets and findings, the computational work done by researchers is often poorly documented, or absent (Stodden et al. (2013)). This introduces the risk that views or principles that are considered to be scientifically verified may, in fact, be false.

Gaps in documentation allow bad reasoning, calculation errors or spurious results to creep in to the corpus of knowledge of a discipline because peers are forced to take it on faith that a researcher has a proper understanding of underlying mathematical concepts, and that workings have been thoroughly cross-checked before release.

Worse still, because replication is difficult with poor documentation or insufficient data, incorrect beliefs can persist for years, or even decades (Munafò et al. (2017)). *P-hacking*, or the selection of data and finessing of results to conform to significance tests of 5% or lower, has resulted in a situation where a research finding is as likely to be false it is to be true (Ioannidis (2005)).

Up until the late 20th century, practical constraints limited the amount of supporting material a researcher could distribute along with a research paper. Documentation of every single mathematical operation conducted with pencil and paper would be extremely laborious; duplication and distribution of source data and computations would be extremely expensive. However, in recent years, as computational power and storage has become increasingly cheap and available, there have been persistent calls for a review of the way that scientific research is packaged and presented (Stodden et al. (2013), Koenker and Zeileis (2009)).

3.1. Reproducible research terminology

In response to these calls, a taxonomy of reproducibility has emerged, allowing peers to effectively categorize research.

This taxonomy allows us to distinguish *reviewability* (which assumes mathematical competence and focuses on reasoning) from *reproducibility* (which allow the extension of the review into assessment of mathematical competence and quality of data).

Table 3.1: Common Terminology in Reproducible Science (Stodden et al. (2013))

Term	Description
Confirmable	Main conclusions can be attained independently
Reviewable	Descriptions of methods can be independently assessed and judged
Replicable	Tools are made available that would allow duplication of results
Auditable	Sufficient records exist (perhaps privately) to defend research
Reproducible	Auditable research is made openly available, including code and data
To Verify	To check that computer code correctly performs the intended operation
To Validate	To check that the results of a computation agree with observations

This paper adopts the definition of reproducibility as it has been done in 5.4, which is to say, it refers to research output being bundled and distributed with well-documented, fully-transparent code and data that allows a peer to fully review, replicate, audit and thereby confirm the results of a body of work (Stodden et al. (2013)).

3.2. The relationship between reproducible research and open source software

Open source software development principles exhibit properties which are useful to practitioners of reproducible research.

In general, software is written in text files, by humans, in a particular programming language. This *source code*, which is made up of nouns, verbs, adjectives and operations, is *compiled* or *interpreted* by another program into binary *machine code* (which is a long sequence of 0 and 1 digits and is not readable by humans). Binary machine code is passed to a computer's central processing unit (CPU) for execution. The software for the popular spreadsheet program Microsoft Excel, for instance, would be written by a group of humans in a human-readable programming language. These text files would be compiled by a compiler program into a binary executable program, which is distributed to consumers who can execute the binary using the CPUs in their computers. When someone clicks the Excel icon on their computer to 'launch' the program, they are executing the binary.

Open source software is software for which the source code is made publicly available. Microsoft Excel is closed source, and the source code is not made available. Source code can be read by humans, binary cannot. Changes to source code can be meaningfully interpreted by humans, because one can read the changes in plain text. Changes to binary code cannot be interpreted by humans, because the changes are simply alterations to a very long sequence of `0` and `1` digits. It is trivial to compile source code into binary code. It is extremely difficult to accurately decompile binary code into source code, and the tools are not readily available (Li (2004)).

The text-based nature of source code makes all source code reproducible (and, by corollary, auditable), and code-versioning tools make the tracking of changes in source code very simple. The implications of this are twofold. Firstly, a researcher using open source software for computation can be assured that the full software stack will be open to scrutiny. That is, at the limit, an auditor could inspect every single line of code from some arbitrary 'open starting point' up to verify that the computations are correct. This 'open starting point' is from the operating-system level up for Linux-based machines, and from the application level up for Mac and Windows-based machines.

Secondly, and perhaps more importantly, the open source software community has decades of experience in defining good practice for creating and maintaining large, open, verifiable and repeatable environments. These principles are often quoted in the folkloric 'Unix Way', which manifests in epithets such as "text is the universal interface" (Baum and Sirin (2002)), "each unit should do one thing and do it well", and "build things to talk to other things" (Raymond (2003)). Reproducible research practice is built on the same principles - simple, readable text-based data and code where possible; clear separation between data, code and results; complete transparency; and portability by design (see Baum and Sirin (2002), Bache and Wickham (2014), Wickham (2014)).

Historically, there has been close collaboration between the open source software and the free software communities. This means that, as a rule, most open source software is free or has a free analogue which promoted accessibility to sophisticated tools. The core tools of data science, for instance (such as the python and R programming languages, the Apache Foundation of big data processing packages and the Jupyter and RStudio interactive development environments), are all open source and free. This means that the cost of replication is not prohibitive: in general, reproducible research should be reproducible for free on commodity hardware.

3.3. The suitability of R for reproducible research

The R programming language was originally conceived of as a project to build an open source statistical programming environment. In development since 1997, base R is now a mature ecosystem comprising a scriptable language, a graphical rendering system and a debugger (R Core Team (2018)). A large community of third party developers (some commercial, most free) have extended base R with over 13,000 packages, including a fully-fledged interactive development environment (Rstudio), interactive dashboarding web server (Shiny) and bindings to LaTex for rendering of publication-ready LaTex documents from markdown (knitr and rmarkdown) (RStudio Team (2015)). Because of its roots in open source, it adheres to many open source software conventions. For instance-

- All source code is freely available.
- All development work in R is done in plain text files, which are transparently human and machine readable.

- The R project has adopted the GNU General Public License version 2, which "does not restrict anyone from making use of the program in a specific field of endeavour" (R Core Team (2018)). This free availability means that anybody with a commodity computer and an internet connection can install and use R.

Although the reproducible research community is principle-based and language-agnostic, the free, text-based nature of the R language have made it a good candidate for reproducible work (Gentleman and Lang (2004)). Marwick, Boettiger, and Mullen (2018) review the concept of a research compendium - a bundle of research, method and data that facilitates replication - and explore how research done in R can be packaged into compendia. They argue that a compendium should follow these three principles -

1. Organise files according to a prevailing standard so that others can immediately understand the structure of the project.
2. Maintain a clear separation between data, method and output. Separation means that data is treated as read only, and that all steps in getting from source data to output is documented. Output should be treated as disposable, and rebuildable from the programmatic application of the method to the data.
3. Specify the computational environment that was used to conduct the research. This provides critical information to a replicator about the underlying tooling needed to support the analysis.

These principles are intended to ensure that a certain input (the data) and a certain operation (the method) results in a deterministic output (the analysis). With these details out of the way, an auditor can focus on verifying the correctness of the reasoning (i.e should we use this theory?) and of the implementation (i.e are there any errors in the math?).

The richness of the R ecosystem means that one can complete the full research life cycle from data-loading to publishing without leaving the R ecosystem (Baumer et al. (2014)). Data can be loaded, cleaned, transformed, modelled and aggregated in R. The results can be written up in RStudio - in fact, the code and write-up can be combined into a single text document using the RMarkdown format - and exported to a wide range of academically accepted typesets and formats (such as Tex, Microsoft Word, Markdown and HTML). Because R is scriptable, R scripts can also be used in production environments, where results can be periodically recomputed and handed on to some other production process.

This property dramatically simplifies the complexity of building a compendium and the required skill-set of a replicator. For instance, a single-platform research compendium can be assessed by anyone

with knowledge of that single environment. Multi-platform compendia, on the other hand, need replicators well-versed in each platform, which dramatically reduces the pool of potential replicators.

Upon publication, code-versioning tools such as Git allow the public to view all future changes to the code or data transparently, and, if code changes over time, to assess the impact of those changes on the result.

3.4. Tidy extensions to base R

The open source and reproducible science principles of interoperability, readability, modular code and common standards have been extended into the practice of data manipulation by Wickham (2014). The collection of R packages which fall under this banner are collectively known as the 'Tidyverse'. These packages provide a common, idiosyncratic syntax for data ingestion, munging, manipulation and visualization that are opinionated in their expectations about data structure and focused on enabling readable code (Wickham (2017)). Wickham defines 'tidy' data as data that intuitively maps the meaning of a dataset to its structure -

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

Whether there is a single optimal data structure is up for debate. What is not up for debate is that the adoption of a common standard for structures allows researchers to make assumptions about the meaning of data before engaging with the content, i.e simply knowing that data is tidy means that a researcher knows that each column maps to a variable, and each row an observation, prior to looking at the data. Setting standards enables the decoupling of processes - process (a) can produce some piece of analytical output with no knowledge of what it will be used for. As long as it is tidy, any downstream process can consume that output on the understanding that is confirms to the tidy standard and take the analysis further. In this way, the tidy data concept pushes the Unix Way concept of 'build things that talk to other things' into the practice of structuring data for reproducible research.

3.5. The potential for reproducible research to address the replication crisis in anomalies research

We have explored how the sheer number of judgement calls involved in building a rigorous back-test inhibit replicability, both in academia and in a professional setting. Obstacles differ between them; academics suffer from incomplete documentation, professional traders suffer from a dearth of simple, transparent toolkits to automate the entire analysis chain. Both communities have serious shortcomings when it comes to transparent data acquisition and preparation.

We have also shown how these obstacles are being addressed in the scientific community at large by the concepts of reproducible research, the research compendium, and tidy data. These concepts are supported by certain technologies: cheap commodity computing, free and open source scientific computing software and text-versioning tools, and the R programming language and ecosystem for acquisition-to-publication analysis.

It stands to reason that the concepts and tools of reproducible research could be used to address the issues in backtesting at large. These tools would have to -

1) Be able to run on a typical researcher's computer.
2) Produce artefacts (models and data objects) that are portable across the academic/industry divide.
3) Encompass the full analysis chain from acquisition through to reporting, with a particular focus on raw data processing.
4) Accommodate deep customizability without requiring excessive complexity of code.
5) Expose underlying workings in a transparent, procedural manner to an auditor.

It is crucial to bear in mind that the objective of such an implementation would not be to build a rigorous backtest. Rather, its immediate goal would be to build a totally transparent engine that places ease of audit first and foremost. Borrowing a concept from the open source community, it is assumed that stability and correctness of implementation are derivative properties of a rigorous, continuous review process. If researchers use it, bugs will be found, and fixed, and it will become ever more useful.

These objectives are possible within the current capabilities of the R ecosystem right now.

1) R and all R packages run on any x86 processor.
2) Research compendia are structured with portability in mind. The scripting capabilities of R allow an end-user to run the entire backtest non-interactively if desired.
3) R, RStudio and the Tidyverse are a mature toolkit for end-to-end research.
4) Like any programming language, R accommodates the creation of parameter variables which can be passed to user-defined functions. These parameters can be altered in a transparent manner. Enhancements and additions to the engine code are completely transparent and can be easily tracked with versioning tools such as git.
5) Although R has some OOP concepts, it is primarily a procedural language, which lends itself to a linear, top-to-bottom logical code flow.

In the following section we unpack our implementation of a reproducible backtester.

4. A Reference Implementation of a Data Scientific Equity Backtester in R

We provide a working example of a reproducible equity-backtesting workflow that covers the full data-processing chain from acquisition to reporting.

All code is available at https://github.com/riazarbi/equity_analysis (Arbi (2019)).

The data processing scripts can consume multiple data sources, for multiple equity indexes, for any timeframe, and for arbitrary data attributes. The implementation is index-based. That is, data acquisition begins with obtaining constituents for an equity index at a point in time, and then proceeds to obtaining metadata and timeseries attributes for the constituents.

Data queries are scripted, and therefore fully reproducible and repeatable. All queries are saved to a log directory. Reference datasets are generated from the log directory. We use rudimentary log compaction and data versioning to enable point-in-time analysis and check data quality, but the tooling for this analysis is beyond the scope of this paper.

Datasets are read into memory for backtesting, which is done in a step-through fashion. That is, the procedure loops through the backtest range in fixed increments. In each loop, it is presented with new data that triggers appropriate trading actions. These actions are saved to various transactional files on disk.

This backtester has been built with multi-trial, parameterised backtesting in mind. That is, certain global parameters are set (backtest start and end date, target index, etc.) and then multiple user-specified algorithms are tested at once. We adopt the multi-trial approach to facilitate control for backtest overfitting, as discussed in Section 2.3.3.

We have built rudimentary support for transaction costs and slippage; these can be extended in a transparent manner.

Input data has been segregated from the code and from the results. It is anticipated that a researcher will delete the input data prior to distribution to avoid proprietary vendor licensing issues; this raw data should be regenerated from source using supplied query scripts.

4.1. Intended audience

This project should be useful to:

- Finance students at all levels wanting to conduct statistically rigorous equity backtests.

- Post-graduates and academics looking to conduct research on a common platform to facilitate replication and peer review.
- Research professionals looking to build out an in-house backtesting environment for proprietary equity investment strategies.
- Equity researchers looking for a bridge between Excel-based research and `R` or `python`.

4.2. Intended use case

It is intended that a researcher will clone this repository to a local directory and then create an RStudio project in the project root directory. Sourcing the script `scripts/0_all.R` will result in several dialogs, after which the script will run the entire backtesting workflow from data acquisition to cross validation in an automated fashion. The scripts will output actions to the console as they step through their procedures.

The scripts which `0_all.R` calls can all be run stand-alone, and can be edited transparently. A common workflow is to copy the project directory to a removable flash drive, run `data_processing/1_bloomberg_to_datalog.R` on its own on a Bloomberg terminal, and then return to an analyst's workstation (or docker container) to continue analysis. The researcher will then edit the `scripts/parameters.R` file to specify backtest behaviour, add a series of algorithm scripts in the `trials` directory and run the `0_all.R` again. The results of the backtest are saved to the `results` directory.

These results can be used to create a research paper, which can be saved to the project root. Prior to release, the researcher can delete the source data for licensing reasons (but retain the query scripts!) and publish the entire compendium to a code hosting site such as GitHub. Distributing query scripts along with the code means that a replicator need not have any ambiguity around the method of data collection and cleaning.

4.3. Included data

This repository does not include any equity data, but it does include working scripts to automatically extract data from data vendors, and to save that data in a well-formed way. It is assumed that the user will acquire data using the scripts which have been included in this repository. This will only be possible if a user has access to the relevant data services - namely Bloomberg, DataStream or iNet.

We also include a script for generating simulated index datasets, complete with constituent lists, metadata, market and fundamental ticker timeseries. This script accepts parameters from the `parameters.R` file to control various characteristics of the stock universe. The resulting simulated datasets can be used to explore the functioning of the code and test that it is working as expected.

4.4. Future development

This project is ongoing, and many rudimentary features will be refined as time unfolds. For this reason, we recommend that users fork the project prior to use, in order to lock their research to a particular version of the compendium. Users are encouraged to submit pull requests on GitHub for bug fixes and to add features.

4.5. Intellectual property statement

5. Detailed Documentation, With Design Justifications

Because the workflow is linear and procedural in design, this documentation will proceed in a linear, procedural fashion. This design facilitates ease of understanding as regards the process flow, but means that code can be brittle and repeat in places. After familiarising the reader with the directory structure, we will work through the code in a step-wise fashion from data acquisition through to final reporting.

5.1. Design considerations

Because this compendium is designed to be run by researchers on commodity consumer hardware, our design choices sacrifice speed in order to keep memory usage low. We have split the backtesting procedure into five stages - querying, data processing, backtesting, cross-validation and reporting. Each stage expects to read on-disk files from a pre-specified location and writes files to a prespecified location. Often the first script in a stage will clear the environment in order to release memory.

This has the added benefit of making these stages asynchronous. Processing data does not require a new query to have been conducted; cross validation does not require new trials to be run. These stages can even be run on different machines that share network drives - although the benefits will only be apparent if each stage's processing time is longer than the network transfer time.

Where possible, we have made disk read/write optimizations. This includes converting `.csv` files to `.feather` files, which are much faster to read and write, and parallelizing select pieces of code. Ticker-wise data pipelining is multi-threaded. The trading engine is single-threaded, but asynchronous. That is, several trading engines can simultaneously consume the `trials` directory's contents and backtest in parallel without collision. Note, however, that since the random seed is set globally, each trading instance will initialize the same seed. The implication is that if the portfolio-weighting algorithms make use of random number generation, then each instance will produce the same random numbers and therefore produce copies of each other. Multi-instance trading should only be done on algorithms that do not rely on random number generation.

All the actions required to conduct a complete backtest are captured in plain text files. This means that every single computation and data manipulation can be inspected and its functioning verified. The entire code base can be versioned using a software versioning tool such as git. A researcher can fork this codebase, modify it, and push the modifications back to the origin. All of these changes are immutably recorded; future researchers can always see exactly what has changed, who has made the changes and what the reason for the change was.

5.2. Launching a Docker image

In order to ensure reproducibility the repository includes a `Dockerfile`. The codebase has been written with broad compatibility to the `rocker/tidyverse` docker image. Because of this, our `Dockerfile` uses `rocker/tidyverse` as a base image and merely adds a few specific packages. Rocker is the most popular docker image download on docker hub. Any machine that has docker installed can build a Docker image from this `Dockerfile` which will run the entire codebase with no additional configuration required.

For a Linux-based computer with docker installed, the following steps should suffice to begin interacting with the codebase -

1) Open a terminal
2) Enter the following command: `docker run -p 8787:8787 -e PASSWORD=complicatedpassword riazarbi/equity_analysis`
3) Open a web browser
4) Navigate to `localhost:8787`
5) You should be prompted for a username and password
6) Enter the username `rstudio` and password `complicatedpassword`
7) Once inside Rstudio, launch a terminal `Tools -> Terminal -> Launch New Terminal`
8) Clone the `equity_analysis` repository `git clone https://github.com/riazarbi/equity_analysis`

5.3. Directory structure

The directory structure loosely conforms to Marwick, Boettiger, and Mullen (2018)'s definition of a valid R packages structure, and most closely resembles the author's example of an intermediate-level compendium. The root directory contains the following files and directories:

- A `README.md` file that describes the overall project and where to get started.
- A `Dockerfile` allows a user to build docker image identical to the researcher's environment.
- A `DESCRIPTION` file provides formally structured metadata such as license, maintainers, dependencies, etc.
- A `LICENSE` file specifies the project license. This information is duplicated in the REQUIREMENTS file but a separate LICENSE file is expected for automatic parsing by hosted git repositories such as GitHub.
- An `R/` directory which houses reusable functions.
- A `data/` directory to house raw data and derived datasets.
- A `scripts/` directory to house data processing and backtesting scripts.
- A `trials/` directory houses algorithms.
- A `results/` directory houses the result of the applications of algorithms housed in `trials/` on the datasets housed in `data/`.

The `data/`, `trials/` and `results/` directories are generated at runtime if they do not exist.

`Complete directory structure:`

```
. equity_analysis/
|    data/
|    |   datalog/
|    |   datasets/
|    |   |  constituent_list/
|    |   |  |  metadata_array.feather
|    |   |  metadata_array/
|    |   |  |  metadata_array.feather
|    |   |  ticker_fundamental_data/
```

```
|    |   |  |  ISIN_1.feather
|    |   |  |  ISIN_n.feather
|    |   |  ticker_market_data/
|    |   |  |  ticker_1.feather
|    |   |  |  ticker_n.feather
|    R/
|    |   |   data_pipeline_functions.R
|    |   |   set_paths.R
|    |   |   trading_functions.R
|    |   |   backtest_trading_functions.R
|    |   |   live_trading_functions.R
|    scripts/
|    |   parameters.R
|    |   0_all.R
|    |   1_query_source.R
|    |   2_process_data.R
|    |   3_load_data.R
|    |   4_run_trials.R
|    |   5_report.R
|    |   6_cross_validate.R
|    |   data_processing/
|    |   |   1_bloomberg_to_datalog.R
|    |   |   1_simulated_to_datalog.R
|    |   |   2_datalog_csv_to_feather.R
|    |   |   3_constituents_to_dataset.R
|    |   |   3_metadata_to_dataset.R
|    |   |   3_ticker_logs_to_dataset.R
|    |   data_loading/
|    |   |   load_slow_moving_data.R
|    |   |   compile_data_report.R
|    |   trading/
|    |   |   sample_trials/
|    |   |   trade.R
|    |   reporting/
|    |   |   data_quality.Rmd
|    |   |   tearsheet.Rmd
|    |   |   cross_validate.Rmd
|    trials/
|    results/
```

Marwick, Boettiger, and Mullen (2018) recommend an `analysis/` directory to house all scripts, reports and results. Because our code applies parameters to an arbitrary number n algorithms to compute n results, we have split this `analysis/` directory into a `scripts/`, `trials/` and `results/` directory.

In short, scripts in the `scripts/` directory pull data into the `data/` directory and process that source data into tidy datasets. They then run the algorithms in the `trials/` directory on `data/datasets` in order to generate results, which goes into the `results/` directory. A detailed data flow diagram is rendered below.

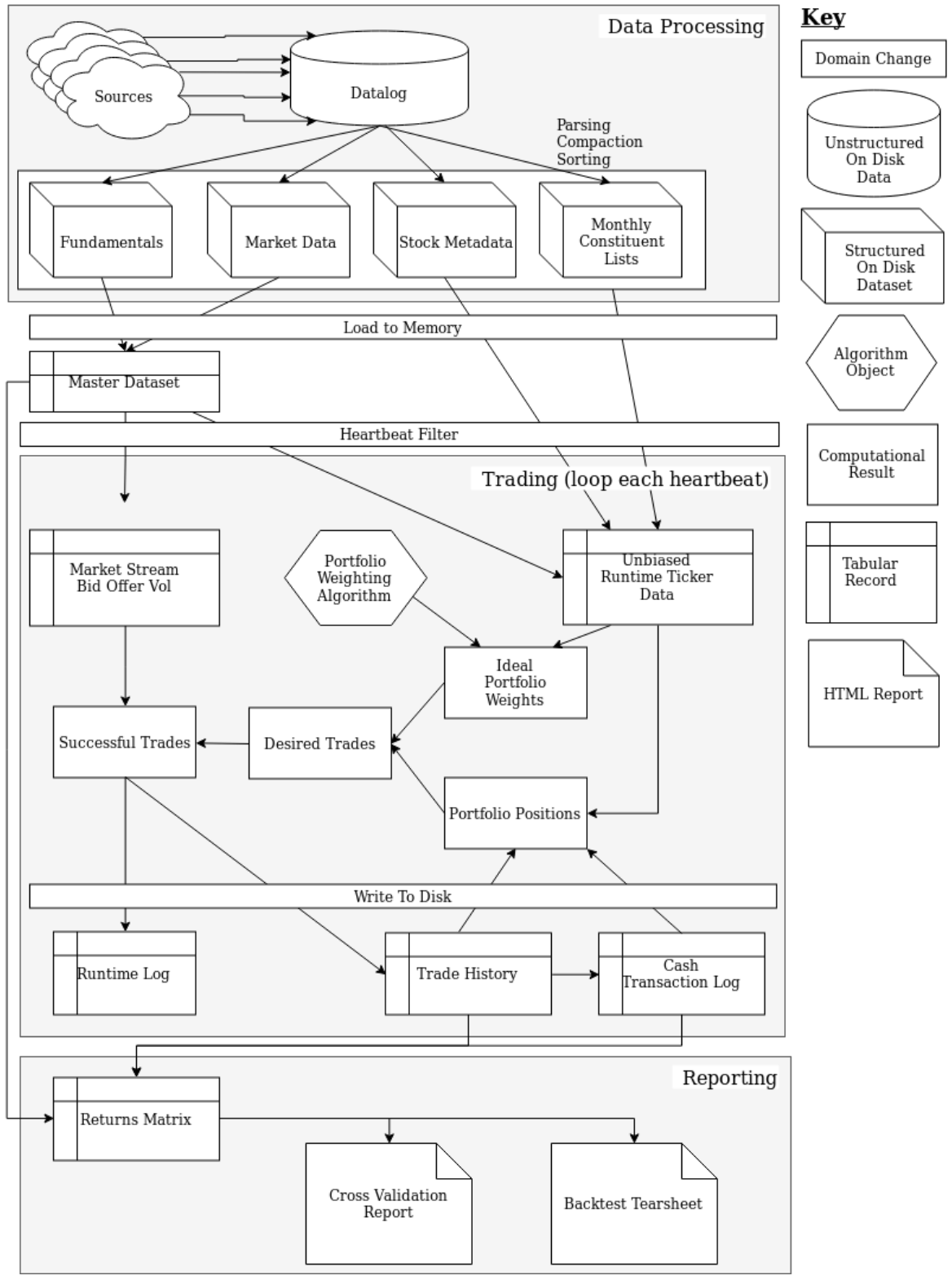

Figure 5.1: Data Flow Diagram

5.4. Parameter setting - `parameters.R`

The `scripts/parameters.R` script contains all the parameters for the backtest session. This file, along with the `trials/algorithm_X.R` scripts, are the only ones that a user changes to run unique backtests. They are R files, rather than markdown or yaml, because it enables us to compute some parameters based on the values of other parameters.

```
Relevant parts of the directory tree:

. equity_analysis/
|    scripts/
|    |   parameters.R
```

This file is sourced in various other scripts. It determines the structure of simulated datasets, the characteristics of the trading environment, and the behaviour of the trading engine. It has the following sections -

1) Reporting Parameters: here we set the risk-free rate, which is needed for computing the Sharpe ratio.
2) Simulated Data Parameters: characteristics of simulated data such as universe size, index size, start and end of simulation, price and earnings growth.
3) Trading parameters: mode (backtest or live), heartbeat duration, rebalancing periodicity, target index, start and end of backtest.
4) Lag adjustment parameters: whether or not to conduct lag adjustment on fundamental data, and whether to try determine which fundamental metrics need adjustment.

5.5. Data acquisition - `1_query_source.R`

The objective of the scripts called in `scripts/0_query_source.R` is to query a data source (such as Bloomberg, Datastream or iNet) and save the results in an appropriate format in the `data/datalog` directory. This directory can grow without limit. It can be deleted and regenerated from the query scripts. It will be mined by downstream scripts to create datasets for backtesting.

```
Relevant parts of the directory tree:

. equity_analysis/
|    data/
|    |   datalog
```

```
|    scripts/
|    |   parameters.R
|    |   1_query_source.R
|    |   data_processing/
|    |   |   1_bloomberg_to_datalog.R
|    |   |   1_simulated_to_datalog.R
```

We include two sample scripts. One for querying Bloomberg, the other for generating simulated dummy data for demonstration purposes. These scripts can be found in the `scripts/data_processing/` directory. This directory adopts a sequential naming convention because they are intended to be run sequentially, but only one query script is intended tobe run at any one time. `0_run.R` will prompt the user to select a query script to run.

`1_Bloomberg_to_datalog.R` follows the broad procedure of collecting all the data required to build a dataset. The query script performs the following steps in a non-interactive manner -

1) Connect to the vendor via an API (in the case of Bloomberg, this is via Armstrong, Eddelbuettel, and Laing (2018)'s R package Rblpapi).
2) Define a target index. For instance, we extract JALSH, which is the Johannesburg Securities Exchange (JSE) All Share Index.
3) Define how far back to extract constituent lists.
4) Extract monthly constituent lists for the index and save to the `datalog`.
5) Read in all constituent lists and compile a unique list of tickers. These will include delisted shares as well as survivors .
6) Define required ticker metadata. At a minimum, the ticker code and the ISIN is required. Market data is extracted with the ticker code; fundamental data is extracted with the ISIN.
7) Extract and save metadata for all tickers.
8) Extract and save market data for each ticker based on ticker code.
9) Extract and save fundamental data for each ticker based on ISIN.
10) Reload metadata arrays and give critical metrics (`fundamental_identifier` and `market_identifier`) common names. This is so that the downstream scripts know which fields to use to join fundamental data to market data during data loading.

5.5.1. File naming conventions

All queries are saved to the `data/datalog` directory under the following naming convention -

```
TIMESTAMP__source__data_type__data_label
```

Where

- `TIMESTAMP` is millisecond time, as calculated by `as.numeric(as.POSIXct(Sys.time()))*10^5`.
- `source` is the data source, such as `BLOOMBERG`.
- `data_type` is one of `constituent_list`, `metadata_array`, `ticker_fundamental_data` or `ticker_market_data`.
- `data_label` is the identifier of the particular object.
- `constituent_list` types are of the form `YYYMMDD_INDEX`;
- `metadata_array` types are simply the `INDEX`;
- `ticker_market_data` are of the form `TICKER`,
- `ticker_fundamental_data` are of the form `ISIN`.

Using this file-naming convention is essential. It allows us to save many versions of the same data in a manner that is very fast to filter for downstream processing. Downstream scripts use these filenames to mine the datalog and build master datasets. An example of a well-formed datalog file name is -

```
152874575747280__bloomberg__ticker_fundamental_data__ZAE000018131 Equity.csv
```

5.5.2. File contents conventions

Datalog files should also be tidy. That is, each file related to a particular object in the real world (i.e. an index, a metadata array, or a ticker), and the dat is flat and tabular, with each row being an observation (i.e. a date) and each column an attribute (e.g. CLOSE, LOW, HIGH etc). An example of a tidy `fundamental_data` file is -

date	BS_ACCT_NOTE_RCV	BS_INVENTORIES
2014-12-31	NA	1182822
2015-06-30	782550.2	1158028
2015-12-31	NA	1260626
2016-06-30	605207.4	1048830
2016-12-31	NA	1057549
2017-06-30	522236.0	1019666

5.5.3. Chunking and reruns

It is not necessary that all `data_types` attributes are contained in a single file. For instance, the results of the query to be saved under filename `bloomberg__ticker_fundamental_data__ZAE000018131 Equity.csv` could be chunked across multiple queries and spread across the files

```
152874575747280__bloomberg__ticker_fundamental_data__ZAE000018131 Equity.csv
152874575747380__bloomberg__ticker_fundamental_data__ZAE000018131 Equity.csv
152874575747480__bloomberg__ticker_fundamental_data__ZAE000018131 Equity.csv
152874575747580__bloomberg__ticker_fundamental_data__ZAE000018131 Equity.csv
```

This chunking can be done along either rows or columns or both. Chunking does not have to be sequential, either. A query can be run multiple times, saving more and more data to the datalog.

`1_simulated_to_datalog.R` reads in parameters from the `parameters.R` file and generates a datalog that confirms to the above specification. The ticker names are randomly generated, but reproducible through the specification of a seed. This datalog can be consumed by downstream scripts in the same manner as queried data.

5.6. Data processing - `2_process_data.R`

The `data/dataset` directory is the canonical data store of the system. Unstructured data in the datalog is compacted and organised according to a predictable directory structure inside the dataset directory.

`Relevant parts of the directory tree:`

```
. equity_analysis
|    data/
|    |   datalog/
|    |   datasets/
|    |   |  constituent_list/
|    |   |  |  metadata_array.feather
|    |   |  metadata_array/
|    |   |  |  metadata_array.feather
|    |   |  ticker_fundamental_data/
|    |   |  |  ISIN_1.feather
|    |   |  |  ISIN_n.feather
|    |   |  ticker_market_data/
|    |   |  |  ticker_1.feather
|    |   |  |  ticker_n.feather
|    scripts/
|    |   2_process_data.R
|    |   data_processing/
|    |   |   2_datalog_csv_to_feather.R
|    |   |   3_constituents_to_dataset.R
|    |   |   3_metadata_to_dataset.R
|    |   |   3_ticker_logs_to_dataset.R
```

In order to speed up downstream processing time, all `.csv` files in the `datalog` are converted to `.feather` format. `.feather` files are much faster to read and write, and do not require the type parsing typically required of `.csv` files.

The objective of the scripts in `1_process_data.R` is to read in the contents of the `datalog`, identify what datasets can be generated from the logs, and commit them to the `dataset` archive. Suitable target datasets are, for instance, `metadata/metadata.feather` or `ticker_fundamental_data/ISIN_1.feather`.

5.6.1. Melting

In contrast to datalogs, datasets are stored in a 'tall and narrow' data structure.

The following table is 'short and wide', with one row per date, and one column per attribute (we omit columns due to limited space. There are also timestamp and source columns in this table) -

date	BS_ACCT_NOTE_RCV	BS_INVENTORIES
2014-12-31	NA	1182822
2015-06-30	782550.2	1158028
2015-12-31	NA	1260626
2016-06-30	605207.4	1048830
2016-12-31	NA	1057549
2017-06-30	522236.0	1019666

The same table can be converted into 'tall and narrow' format using the `tidyr:gather()` function into this format -

date	timestamp	source	metric	value
2014-12-31	152874575747280	bloomberg	BS_ACCT_NOTE_RCV	NA
2015-06-30	152874575747280	bloomberg	BS_ACCT_NOTE_RCV	782550.2
2015-12-31	152874575747280	bloomberg	BS_ACCT_NOTE_RCV	NA
2016-06-30	152874575747280	bloomberg	BS_ACCT_NOTE_RCV	605207.4
2016-12-31	152874575747280	bloomberg	BS_ACCT_NOTE_RCV	NA
2017-06-30	152874575747280	bloomberg	BS_ACCT_NOTE_RCV	522236.0
2017-12-31	152874575747280	bloomberg	BS_ACCT_NOTE_RCV	1066796.2
2014-12-31	152874575747280	bloomberg	BS_INVENTORIES	1182822.4
2015-06-30	152874575747280	bloomberg	BS_INVENTORIES	1158028.4
2015-12-31	152874575747280	bloomberg	BS_INVENTORIES	1260625.5

We do this conversion for a number of reasons. Firstly, it allows us to filter for a particular data point (i.e, a date, attribute, and ticker cell) much more quickly by simply filtering each column. This enables efficient deduplication, compaction and versioning, which reduces the dataset size dramatically and speeds load, write and processing operations.

Secondly, it allows us great flexibility in the attributes we collect. Collecting variable numbers of attributes in the querying step results in tables with varying columns. Joining multiple files to create one dataset on 'short and wide' data would require column matching on every single join operation,

which is expensive and error-prone. In contrast, columns are stable when 'tall and narrow'. No matter what the original attributes are, or how they change across queries, we know that we will always have the same columns, namely `date`, `timestamp`, `source`, `metric` and `value`.

Finally, the 'tall and narrow' format allows us to deal with NA values properly. With this format, we know that if the `value` column is `NA`, then we should drop the row. Consider the 'short and wide' example. There are multiple attributes. Some are `NA`, others are not. How do we deal with this? There is no easy way to drop a single attribute for a single date without dropping the other attributes for that date. This is not a big issue for single-query datalogs. But consider what happens when we have multiple versions of a data point, some of which are `NA`, and some of which are not. Finding the latest value of a data point that is not `NA` and using that as your latest value is quite complex with 'short and wide' data. For 'tall and narrow' data, it's quite simple. Just filter the rows appropriately, drop any `NA` entries and then take the most recent timestamp. Dropping of `NA` values is done without loss of information, since recasting into 'short and narrow' format inserts `NA` values into empty cells.

'Tall and narrow' formats also have disadvantages, especially when one wants to perform row-wise computations across attributes. Because of this, we convert the datasets back to 'short and wide' format during backtesting.

5.6.2. Using dataframes rather than timeseries objects

R supports several timeseries data structures, including `ts`, `zoo` and `xts` formats. These formats come with methods that are very useful for the manipulation of time series. We have decided to use dataframes instead of these formats because they require that all attributes be the same data type. This is a consequence of the fact that the `zoo` package is actually an indexed matrix; matrices in R can only be one data type. Sticking with dataframes allows us to mix many types of data in our timeseries.

One exception to this is the reporting done after the backtests have been completed. The packages `pbo` (for backtest overfitting detection) and `dygraphs` expect timeseries objects; for these we convert daily returns to time series.

5.6.3. Script procedures

The series of scripts sourced in `1_process_data.R` each perform the following procedures -

1) Read all files in the `datalog` into a dataframe, splitting each fieldname into an appropriate column.
2) From the file names, determine the set of datasets that can be generated from the logfiles.
3) For each dataset:

1) Read in logfiles relevant to the dataset
2) Convert to 'tall and narrow' format
3) Check if a dataset already exists and, if it does, read it in to memory
4) Join the datasets
5) Filter out any NA values
6) Remove any duplicates
7) Filter out any records that have not changed since the last timestamp, but update the timestamp field

The end result is a set of directories in `data/datasets`. Because the scripts merge, rather than overwrite datset files, it is possible (but not necessary) to periodically empty or delete the datalog without loss of information.

The files in the `datasets` directories are **not** tidy. In particular -

1) The constituent and metadata files contain all data for all indexes and tickers in a single file. This contradicts the tidy guideline of separating each 'thing' (ie ticker, or date) into a separate table.
2) Each attribute is not a column.

The datasets are converted to tidy format in-memory at backtest runtime.

5.7. Loading Data - `3_load_data.R`

The purpose of this series of scripts is to load at-rest data from datasets and save them in a backtest-ready form in a fast file format. These binary files are considered temporary and saved in the `temp` folder; deleting them does not impact reproducability.

`Relevant parts of the directory tree:`

```
. equity_analysis/
|    data/
|    |   datasets/
|    scripts/
|    |   parameters.R
|    |   3_load_data.R
|    |   data_loading/
|    |   |   compile_data_report.R
|    |   |   load_slow_moving_data.R
|    |   reporting/
|    |   |   data_quality.Rmd
|    temp/
|    |   ticker_data.Rds
|    |   price_data.Rds
|    |   constituent_list.Rds
|    |   metadata.Rds
|    |   data_report.Rdata
```

5.7.1. Filtering

The `load_slow_moving_data.R` script reads in parameters from the `parameters.R` file and then reads in the necessary files from the `datasets/` directory. It selects only the relevant constituent index, joins the necessary ticker market and fundamental data, drops unnecessary fields, and saves the results to a a `ticker_data.Rds` file. This file is then read at every rebalancing period by the backtest algorithm.

5.7.2. Quote Engine

The trading algorithm will run a `get_quote` function to get price quotes for each trade at each heartbeat. This function will read in prices from a `price_data` object. This `price_data` object is derived from the `ticker_data` object above, but saved to a separate file. This is because the

`price_data` has several changes that may introduce backtest bias -

- It it not date-filtered at runtime, so contains look-ahead information
- A high and low price are generated synthetically from available data
- `NA` values are backfilled (for price data) or changed to `0` (for volume data)

5.7.3. Lag-detection

The script also reads from `parameters.R` whether a lag must be imposed on fundamental data. This is necessary when fundamental dates are based on official financial reporting dates, not actual dates of release (which can be much later). Sometimes accompanying financial data is of a hybrid variety, where some metrics need to be lagged (typically financial statement data) while others (typically ratios) do not. For these datasets, the user has the option of enabling k-means cluster detection of the two populations of metrics. The k-means algorithm tries to cluster the fundamental metrics according to the number of occurrences of each metric in the population, the rationale being that core financial data is updated far less frequently than financial ratios with a market component. Once the clusters have been computed, the cluster of metrics with the lowest number of occurrences have the dates of their occurrences lag-adjusted, while the other cluster(s) are left unmodified.

5.7.4. Data quality reporting

Data quality is critical to running a realistic backtest. The `compile_data_report.R` script computes a range of health measures for the entire dataset, including, but not limited to -

- Proportion of `NA` in selected market metrics.
- Histogram of `NA` occurrences per ticker, per metric type
- Results of k-means detection (if selected), along with verification that the lag has been imposed.

If selected in `0_all.R`, these measures are fed to a `data_quality.Rmd` file, which generates an html report in the `results` directory that can be used to inspect the data.

A typical workflow using these scripts is to run the backtest, while including all fundamental and market metrics, up until the data quality report is generatred, and then cancelling it. Inspection of the report guides the researcher as to which metrics may be useful in constructing a backtest, and whether there is sufficient data for a realistic backtest. The trading algorithm will drop any tickers that do not have sufficient data at runtime.

5.8. Backtesting - `4_run_trials.R`

Unlike the data-processing scripts, the backtesting scripts are not named in a sequential fashion. This is because their sourcing is not sequential.

```
Relevant parts of the directory tree:

. equity_analysis/
|    data/
|    |   datasets/
|    scripts/
|    |   parameters.R
|    |   4_run_trials.R
|    |   trading/
|    |   |   sample_trials/
|    |   |   trade.R
|    temp/
|    |   ticker_data.Rds
|    |   price_data.Rds
|    |   constituent_list.Rds
|    |   metadata.Rds
|    trials/
|    results/
```

Rather, these scripts follow a parent-child relationship. The parent script is `4_run_trials.R`. It sources the other scripts and uses their functions when necessary. Understanding how backtesting is accomplished is best achieved by working through this script from top to bottom, referring to other scripts when they are called.

5.8.1. Global parameters

The `parameters.R` file sets parameters that are global to the trading session. The parameters in the trading section are -

1) Index-specific

- `constituent_index` - the index. `"JALSH"`
- `data_source` - the data source. `"bloomberg"`

- `price_related_data` - a list of all metrics that are price related. `c("date", "PX_OPEN", "PX_HIGH", "PX_LOW", "PX_LAST")`
- `last_price_field` - a list of the metrics used to value the portfolio holdings. `c("date", "TOT_RETURN_INDEX_GROSS_DVDS")`
- `volume_data` - the volume metric. `c("date", "VOLUME")`
- `market_metrics` - a list of all market metrics required for the algorithm to work. `c("CUR_MKT_CAP")`
- `fundamental_metrics` - a list of all fundamental metrics required for the algorithm to work. `c("BS_SH_OUT", "DIVIDEND_YIELD")`

2) Lag-adjustment

- `fundamental_data_lag_adjustment` - how much to lag data, in days. 0 will skip adjustment.
- `fundamental_data_metric_types` - either 1, 2 or auto. 1 will lag all fundamental metrics.

3) General trading parameters

- `run_mode` - either LIVE or BACKTEST (live not implemented).
- `heartbeat_duration` how often to trade, in seconds.
- `rebalancing_periodicity` how often to rebalance, in seconds.

4) Timeframe

- `start_backtest` - YYYYMMDD inclusive
- `end_backtest` - YYYYMMDD not inclusive

5) Portfolio Characteristics

- `portfolio_starting_config` - CASH or STOCK. CASH initialises the portfolio with 100% cash. STOCK initialises the portfolio with perfect portfolio weights at start. STOCK not implemented yet.
- `portfolio_starting_value` - starting cash value of portfolio.
- `cash_buffer_percentage` - desired cash percentage of portfolio to target.

6) Trading characteristics

- `commission_rate` - percentage of trade value charged as commission.
- `minimum_commission` - minimum value of commission.
- `standard_spread` - simple assumed spread between bid and ask, expressed as a percentage of stock price.
- `soft_rebalancing_constraint` - cancel trade if the holding is within a certain percentage of the ideal weight. This is to mitigate churn from chasing an unnecessary degree of precision.

We see from these parameters that the backtester uses a stock index as the defining unit of a stock universe. It can only use one data source at a time, but synthetic datasets can be built into the data pipeline stage. It expects a start and end date for the backtest, and we have to specify a portfolio starting value and desired cash buffer. This is because excess returns are portfolio-size dependent. Large portfolios may not be able to trade illiquid stocks; small portfolios may suffer from high commissions. We have also specified rudimentary transaction-cost modelling and slippage, in the form of a simple commission structure and a standard spread which will be imposed at trade.

The script runs `trade.R` against each `algorithm_X.R` file in the `trials/` directory, saving the results, and the `algorithm_X.R` file, to its own subdirectory in the `results/` directory. The subdirectory naming convention is `INDEX__algorithmX`.

That is, an algorthm named `market_weighted.R`, run against the index `JALSH`, will result in a results subdirectory `JALSH__market_weighted`. This subdirectory will contain the original `market_weighted.R` file as well as several reporting and runtime history files. It is therefore possible to take a `results` subdirectory contents, rerun the algorithm, and compare it with the original results.

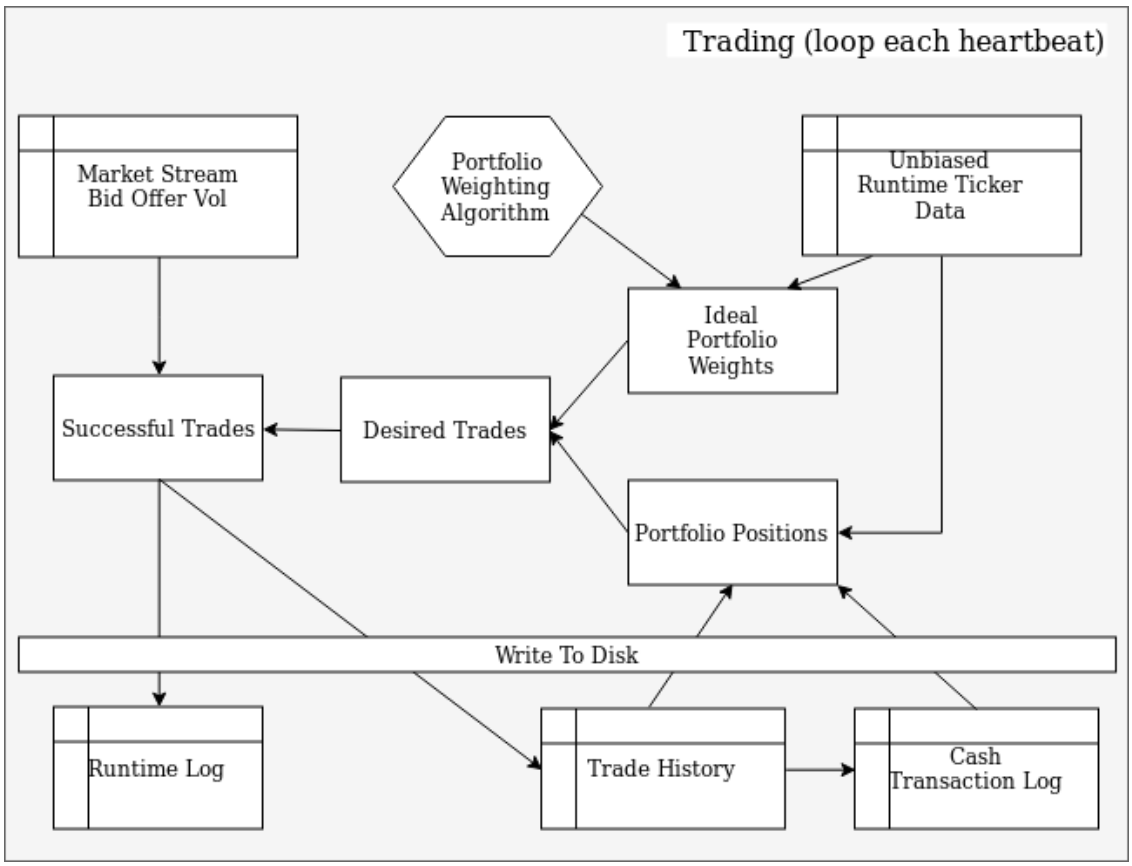

Figure 5.2: Trading Workflow Diagram

5.8.2. The `ticker_data` and `runtime_ticker_data` objects

The trading engine reuses several dataframes during the course of operations. Two of these are critical to understanding how it avoids look-ahead bias and generates trades.

1) The `ticker_data` data object is a list of dataframes containing tickers relevant to the backtest. It is **not** free of survivorship bias or look ahead bias. It is constructed by loading `ticker_data.Rds`.
    1) It reads in the `constituent_list.Rds` and `metadata.Rds` datasets, and filters these datasets to include only `source` and `index` parameters included in the global environment.
    2) It then constructs a list of dataframes, where each dataframe contains the market and fundamental data of a ticker in the filtered constituent list, joined on the metadata market and fundamental identifier fields.
    3) It filters all dataframes to take only the most recent timestamp version of data points with multiple timestamped versions.
    4) It spreads the dataframes into 'short and wide', tidy form.
2) The `runtime_ticker_data` data object is derived from `ticker_data` and **is** free of survivorship and look-ahead bias. It is recomputed every time the `runtime_date` (i.e the date that the trading engine thinks it is) changes, and is computed by calling the `get_runtime_dataset` function. This function takes `execution_date`, `constituent_list` and `ticker_data` objects as arguments, and returns a subset of `ticker_data` that contains only those constituents that were in the constituent list at the execution date, and only those data points that were available prior to the execution date. It also backfills any `NA` values with the last-known value. This converts low-periodicity data, such as balance sheet line items, into daily data for easy row-wise operations.

5.8.3. The `compute_weights` function

This function comprises the entire contents of a `trials/algorithm_X.R` script. This function is designed by the researcher, and must take in an in-memory `runtime_ticker_data` dataset and a vector of `metrics`(ticker attributes), and returns a list of ideal portfolio weights. That is, it reads in the state of the world as it would appear at `runtime_date` and makes some decision about what the optimal portfolio weighting should be. This decision rule can be whatever the researcher wants, and can make full use of the R universe of statistical packages in construction.

The power of this arrangement is that it allows a researcher to modify the weighting algorithm while holding all other variables constant. Because it accepts a whole `runtime_ticker_data` object, it can conduct `map` operations across the entire list for dataframes - it can specify the target weight of a ticker contingent on, say, its ranking amongst all other tickers based on some ranking procedure that accepts arbitrary attributes. This is immensely flexible - one can build `map` operations that set all

non-industrial stocks to weight 0, for instance, or nest rankings according to sector and then hard-code overall sector weights.

Because the wrapper script `trade.R` only cares about getting back a vector with portfolio weights, the internals of the function can be very elaborate. It would be entirely possible to spin up an `h2o` machine learning instance inside this function, train a deep learning model, predict an outcome and use that classification to return `target_weights`. Because we know that the function only has access to bias-free data, we know the predictions won't be able to cheat. Since the dataframes are in tidy format, they are accepted without modification by many R model-fitting packages.

Since these weights only need to be recomputed daily, this training can take quite long. Higher-frequency prediction would constrain model complexity to occur within the heartbeat window.

5.8.4. Trade submission procedure

The `trade.R` script performs the following operations -

1) Source the `algorithm.R` script.
2) Set `heartbeat_count` to 0.
3) Set `runtime_date` to start of backtest + `heartbeat_count`.
4) Check if `ticker_data` has been loaded to memory and if it is less than 24 hours (in heartbeat time) old.
    - If not, reload `ticker_data`.
5) Check if `runtime_ticker_data` is stale.
    - If it is stale, run `get_runtime_dataset` again and then run `compute_weights` again to obtain new `target_weights`. We only re-compute `target_weights` when `runtime_ticker_data` changes because it is the only input to the `compute_weights` function. **Drop any tickers that do not have sufficient data to compute a target weight**. The `market_metrics` and `fundamental_metrics` parameters are the metrics that are checked for eliminating tickers. **If the algorithm uses a metric not specified in these parameters, then a ticker that is included but has insufficient information will cause a halt to the backtest.**
6) Get the `transaction_log` and `trade_history` data objects and compute our current portfolio and cash positions.
7) Query latest prices for the positions to obtain a portfolio valuation.
8) Compute `trades` by calling `compute_trades`, using `target_weights` and `positions` as arguments.
9) Submit `trades` by calling the `submit_orders` function, using `trades` as an argument.

10) Increment the `heartbeat_count` value by the global `heartbeat_duration` value.
11) Repeat steps 3 - 10 until `runtime_date` is equal to or greater than `end_backtest`.

In a live environment, the `submit_trades()` function would submit the trades to a broker via the broker-supplied API. Submitted trades would be matched (or partially matched) by an external broker and our broker-side trade history and transaction logs would be updated. In this setup, getting `transaction_log` and `trade_history` objects would amount to querying the broker API and saving the result to the `results/INDEX__algorithmX` subdirectory. These logs are available to the trade loop and perform the function of updating the portfolio weights, allowing a new `trades` object to be computed.

In a backtest environment, this trade matching needs to be simulated. In this context, the `submit_orders()` function simulates the actions of an external broker. This function accepts a `trades` dataframe as an argument, and then -

1) Obtains price quotes for each ticker in the `trades` dataframe. The `get_stock_quote` function queries ticker_data for the highest price, lowest price and volume on the execution date, and then selects a random sample from the price range. This random sample is the `midpoint`. The `bid` and `offer` are computed as the difference of `midpoint` and the `spread`, as defined in the global parameters. It then computes the `size` of available units as daily volume divided by (trading day / heartbeat duration). That is, it assigns trading lots uniformly throughout the day. In this manner, liquidity is constrained to a reasonable approximation of the volume that would have been available throughout the day. Finally, it returns a vector with the `bid`, `ask` and `size`.
2) Computes successful trades as those where the correct side (i.e bid for a sell order, ask for a buy order) is within the limit order price specified in the `trades` dataframe.
3) Assigns a trade size (the minimum of units available and units asked for) to each trade.
4) Generates a session trade log and transaction history.
5) Reads in the `trade_history` and `transaction_log` objects.
6) Appends the session values.
7) Saves the `trade_history` and `transaction_log` objects to file.
8) Returns a short summary of successful trades.

5.8.5. Backtest results

A well-specified `compute_weights` function will result in a successful backtest. The results of each backtest run by `4_run_trials.R` will be placed in the `results` directory according to the naming convention covered earlier.

On completion of `4_run_trials.R`, the `results` subdirectories contain four files -

1) The original `algorithm_X.R` script from the `trials` directory.
2) A `runtime_log` file, which contains relevant runtime information. Each line of this file represents another heartbeat.
3) A `trade_history.feather` file, which contains a full trade history, similar to what would be obtained from an external broker.
4) A `transaction_log.feather` file, which contains a full transaction log, similar to what would be obtained from an external broker. This log is analogous to a bank account, and keeps track of account cash balances as trades, deposits, withdrawals and account charges are levied.

In addition, the `parameters.R` file, which contains important information related to the environment, is copied into the root of the `results` directory.

The `results` subdirectories contain all the necessary data to compute portfolio performance characteristics.

```
Relevant parts of the directory tree:

. equity_analysis/
|    R/
|    |   |   set_paths.R
|    scripts/
|    |   parameters.R
|    |   5_report.R
|    |   reporting/
|    |   |   tearsheet.Rmd
|    results/
```

There exist several portfolio performance reporting packages on CRAN; it is beyond the scope of this exercise to replicate the functionality of those packages. The data created by the backtest trading algorithm are designed to support general customisability of the results tear sheet.

The `5_report.R` script has three objectives -

1) Create a matrix of daily returns for piping into a downstream portfolio analysis package.
2) Generate a minimal performance tear sheet in order to demonstrate that portfolio analysis can, in principle, be conducted on the folder contents.
3) Collect the daily returns of all of the trials into a single matrix for downstream analysis for overfitting.

To achieve these aims, the script performs the following steps on each result subdirectory -

1) Load the transaction log, trade history and runtime log.
2) Determine the start and end date of the backtest from the runtime log.
3) Compute cash balances, portfolio positions, and stock prices for each date in the backtest.
4) Compute the total portfolio value for each date in the backtest.
5) Combine the cash balance, total stock value, and total portfolio value into a tidy dataframe.
6) Compute miscellaneous rolling performance statistics, such as daily return, excess return and rolling average return and add them as columns to the dataframe.
7) Save the dataframe to a `portfolio_stats.feather` file in the subdirectory.

8) Copy in the `tearsheet.Rmd` template from `scripts/reporting/`.

9) Parametrically render the `tearsheet.Rmd` into a `tearsheet.html` web page that can be read and distributed.

As the script works through each subdirectory, it compiles cross-sectional dataframes that contain all the trials' daily and total returns. That is, dataframes with a row for every date, and a column for the daily returns or total returns for each trial result. These dataframes are saved to `results/daily_returns.feather` and `results/total_returns.feather`, where it can be consumed by the `6_cross_validate.R` cross-validation script.

Backtest Results Report

December 16, 2018

Parameters

The backtest is specified by the following parameters:

Parameter	Value
Data Source	simulated
Target Equity Index	RISK_FREE_GROWERS
Number of Stocks Traded	50
Algorithm Name	random_trial_13
Start of Backtest	2018-12-01
End of Backtest	2008-01-01
Backtest Duration (months)	131

Trade Characteristics

Characteristic	Value
Number of Trades	194390
Total Pre-Commission Value of Trades	2631.94
Total Commission Paid	0
Total Commission % of Trade Value	0
Trade Value / Final Portfolio Value	0.31
Max Drawdown (%)	0.23
Final Value of Portfolio	8403.71

Figure 5.3: Tearsheet Excerpt

It must be emphasised that the trial-specific `portfolio_stats.feather` tables and the tearsheets are minimal examples, and they should only be used for serious analysis after heavy augmentation. We decided not to flesh out complete statistical summaries because there are other R packages that can do that (see Peterson and Carl (2018) as an example). Our objective here is to lay a foundation that can be quite easily extended to serve the needs of a researcher.

The parametrised template tearsheet resides at `scripts/reporting/tearsheet.Rmd`. It can be modified in exactly the same manner as any other Rmarkdown script.

5.10. Cross validation - `6_cross_validate.R`

Relevant parts of the directory tree:

```
. equity_analysis/
|    R/
|    |   |   set_paths.R
|    scripts/
|    |   parameters.R
|    |   6_cross_validate.R
|    |   reporting/
|    |   |   cross_validate.Rmd
|    results/
|    |   daily_returns.feather
|    |   total_returns.feather
```

In order to investigate whether backtest overfitting has occurred, we make use of the `pbo` package, available in CRAN. The `pbo` package expects a matrix or `xts` object of daily returns for a range of trials; it uses this array to compute the various metrics in Lopez de Prado (2013).

The computations done in this section are the final part of the analysis; the output is intended to be read by humans. `6_cross_validate.R` is just a wrapper that knits an Rmarkdown notebook, `results/cross_validate.Rmd`. The end result is an html report that includes details on the probability of backtest overfitting as it relates to the array.

The template `Rmd` file can be modified at `scripts/reporting/cross_validate.Rmd`

Probability of Backtest Overfitting

We use the methods described in the (Probability of Backtest Overfitting) [https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2326253] to detect overfitting.

To compute the probability of backtest overfitting, we need to define a performance metric. We can use any performance metric we want. We use the same performance metric here as was used in the orignal paper - the Sharpe Ratio. This ratio is defined in the `pbo` package vignette as -

```
sharpe <- function(x,rf=daily_risk_free_rate) {
  sr <- apply(x,2,function(col) {
    er = col - rf
    return(mean(er)/sd(er))
  })
  return(sr)
}
```

Results

```
## Performance function sharpe with threshold 0
```

	Value	Range	Desirable
PBO	0.8142857	0->1	1
Slope	0.0316130	-inf->inf	NA
Adjusted R-squared	0.5000000	0->1	1
Probability of Loss	0.9290000	0->1	0

Histogram

This is a histogram of the logits. A negative value indicate a best in-sample trial that performed below the median trial out of sample; a positive value indicates a best in-sample trial that performed better than the median trial out of sample.

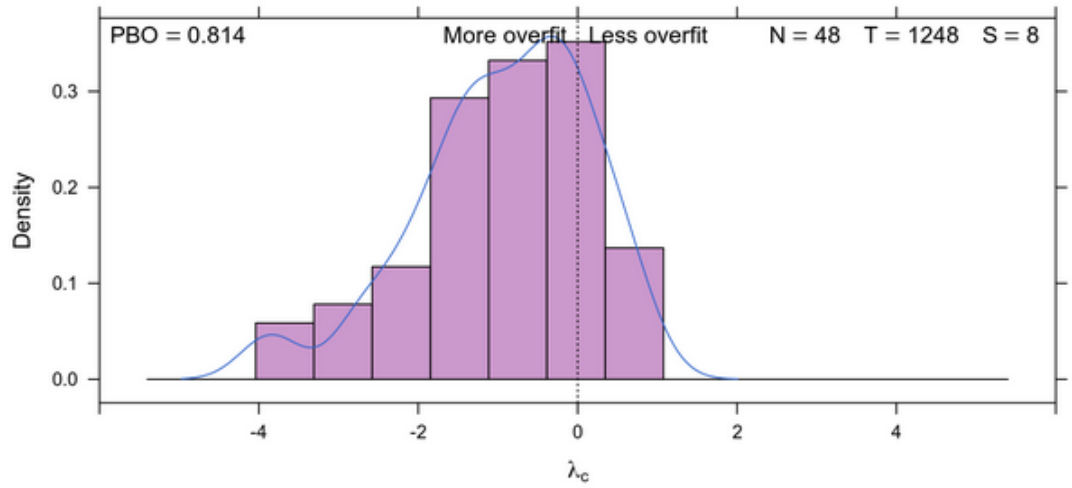

Figure 5.4: Cross Validation Report Excerpt

5.11. Application and Examples

This section details the use of the environment to perform a variety of demonstrations and tests. All of these tests were conducted on the `lag-correction` branch of https://github.com/riazarbi between the 1st and 5th February 2019. This branch will be merged into `master` on the 11th February 2019. This code was used largely unmodified for each test; the only differences between each test are those contained in the `parameters.R` file and the specific `trials` files. Unmodified copies of these files can be found at https://github.com/riazarbi/equity_analysis_trials.

Each test followed the same protocol -

1. Create a `parameters.R` file and trial script (or several, if necessary)
2. Create a `deploy.R` script
3. Zip each backtest
4. `docker run ... riazarbi/equity_analysis` a new backtest environment. This environemt already contains the code from the GitHub repository mentioned above.
5. Navigate to the correct url for the backtest environment
6. Open the `equity_backtest` R project.
7. Upload the relevant zip file. It is unzipped automatically by RStudio Server.
8. Type `source("<unzippedfolder>/deploy.R")` and press `<ENTER>`.
9. Follow prompts, entering appropriate answers for the particular backtest, and then wait for the backtest to complete.
10. Download results and compile results.

In general, a backtest takes about 90 minutes per algortihm in the trials directory. That is, a 50-trial backtest will take up to 75 hours, if rebalanced daily and traded multiple times per day. These can be run in parallel, however, if one varies parameters to investigate outcomes in a range of trading conditions.

5.11.1. Querying and assessing real data

The first step in constructing a backtest is data collection. Once the data has been collected, it needs to be assessed for quality. Typical questions to be asked include -

- How likely is a researcher to obtain an accurate or representative result, given the quality of the data?
- If an algorithm presumes that all data is available for each metric and for each ticker, what is the impact of a dataset that is missing entire ticker datasets on the accuracy of the result?

- What about data that contains all tickers, but is missing certain metrics?

These sorts of issues are case-specific, so we provide an automated report that provides the researcher with measures of dataset health. The expectation is that the researcher will reflect on these measures and decide on an appropriate course of action. Appropriate courses of action could be -

- Collect more data
- Change the backtest window
- Alter the algorithm to rely on data you have
- Use a subset of the index

These decisions can impact the final backtest outcome, but since the algorithm, parameters and data processing scripts are all versioned these changes are transparent to a replicator.

The following paragraphs walk through a typical data quality assessment. They assume that the user has an already-initialized environment, as detailed in 5.11. The data source in this example is Bloomberg, and the query method is via the `Rblpapi` package.

A copy of the `equity_analysis` code package must be loaded on a Bloomberg terminal. If the user does not have rights to install R on the terminal, a portable version of R and RStudio for Windows can be installed on to removable-storage media. The R installation must have the necessary packages (as detailed in `0_all.R`) pre-installed, since installing packages on-the-fly requires administrator rights.

Once RStudio has been loaded on the terminal and the `equity_analyis` R project has been loaded, the user needs to log in to the Bloomberg user interface using provided credentials. This should start the `bbcomm.exe` background process, which is what R uses to communicate with Bloomberg. One can verify that `bbcomm.exe` is running by inspecting the Task Manager process panel.

Query parameters are set directly in the `1_bloomberg_to_datalog.R` script. The user should set the stock index to query, the date range to query constituent indexes for, the currency to be returned, and the list of market and fundamental metrics to query. These metrics are separated because market data is an intersection of an index and a stock, while fundamental data is a property of the stock alone. Therefore market data will vary from exchange to exchange, but fundamental data will not. Fundamental data is always associated with the primary exchange ticker code, not the secondary exchange code, and the ISIN. We rely on ISIN codes for stability as these are unchanging over time.

Our example queries the JSE All Share Index and Top 40 Indexes for a date range of 240 months, beginning on the 27th of January 2019. We query 15 metadata fields, 26 market data fields and 79

fundamental fields. The fields specified can be inspected in the code repository of this project. The script queries all data relating to those metrics for the period 1 January 1960 to 27 January 2019.

Once these query parameters have been set, we enter `source("scripts/1_bloomberg_to_datalog.R")` into the R console and press `<ENTER>`. The script takes approximately 30 minutes to run and saves over 14000 files to the datalog directory. Once complete, we transfer the removable media to a permanent workstation and transfer across the results.

Here, we simply open up the `equity_analysis` R project and execute `source("scripts/0_all.R")`. We do not clear the dataset directory or trials directory, but we do run the data quality check. The `parameters.R` settings are default, except that `market_metrics` and `fundamental_metrics` are set to empty vectors to ensure no filtering is done, the fundamental lag is set to 120 days and the fundamental metrics clusters parameters is set to `auto`.

Once the script has passed the data quality report generation step, it can be terminated. One convenient way to terminate the script prematurely is to leave the `trials/` directory empty; this results in speedy conclusion of the script.

The data quality `html` report can be obtained from the `results/` directory. It is accompanied by a sidecar `.Rmd` file, which can be edited and re-run for exploratory analysis. The report contains some datalog summary statistics which validate that all the information in the datalogs are being transferred to the at-rest datasets. It also contains a market data field completeness chart, which provides a visual representation of the proportion of each market metric that contain NA values across the dataset.

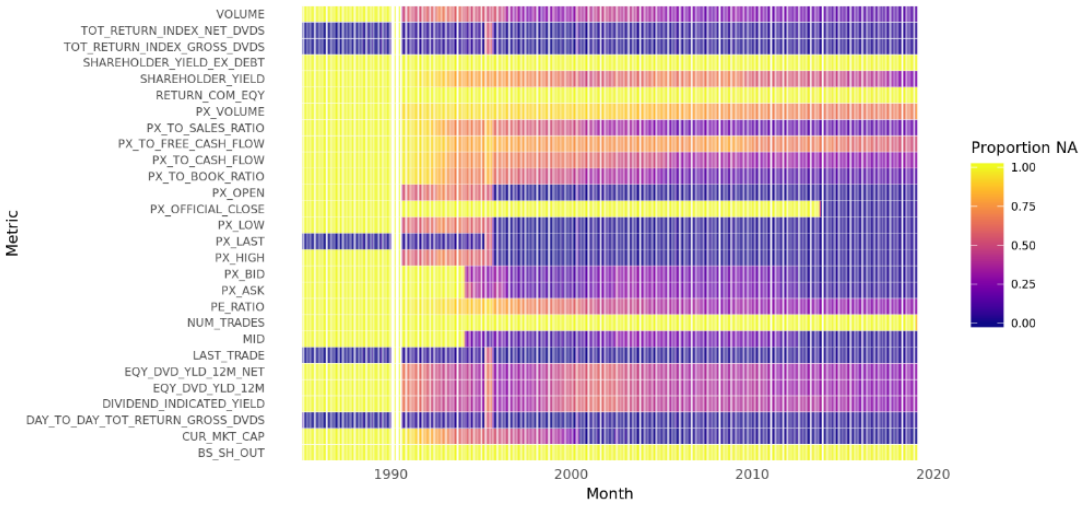

Figure 5.5: Market Metric Field Completeness, Full Dataset

This chart shows that some metrics are more complete than others. In particular, total return with gross dividends and market capitalization are largely complete from the year 2004 onwards. These metrics may be good candidates for portfolio performance scoring and the algorithm weighting, respectively.

The report also shows the overall monthly market data `NA` percentages over time. `NA` values are problematic because they result in constituents being dropped from the dataset. Ideally a user will select the subset of metrics and the date range that minimizes this error while testing useful hypotheses.

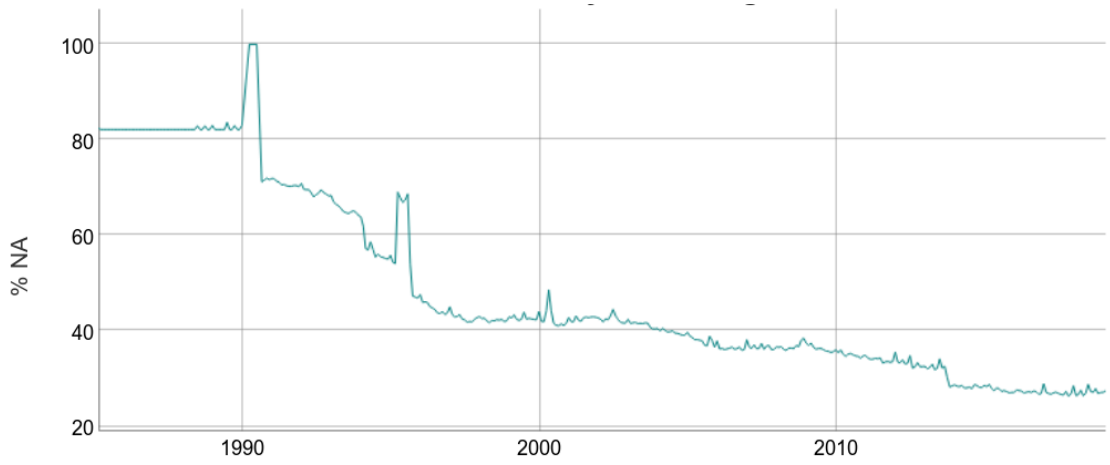

Figure 5.6: Monthly Overall Market NA Over Time

The report also checks whether there are any tickers which do not have metadata, market data and fundamental data. These tickers will be excluded from the backtest.

Fundamental data cannot be submitted to the same `NA` criterion as above becuse it is slow-moving, and hence has many `NA` values on a daily time scale. The backtester backfills fundamental data into daily values on the assumption that, if a stock has one fundamental data entry, it probably has a complete relevant set. Stocks with no fundamental data for a particular field have that field dropped, and, if they are required for the backtest, the ticker is excluded from the set. These exclusions are recorded in the console output at runtime. However since data is released on different schedules for different stocks, it is difficult to impute the best periodicity and aligment for assessing `NA` values.

Another issue with fundamental data is lag adjustments. The amount of lag to be applied to fundamental data is specified in the `parameters.R` file. Some fundamental datasets contain a mix of daily and low-frequency data; the former does not need adjustment. To address this, the data loading script will auto-detect the number of clusters in the dataset, and, if there are two or mre, apply the lag to only the least-frequently-updated set of metrics. Our dataset does indeed have two clusters, as shown by the k-means silhouette below. The metrics contained in cluster 2 in the below cluster plot have their dates lag-adjusted by 120 days.

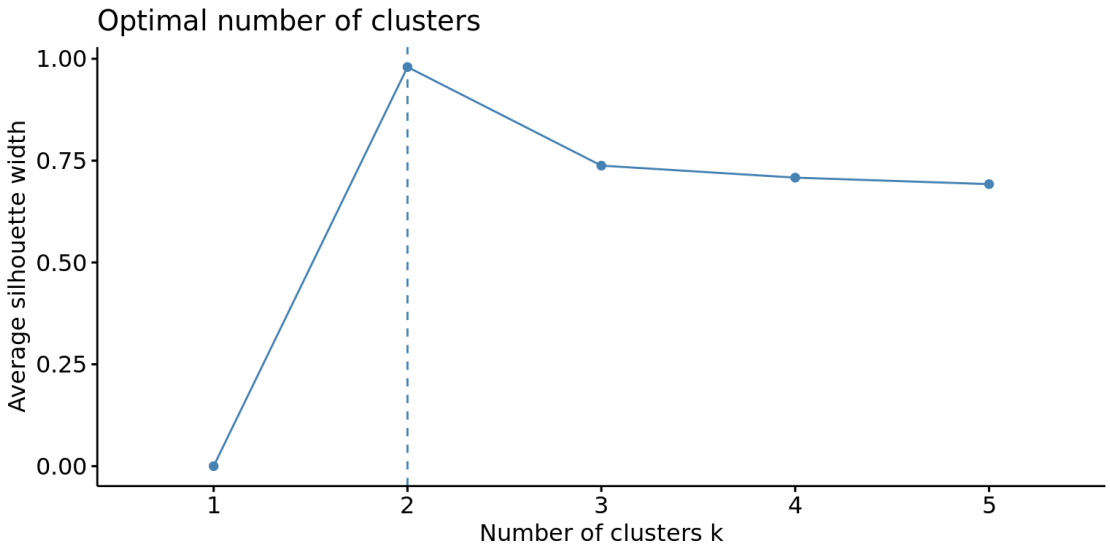

Figure 5.7: K-means Fundamental Metric Counts Silhouette Plot

The data health report also show histograms of the `NA` proportions per market metric group per ticker. The groups are overall market fields, algorithm-related fields, price-related fields, volume, and last price. This allows the user to discern the distribution of `NA` values across the constituents. One wants as many of the tickers as possible to have as low `NA` counts as possible, because this means less tickers will be dropped from the dataset at runtime.

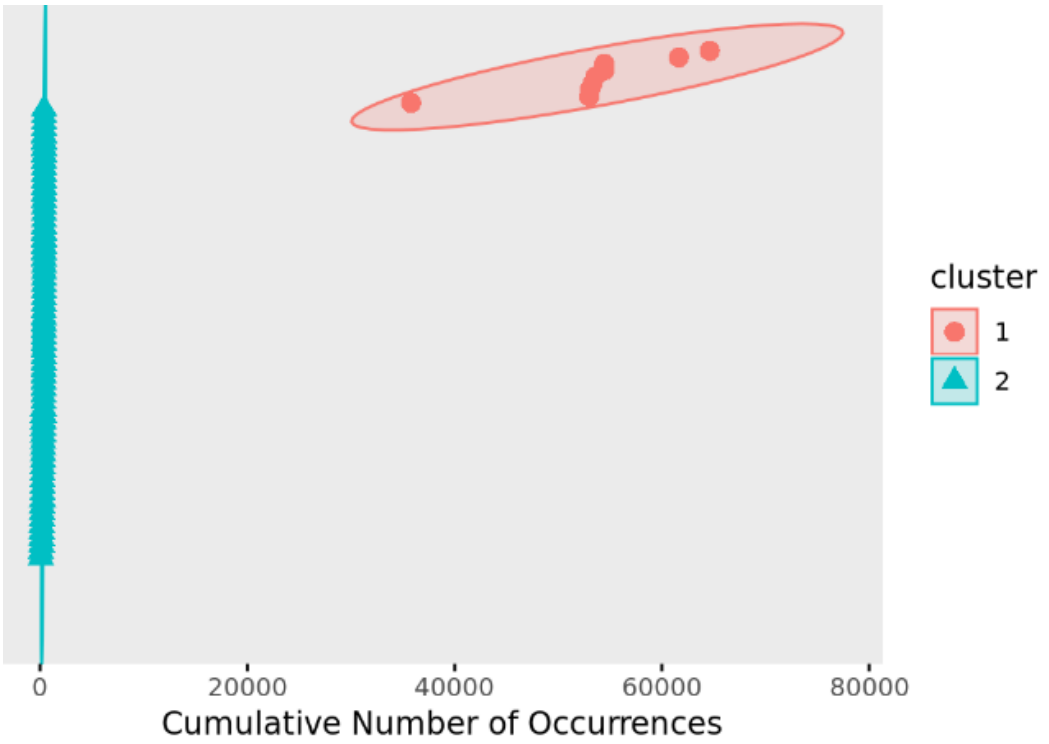

Figure 5.8: Fundamental Metric Clusters

Having reviewed the data health report above, we can filter the dataset to exclude poorly represented metrics. The following market metric completeness chart shows a dataset that excludes all market metrics except price, volume and market cap metrics.

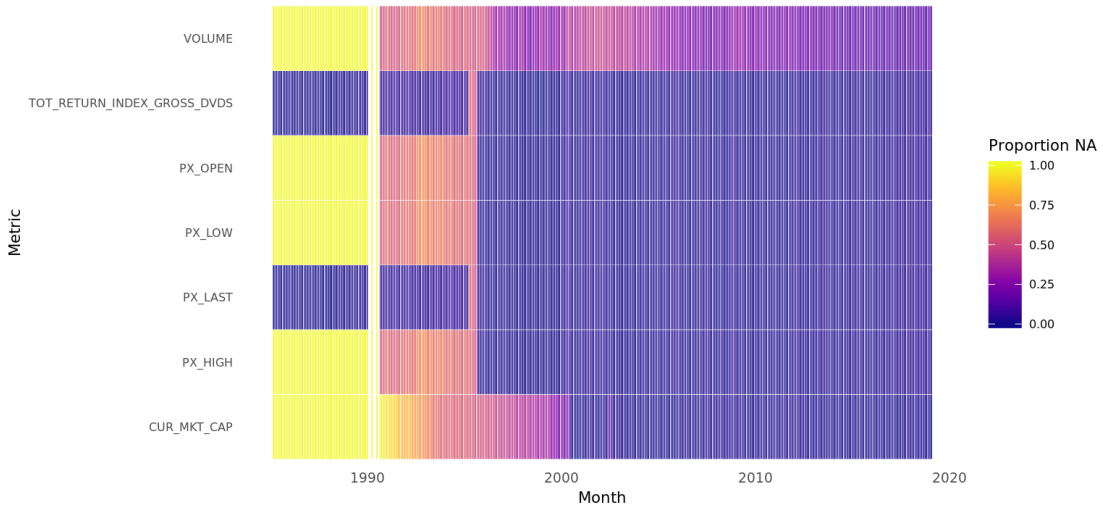

Figure 5.9: Market Metric Field Completeness, Filtered Dataset

The corresponding monthly `NA` timeseries plot is shown below. The best-quality period appears to be after 2010, and the base `NA` rate for that period has dropped from over 20% to under 5%.

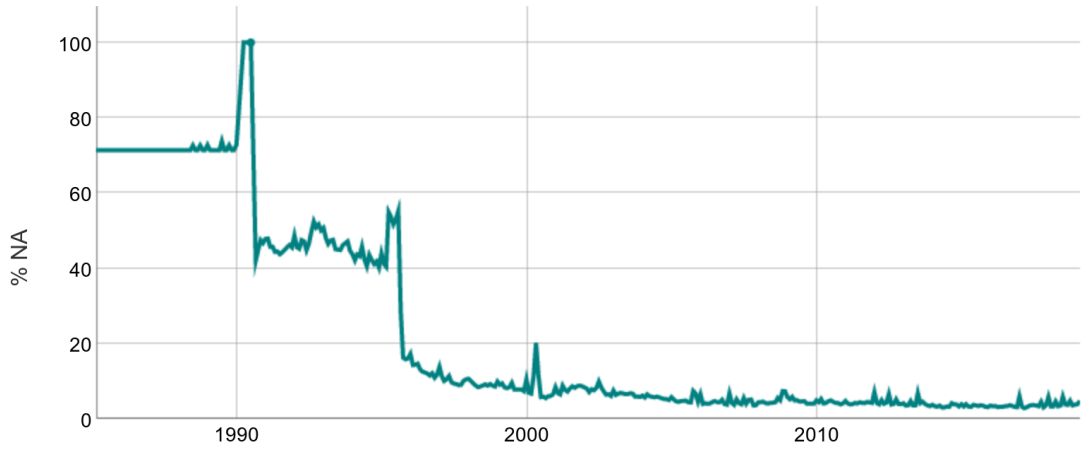

Figure 5.10: Monthly Overall Market NA Over Time, Filtered Dataset

This sort of filtering may limit the search space of profitable algorithms, but they also increase the accuracy of the backtest because less stocks are excluded from the exercise.

5.11.2. Trading engine validation

After obtaining data from a suitable source and validating its quality, the researcher is ready to conduct a backtest. This section performs a series of backtests on simulated data in order to validate that certain components of the trading engine are working as intended.

There are three necessary conditions for a backtest -

1) There is data in the `datalog`.
2) The backtest parameters are specified in the `parameters.R` file.
3) There is at least one algorithm file in the `trials/` directory.

If these three conditions have been met, the researcher need only `source("scripts/0_all.R")`, complete the prompts, and then wait for the results.

In order to validate that the backtester is functioning correctly, we run six backtests on identical data, with an identical algorithm.

In order to ensure that data quality is not an issue, we run these backtests against synthetically-generated data, which can be replicated by running `1_simulated_to_datalog.R` prior to running the catchall `0_all.R` script.

Table 5.4: Details of Each backtest Instance for Engine Validation Exercise

Trial ID	Variation
8	4 years, no transaction costs, rebalance daily, trade daily. Weight all tickers by market cap.
9	Same as 8 but rebalance monthly. Expect lower correlation with index than 1.
10	Same as 8 but rebalance quarterly. Expect lower correlation with index than 2.
11	Same as 8 but 0.5% transaction costs. Expect lower performance than 8.
12	Same as 8 but 5% transaction costs. Expect lower performance than 8.
13	Same as 12 but portfolio 10000 and minimum commission of 10. Expect transaction costs to eat portolio.

The results of the above backtests are available at https://github.com/riazarbi/equity_analysis_trials.

The correlation table below confirms the expected result from the table above. Trial 8 has the greatest correlation with the market return, followed by trial 9, then trial 10.

Table 5.5: Identical Portfolios, Varying Rebalancing Periodicity, Return Correlations

	trial_8	trial_9	trial_10	market_return
trial_8	1.0000000	0.9997183	0.9990962	0.9999890
trial_9	0.9997183	1.0000000	0.9994165	0.9997283
trial_10	0.9990962	0.9994165	1.0000000	0.9991166
market_return	0.9999890	0.9997283	0.9991166	1.0000000

With regard to the transaction costs test, trial 8 strictly dominates trial 11, which strictly dominates trial 12. As expected, trial 11 is much closer to trial 8 than trial 12.

Figure 5.11: Identical Portolios, Varying Transaction Costs

The low-value, high-minimum-commission portfolio (portfolio 13) also behaves as expected, descending immediately into bankruptcy. The tearsheet for this portfolio indicates that it only completed 442 trades. The portfolio did manage to trade sporadically until the end of the backtest window, whenever a stock it happened to own could be swapped for a desirable stock in the same heartbeat. This suggests that the code would benefit from a `break` in the event of bankruptcy.

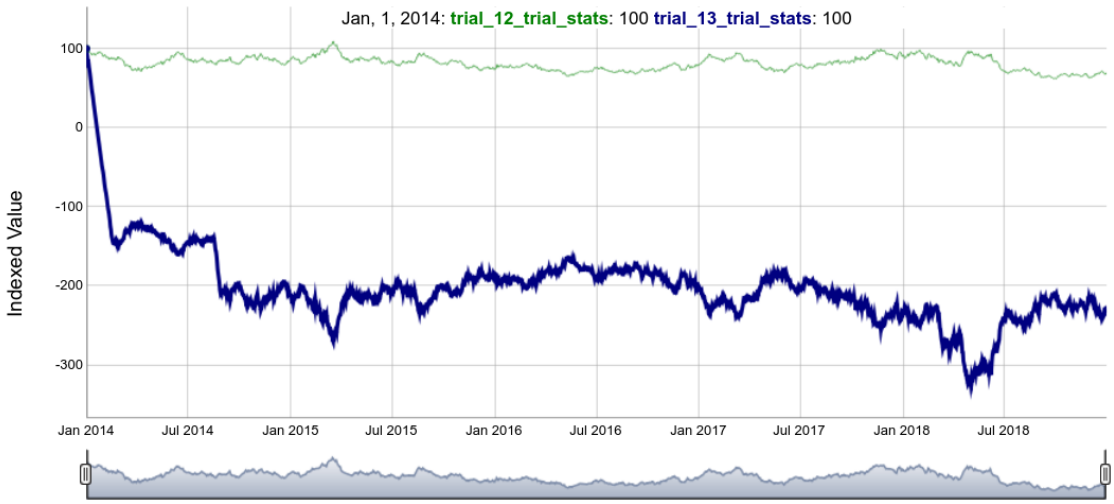

Figure 5.12: Identical Portolios, Smaller Portfolio Size and High Minimum Commission

These tests demonstrate how one would go about conducting a backtest from a cold start. They are fully replicable by anybody who downloads the code and the backtest parameters from the relevant code repositories. They also serve to prove the correct execution of certain features of the trading engine.

5.11.3. Example of Backtest Overfitting Detection

As detailed in 2.3.3, running many trials on the same dataset seriously increases the risk of backtest overfitting. This is especially risky for a process that is amenable to parameter tuning (varying rebalancing periods, portfolio sizes and weighting rules and the like). For this reason, we include a backtest overfitting report in the `results/` directory.

The `5_report.R` script packages the portfolio data from all trials into a pair of dataframes that can be fed into the R `PBO` packages to calculate a range of backtest overfitting tests. These tests are run by `6_cross_validate.R`, which computes the necessary statistics and renders the results in an html table.

In order to demonstrate this functionality, we simulate a set of portfolio returns and portfolio values

that would have been derived from truly random returns. We do this by creating an R `data frame` with 50 columns and 2000 rows; each of these cells has a random sample taken from the standard normal distribution. Each column represents a portolio and each row a date. We adjust the standard deviation to a reasonable daily rate ($sd = \frac{1}{365^2}$) to make the distribution compatible with a daily compounding rates. We assign a dates column to the `data frame` and then save it do disk, in the location that the cross validation script looks for portfolio array returns.

Next we create a price data frame from the returns data frame by converting each column into a rolling compounding column -

$$price_t = (1 + return_t) * (price_{t-1}) \tag{5.1}$$

We save this data frame to disk in the expected location as well. Next, we run `6_cross_validate.R` and inspect the html report.The array of returns is plotted below. The portfolio values appear to be truly random.

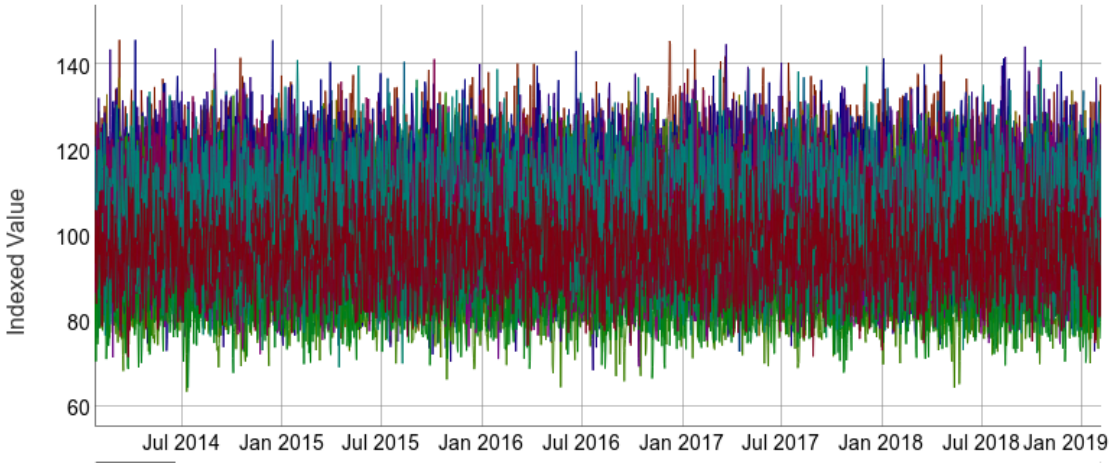

Figure 5.13: Portfolio Returns Time Series, Random Weighting Algorithms

Once the cross validation report has been generated we can inspect it to assess the probability of backtest overfitting. The excess return metric we use is the same as the metric used in Lopez de Prado (2013) - the Sharpe Ratio. This ratio is defined as

$$Sharpe = \frac{\mathbb{E}\left(Excess\ Return\right)}{\text{Std}(Excess\ Return)} \tag{5.2}$$

where the $ExcessReturn$ is the portfolio return over and above the risk free return.

The results of the PBO exercise are summarised below.

Table 5.6: PBO Results

	results	summary_Range	summary_Desirable
PBO	1	0->1	1
Slope	-8e-05	-inf->inf	NA
Adjusted R-squared	0.98	0->1	1
Probability of Loss	1	0->1	0

The histogram of logits confirm that the best in-sample trial performed below the median out of sample trial.

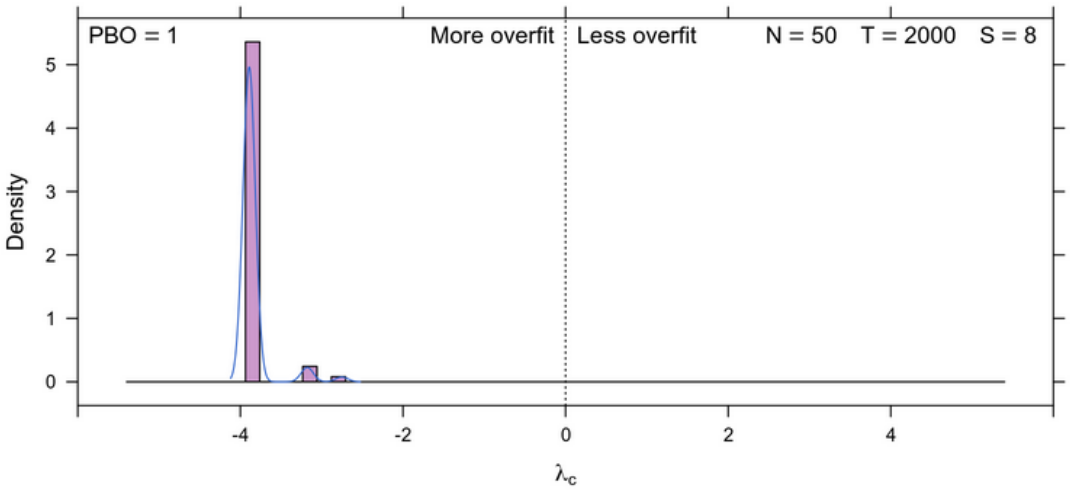

Figure 5.14: Histogram of Logits, Random Weighting Algorithms

The out of sample performance degradation indicates that greater in-sample performance results in worse out-of-sample performance.

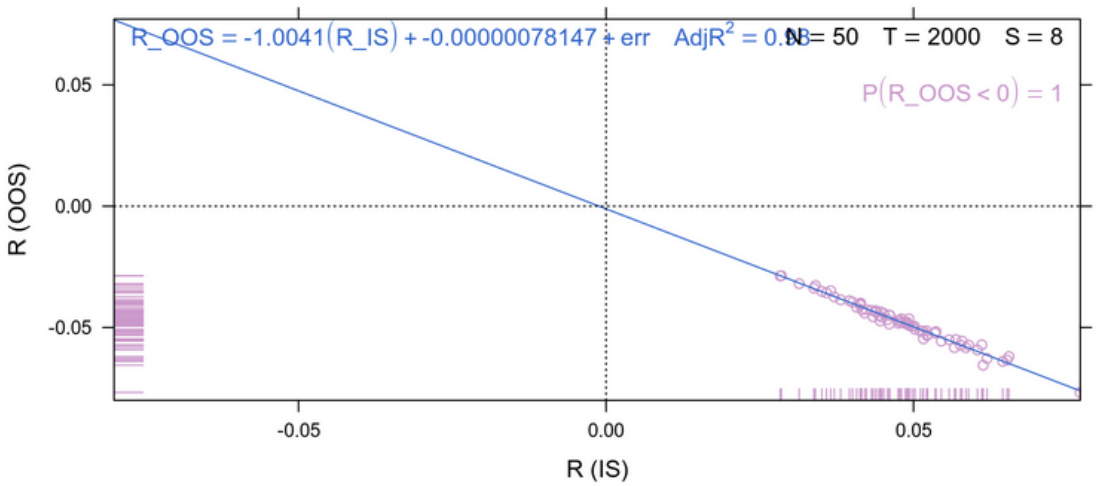

Figure 5.15: Out of Sample Performance Degradation, Random Weighting Algorithms

Finally, the pairs plot shows little relationship between in-sample and out-of-sample performance.

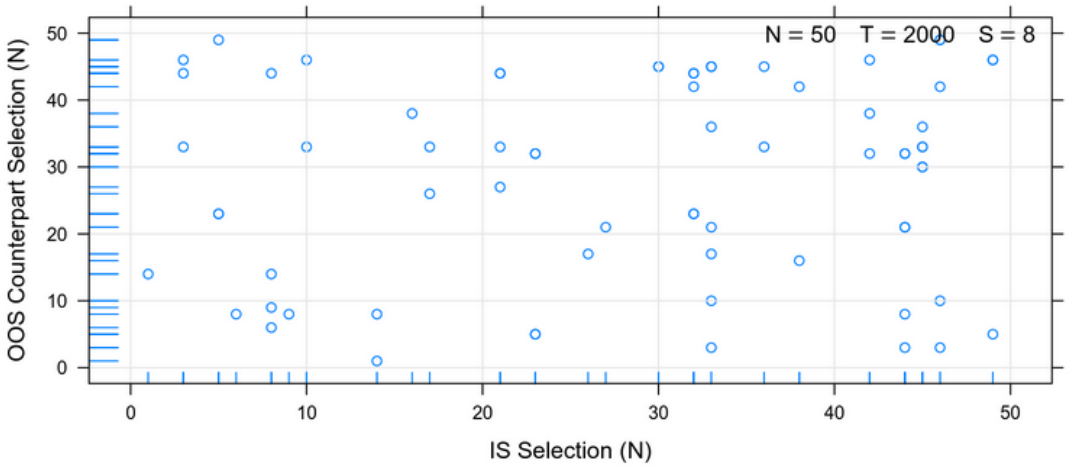

Figure 5.16: Pairs Pot, Random Weighting Algorithms

The cross validation report works as expected. Given a sample of truly random returns, some of which look promising, it reports that the result is probably a result of backtest overfitting.

In order to keep the scope of this project manageable, we have had to allow the codebase to exist with certain limitations. These limitations are not inherent; their existence is a function of time limitations.

5.12.1. Weaknesses

The data processing script (which converts the `datalog` into `datasets`) is highly opinionated, and expects only datalog-like files. In reality, source data comes in a variety of formats. The challenge of transforming these data structures into the strict datalog format is considerable, and the data-processing pipeline would benefit from becoming more versatile in what it can consume. This is a fine balance, however, a major feature of this project is that the results are totally reproducible from source.

Trading heartbeat processing time increases linearly with time. This is a function of disk IO, and the fact that the transaction log and trade history files (which are read and written each heartbeat) grow with time. This processing time could be significantly reduced by keeping both of these datasets in memory. This has not been implemented because reading and writing each heartbeat to disk facilitates transparency. If a backtest crashes, a researcher can load the files of the last heartbeat to investigate the cause. If the files were kept in memory this would not be possible. There are several possible solutions to this problem. For instance, the process could write each heartbeat, but read from memory. Or the IO could write to a ramdisk which is regularly synchronised to the hard disk, effectively decoupling the heartbeat processing time from the bottleneck. But these solutions are not without their risks, and the risks have not been weighed up as part of this project.

The crafting of portfolio-weighting algorithms is challenging. We have included three examples under `scripts/trading/sample_trials` - market capitalisation, equal, and random weighting. It would be useful to include more elaborate examples to give researchers a broader choice of starting points - especially with regard to subsetting portfolios according to sector and using fundamentals and ratios as a basis for portfolio weighting.

Transaction costs and slippage are modelled naively - either as a percentage of the total trade value or a hard minimum. These two factors encompass entire fields of research in their own right. It would be useful to generalise how these factors are incorporated into the backtest so that more complex estimation methods can be dropped in easily.

As present portfolio rebalancing periodicity is a global parameter. However, since the periodicity affects the return (there should be an optimal rebalancing periodicity that balances accuracy with churn), it should be a local parameter to a trial. Modifying the parameter to accept vectors, and modifying `3_trade.R` to run each trial against each periodicity would add significant flexibility to the

researcher's toolkit.

5.12.2. Enhancements

In line with our goal of enabling high-quality, reproducible research, it would be desirable to expand the data quality reporting to compare multiple data sources and make a 'est guess' synthetic dataset out of muliple fragmented sources.

On the other end of the pipeline, the tear sheets could be augmented with more summary statistics. Similarly to the data quality reporting, the underlying data is present in the appropriate form. What is needed is the computation of summary statistics and inclusion of those summary statistics in the parametrised report. This would require research into the relevant R packages and the modification of the `tearsheet.Rmd` template file into a more comprehensive report.

The `trade.R` script has been built to load a different set of trading functions depending on what trading mode (`BACKTEST` or `LIVE`) is selected. The tooling exists for the `trade.R` script to operate on live data and real trades unmodified. What is required is the building out of the `live_trading_functions.R` script to cover the same functions as the `backtest_trading_functions.R` script. Several brokers (for example, Interactive Brokers) offer paper trading accounts, which opens up the possibility of checking the backtest engine calibration by running the code in `LIVE` mode and then, after the fact, running the same algorithms over the same input data in `BACKTEST` mode and comparing the results.

Opening and reading each trial tear sheet is cumbersome, but allows a researcher to share the comprehensive results of a single trial in a human-readable format. Rapid comparison of each trial tear sheet would be made much less onerous by the building of a simple Shiny application that presents a drop-down list of all results in the results directory and allows a researcher to select one, immediately rendering the tearsheet.Rmd file inside that particular result subdirectory inside the Shiny app.

6. Conclusion

This paper has explored the various ways in which the lack of transparency in anomalies research makes it difficult to discern spurious results from genuine findings. This 'replication crisis' has strong analogues in other academic disciplines. We have argued that the 'reproducible science' response to this crisis in science at large has the potential to address much of the issues that bedevil anomalies replication.

We have introduced a collection of R scripts, organised into a compendium, that can be used to conduct anomalies research in a transparent and reproducible way. These scripts utilize only free, open source software and to organize data along the lines of 'tidy' data. Using plain text computer

code to collect, process, structure and analyse data represents a good approach to producing research that is easy to reproduce.

We avoid the problem of proprietary data distribution by including customizable data query scripts. These scripts query proprietary vendors and save the results in a standard way. Bundling these query scripts in a compendium enables a replicator to rebuild the dataset programmatically and non-interactively from source.

This collection of scripts make it possible to test an investment algorithm against an index of stocks, where each stock comprises a set of daily observations of price data plus an arbitrary number of attributes. The scripts use the event-based backtest method (as opposed to vectorized methods) which make it easy to avoid look-ahead bias and to introduce non-standard data to the algorithm. Transaction cost and slippage modelling, while rudimentary, exist and can be refined.

Using this code-base as a starting point should save a researcher a great deal of time in preparing stock data for backtesting, and the open source nature of the project ensures that any researcher can comb the operations for bugs or implement features that are not present. While it is not expected that this code base is necessary or sufficient for end-to-end backtesting, it represents a solid base for ongoing development.

It is hoped that researchers will treat this project as a reproducible project in and of itself. That is, they will inspect the code, ensure that it is working as intended (or suggest fixes) and thereby verify it as a legitmate base for engaging in the real meat of backtesting - discovering anomalies.

References

Arbi, Riaz. 2019. *A Reproducible Event-based Backtester Written in R*. https://github.com/riazarbi/equity_analysis.

Armstrong, Whit, Dirk Eddelbuettel, and John Laing. 2018. *Rblpapi: R Interface to 'Bloomberg'*. https://cran.r-project.org/package=Rblpapi.

Bache, Stefan Milton, and Hadley Wickham. 2014. "magrittr: A Forward-Pipe Operator for R." https://cran.r-project.org/package=magrittr.

Bailey, David H., Jonathan M. Borwein, Marcos López de Prado, and Qiji Jim Zhu. 2014. "Pseudo-Mathematics and Financial Charlatanism: The Effects of Backtest Overfitting on Out-of-Sample Performance." *Notices of the AMS* 61 (5): 458–71. https://doi.org/10.2139/ssrn.2308659.

Baum, Christopher F., and Selcuk Sirin. 2002. "Why should you avoid using point-and-click method in statistical software packages?" http://fmwww.bc.edu/GStat/docs/pointclick.html.

Baumer, Ben, Mine Cetinkaya-Rundel, Andrew Bray, Linda Loi, and Nicholas J. Horton. 2014. "R Markdown: Integrating A Reproducible Analysis Tool into Introductory Statistics." https://doi.org/10.5811/westjem.2011.5.6700.

Carl, Peter, Brian G Peterson, Joshua Ulrich, and Jan Humme. 2018. *quantstrat: Quantitative Strategy Model Framework*.

Damodaran, A. 2012. *Investment Philosophies: Successful Strategies and the Investors Who Made Them Work*. Wiley Finance Editions. Wiley. https://books.google.co.za/books?id=NkOkN0l3vHgC.

Fama, E., and K. French. 1992. "The Cross-Section of Expected Stock Returns." https://doi.org/10.2307/2329112.

Gentleman, Robert, and Duncan Lang. 2004. "Statistical Analyses and Reproducible Research."

Graham, B, D Dodd, and D L F Dodd. 1934. *Security Analysis: The Classic 1934 Edition*. McGraw-Hill Education. https://books.google.co.za/books?id=wXlrnZ1uqK0C.

Harvey, M., D. Hendricks, T. Gebbie, and D. Wilcox. 2017. "Deviations in expected price impact for small transaction volumes under fee restructuring." *Physica A: Statistical Mechanics and Its Applications* 471. Elsevier B.V.: 416–26. https://doi.org/10.1016/j.physa.2016.11.042.

Hou, Kewei, Chen Xue, and Lu Zhang. 2017. "Replicating anomalies." *NBER Working Papers*, no. No. 23394. https://doi.org/10.2139/ssrn.2190976.

Ioannidis, John P. A. 2005. "Why Most Published Research Findings Are False." *PLoS Medicine* 2 (8). Public Library of Science: e124. https://doi.org/10.1371/journal.pmed.0020124.

Koenker, Roger, and Achim Zeileis. 2009. "On reproducible econometric research." *Journal of Applied Econometrics*. https://doi.org/10.1002/jae.1083.

Li, Shengying. 2004. "A Survey on Tools for Binary Code Analysis."

Loonat, Fayyaaz, and Tim Gebbie. 2018. *Learning zero-cost portfolio selection with pattern matching*. Vol. 13. 9. https://doi.org/10.1371/journal.pone.0202788.

Lopez de Prado, Marcos. 2013. "The Probability of Back-Test Over-Fitting." *SSRN Electronic Journal*. https://doi.org/10.2139/ssrn.2308682.

Marwick, Ben, Carl Boettiger, and Lincoln Mullen. 2018. "Packaging Data Analytical Work Reproducibly Using R (and Friends)." *American Statistician*. https://doi.org/10.1080/00031305.2017.1375986.

Munafò, Marcus R, Brian A Nosek, Dorothy V.M. Bishop, Katherine S Button, Christopher D Chambers, Nathalie Percie Du Sert, Uri Simonsohn, Eric Jan Wagenmakers, Jennifer J Ware, and John P.A. Ioannidis. 2017. "A manifesto for reproducible science." https://doi.org/10.1038/s41562-016-0021.

Peterson, Brian G. 2017. "Developing & Backtesting Systematic Trading Strategies," no. June: 42. https://doi.org/10.20964/2016.12.40.

Peterson, Brian G, and Peter Carl. 2018. *PortfolioAnalytics: Portfolio Analysis, Including Numerical Methods for Optimization of Portfolios*. https://cran.r-project.org/package=PortfolioAnalytics.

Quantopian. 2018. "Improved Backtest Analysis." https://www.quantopian.com/posts/improved-backtest-analysis.

Quantpedia. 2018. "Backtesting Software." https://quantpedia.com/Links/Backtesters.

Raymond, Eric S. 2003. *The Art of UNIX Programming*. Pearson Education.

R Core Team. 2018. "R: A Language and Environment for Statistical Computing." Vienna, Austria. https://www.r-project.org/.

RStudio Team. 2015. "RStudio: Integrated Development Environment for R." Boston, MA. http://www.rstudio.com/.

Stodden, V, D H Bailey, J Borwein, R J Leveque, W Rider, and W Stein. 2013. "Setting the Default to Reproducible Reproducibility in Computational and Experimental Mathematics." In *ICERM*

Workshop, 19. http://www.davidhbailey.com/dhbpapers/icerm-report.pdf.

Trice, Tim. 2017. *Backtesting Strategies with R*. https://timtrice.github.io/backtesting-strategies/index.html.

Van Roy, P. 2009. "Programming Paradigms for Dummies: What Every Programmer Should Know." *New Computational Paradigms for Computer Music*, 9–47. https://doi.org/10.1016/j.tet.2003.02.005.

Wickham, Hadley. 2014. "Tidy Data." *Journal of Statistical Software*. https://doi.org/10.18637/jss.v059.i10.

———. 2017. "tidyverse: Easily Install and Load the 'Tidyverse'." https://cran.r-project.org/package=tidyverse.

Zipline. 2018. "Zipline Beginner Tutorial — Zipline 1.3.0 documentation." https://www.zipline.io/beginner-tutorial.html.