A Data Scientific Approach to Equity Backtesting

Riaz Arbi[a]

[a] *University of Cape Town, South Africa*

Abstract

Abstract to be written here. The abstract should not be too long and should provide the reader with a good understanding what you are writing about. Academic papers are not like novels where you keep the reader in suspense. To be effective in getting others to read your paper, be as open and concise about your findings here as possible. Ideally, upon reading your abstract, the reader should feel he / she must read your paper in entirety.

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

JEL classification

Table of Contents

Email address: (Riaz Arbi)

1. Introduction

Intro intro intro

This paper is organised as follows. Section 2 briefly introduces the practice of backtesting of 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, with a special focus on the R statistical programming language, the Tidy approach to data organization and the use of scripting to facilitate reprducibility and transparency in scientific research. Section 4 introduces a 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.

2. Backtesting in Academia and Industry

Investigations into why some stocks outperform others 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 that stand up to scrutiny.

Academics often frame this challenge as attempting to identify whether a population subsetted by the presence or absence of an attribute exhibit statistically significant differences in returns. Fama and French (1992), for instance, split the US stock universe by market capitalization and market 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.

Traders, in contrast, tend to frame the question in terms of raw performance - does a trading rule result in abnormal (risk adjusted) returns over and above some benchmark? There is a rich ecosystem of software packages, tutorials, online communities, books and blog posts that attmepts to tackle this question (see Quantpedia (2018) for a list of software packages).

Confusingly, both of these broad approaches are encompassed by the generic term 'backtesting'. To avoid confusion, we adopt a broad definition from D. H. Bailey et al. (2014) :

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

Because these two communities frame the problem differently, they have developed distinctive approaches. Academics tend to focus on using statistical techniques to explain abnormal returns in terms of independent variables. Traders tend to focus on prediction; 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)).

2.1. Common areas

A common area, however, is in interrogation. Both groups 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 derive rules which perform well IS but poorly OOS (D. H. Bailey et al. (2014)).

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

2.1.1. Data quality issues

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 present, 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. To base our data only on this 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 unuverse 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 histrical 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 Currenr Assets at 31 December 2003, for instance, is only known at release of the annual financial statements, which might have been on the 27th February 2004. Joining this data point to marke tdata on the date 31 December 2003 will introduce look-ahead bias to your dataset.
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.

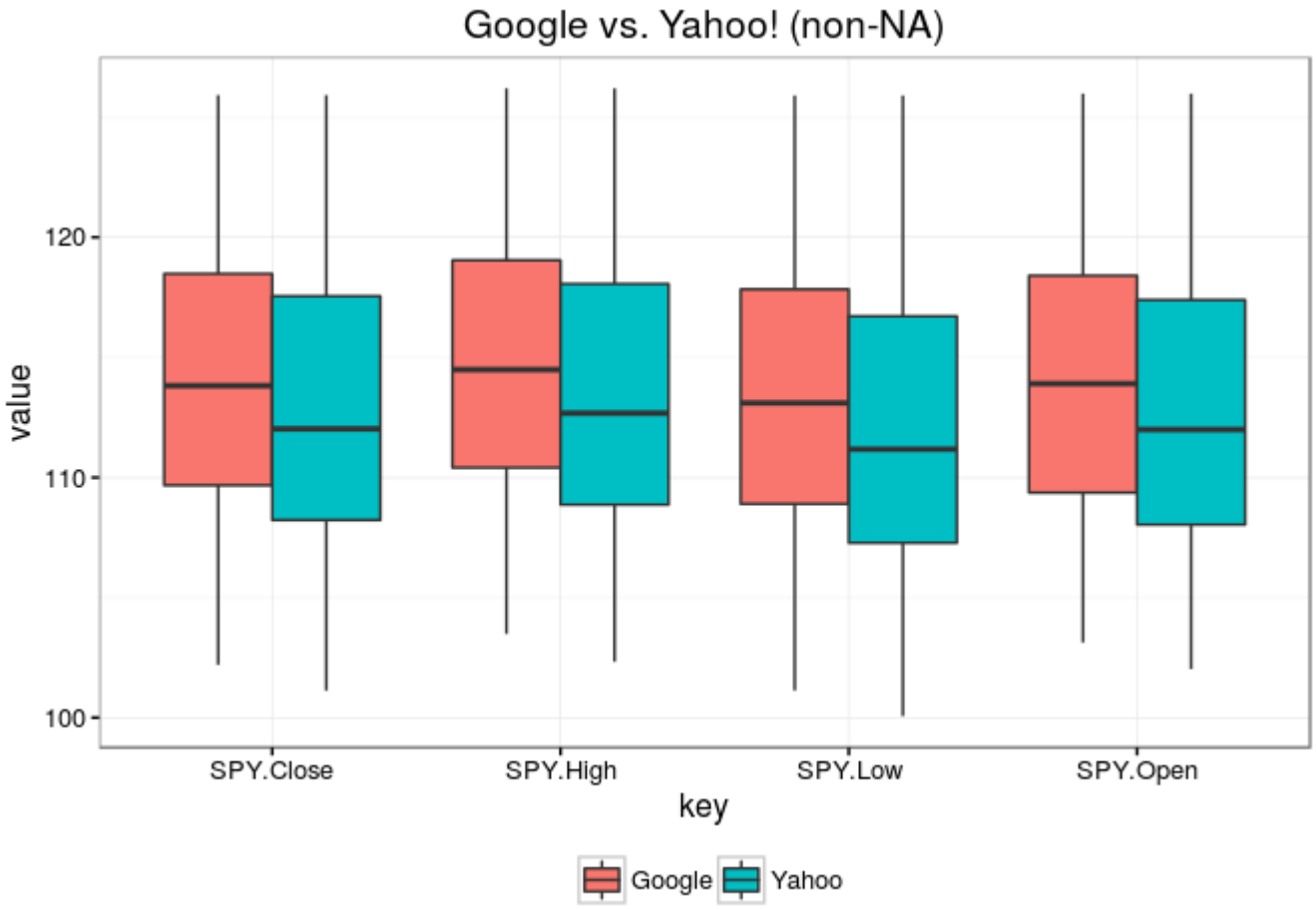

Figure 2.1: Data Discrepancies between Google and Yahoo! (Trice (2017))

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 accountng 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.

2.1.2. Modeling 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 appear to exist be related to illiquidity. That is to say, the returns of a stock (or class of stocks) may appear to be abnormal, but the abnormal returns may be due to the market impact of a trade on the stock. A realistic model will adjust for this by assuming minimal impact for only a small fraction of daily volume traded, build in a price impact if the trade size exceeds this volume, and respect the bid/ask

spread when desiding that a trade matches.

2) Have *transaction costs* been taken into account? Similar to market impact, transaction costs form varying percentages of a trade; this depends on the commission structure. 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 exhorbitant minimum commission for small lots.

3) Failing to account for *short positions or portfolio bankruptcy*. Portfolios can be long only or allow for short positions. Short positions involve interest charges that need to be taken into account. Long only portfolios cannot have negative positions. Similarly, portfolio cash balances cannot be negative of no leverage (which attract interest charges) is employed. Portfolio bankruptcy needs to be accounted for - if a strategy has an average excess return of 2%, but there are two periods in which the return is -100%, then the average return does not matter.

4) Data mining and *backtest overfitting*. When a researcher iterates over many strategies while recycling the same dataset, is is easy to overfit a model. Overfitting a model is when one selects rules which perform well in sample but poorly out of sample. That is, they model the noise in the training set, not the signal (D. H. 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 delat 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)).

2.2. 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 avaialable at 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.

2.2.1. 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, prepearation methodologies, mathematical errors or some other factor unrelated to the question at hand.

The academic discipline of anomalies research has grown into a substantial corpus of contradictory claims. These contradictions are mirrored in the business of investment management by competing investment philosopies, 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 geniune facts or the result of methodological differences.

Hou, Xue, and Zhang (2017) attempt to replicate the entire anomalies literature to identify with 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):

- Specifying datasets: Compustat Annual and Quarterly Fundamental Files; Center for Research and Security Prices (CRSP).
- Specifying breakpoints: Use NYSE breakpoints
- Use value-weighted, not equal-weighted portfolios
- No sample screening (ie don't exclude stocks because of some arbitrary cutoff)
- For annually-composed portfolios, resort at the end of June
- Incorporate fundamental data four months after valid date

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.

2.2.2. 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.

The approach taken by developers of trading backtesters has been to build 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 follows. 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 2.1.1 still exist. In many ways, these issues are worse because these packages default to free data sources such as Yahoo!, Google Finance or Quandl, 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 2.1.1.

These shortcomings in free data sources drive areas of research: 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.

These backtesting packages are able to easily switch between simulated and live trading because they implement event-based backtesting methods. Event-based backtesting allow a trader to step through time, either in fixed increments or through the consumption of a continuous information stream, and trigger actions according the the information made available to them. If implemented correctly, the event stream can be switched from a historical dataset to a real-time stream with minimal changes to the code. This eliminates a possible source of error: the algorithm does not need to be rewritten into

a live implementation.

There is something of an uncanny valley 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 forces 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 the size of a program grows, 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 thouroughly cross-checked before release.

Worse still, because follow-up replication is difficult to impossible with poor documentation or data availability, 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 to be false it is to be true (J. P. A. 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 evey single mathematical operation conducted with pencil and paper would be extremely laborous; 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 dinstinguish reveiwability (which assumes mathematical competence and focuses on reasoning) from reproducibility (which allow the extension of the review into assessement 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
Reproducable	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 uses reproducibility in the sense defined in 3.1, 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 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* is *compiled* or *interpreted* by another program into binary machine code (which is not readable by humans) and distributed for execution. The software for the popular spreasheet program Microsoft Excel, for instance, would be written by a group of humans in a 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. 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 publically 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 readilty available (Li (2004)).

By definition, the text-based nature of source code makes all source code reproducible (and, by corollary, auditable). 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 as the operating systme level 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 practices for creating and maintaining large, open, verfiable and repeatable enviroments. 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 coverage; 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. 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 environemnts), 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 as a project to build out an open source statistical programming environment. In development since 1997, base R is now a mature ecosystem comprising a scriptable language, graphical rendering system and debugger (R Core Team (2018)). A large community of third party developers (some commercial, most free) have extended base R with over 13000 packages, including a fully-fledged interactive development environmnent (Rstudio), interactive dashboarding web server (Shiny) and bindings to LaTex for simple 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 verison 2, which "does not restrict anyone from making use of the program in a specific field of endeavor" (R Core Team (2018)). This free avaialability means that anybody with a commodity computer and an internet connection can install and use R.

Although the reproducible research community is principles-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*, and explore how research done in R can be packaged into compendia. They argue that a compendium is defined by three principles -

1. Files should be organised according to a prevailing standard so that others can immediately understand the structure of the project, and pipe the compendium to environments that expect the standard structure with no structural modifications.
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 on the undelrying tooling needed to support the analysis.

These principles are intended to ensure that, given a certain input (the data) and a certain operation (the method), a deterministic output results (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, modeled 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-only document using the RMarkdown format - and exported to a wide range of academically accepted typsets 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 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, readibility, modular code and common standards have been extended into the prcatice 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)). According to Wickham, tidy data 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 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 of standards enables the decoupling of processes - process (a) can produce some peice of analytical output *with no knowledge of what it will be used for*. As long as it is 'tidy', and 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 push 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 judgment calls involved in building a rigorous backtest inhibit replicability, both in academia and in a prefessional 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 in general, and the R programming language and ecosystem for acquisition-to-publication analysis in particular.

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

4. Be able to run on a typical researcher's computer
5. Produce artifacts (models and data objects) that are portable across the academia / industry divide
6. Encompass the full analysis chain from aquisition through to reporting, with a particular focus on raw data processing
7. Accommodate deep customizability without requiring excessive complexity of code
8. 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.

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 non0-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 trasparent 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 detail a rudimentary reference implementation of a reproducible backtester.

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

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

All code is available at http://github.com/riazarbi/equity_analysis.

The data processing scripts can consume multiple data sources, for multiple equity indexes, for any timeframe, 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 rudimendary log compaction and data versioning to enable point-in-time analysis and check data quality.

Datasets are read into memory for backtesting, which is done in a step-through fashion. That is, the backtester steps through time, is presented with data that would have been available to it at that date, and executes trading rules which are persisted to a trade log and trasnaction history.

This backtester has been built for multi-trial, parametrised 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 per Lopez de Prado (2013).

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

Input data has been segregated from 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 regerated 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. After querying data from an appropriate vendor and processing the data using `scripts/1_process_data.R`, a researcher will specify global parameters in the `scripts/run_trials.R` file and place a batch of algorithms in the `trials/` directory. Running the `scripts/2_run_trials.R` script will backtest each algorithm and save the results in the `results/` directory. Running `scripts/3_cross_validate.R` will run the R package `pbo` to test for the probability of backtest overfitting.

These results will 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.

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 on 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 a range of simulated index datasets, complete with constituent lists, metadata and market and fundamental ticker timeseries. These 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

This is the intellectual property of Riaz Arbi. This code will be released under a strong copyleft license. A license is included in the repository.

5. Detailed Documentation, With Design Justifications

Because the workflow is linear and procedural in design, this documentation will proceed in a linear, procedural fashion. After familiarising the reader with the directory structure, we will work through the code in a stepwise 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 four stages - querying, data processing, backtesting, cross-validation and reporting. Each stage expects to read in 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 I/O optimizations. This includes converting `.csv` files to `.feather` files, which are much faster to read and write, and parallelizing select prices of code. Ticker-wise data pipelining is multi-threaded, and the trading engine could be sped up by making each trial run on its own thread.

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 verisoning 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 alway see exactly what has changed, who has made the changes and what the reason for the change was.

5.2. 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 porject license. This information is duplicated in the REQUIREMENTS file but a separate LICENSE file is expected for automatic parsing by hosted git repostiories 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
|    scripts/
|    |   1_query_source.R
|    |   2_process_data.R
|    |   3_run_trials.R
|    |   4_cross_validate.R
|    |   5_report.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
|    |   trading/
|    |   |   example_trial.R
|    |   |   load_slow_moding_data.R
|    |   |   trade.R
|    |   reporting/
|    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/s` directory on `data/datasets` in order to generate results, which goes into the `results/` directory. A detailed data flow diagram is rendered below.

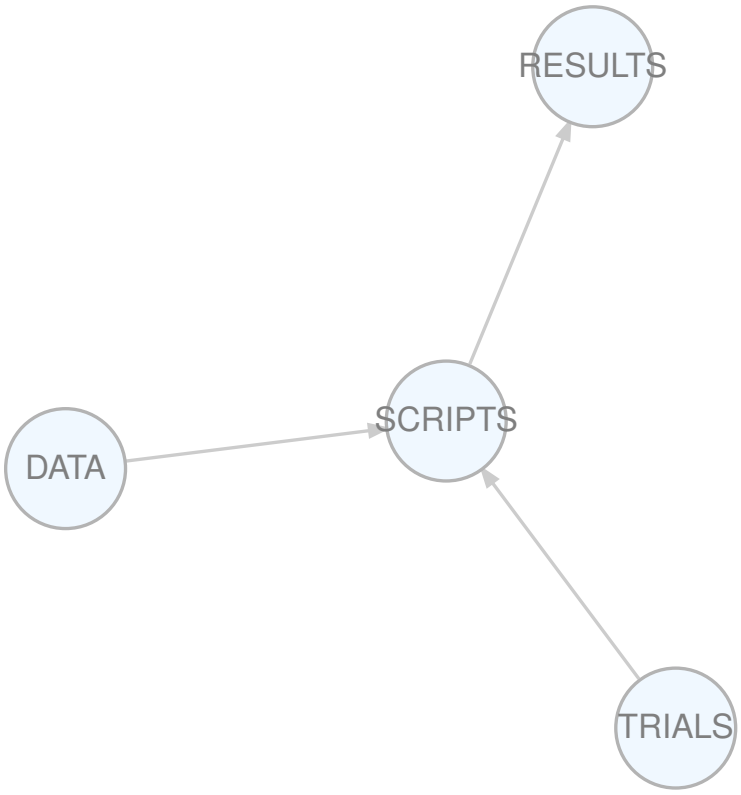

Figure 5.1: Coarse Analytical Workflow

5.3. 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/
|    |   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.

`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 (**???**)'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 tickers based on ticker code.
9. Extract and save fundamental data for each tickers based on ISIN.

5.3.1. 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
```

5.3.2. 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 tablular, with each row being an observation (i.e a date) and each column an attribute (eg. 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

date	BS_ACCT_NOTE_RCV	BS_INVENTORIES
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

5.3.3. 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.

5.4. 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`.

5.4.1. 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 (note - 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 procssing operations.

Secondly, is 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 chang 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 dealwith this? THere is no easy way to drop a single attribute for a songle 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 datapoint, 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 wide formats also have disadvantages, especially when one wants to perform rowwise computations across attributes. Because of this, we convert the datasets back to *short and narrow* format during backtesting.

5.4.2. 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.

5.4.3. 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:
    - Read in logfiles relevant to the dataset
    - Convert to *tall and narrow* format
    - Check if a dataset already exists and, if it does, read it in to memory

- Join the datasets
- Filter out and NA values
- Remove any duplicates
- 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 wothout loss of information.

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

1. Each file does not correspond real thing. The constituent and metadata files contain all data for all indexes and tickers in a single file.
2. Each attribute is not a column.

The datsets are converted to Tidy format in-memory at backtest runtime.

5.5. Backtesting - `3_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/
|    |   3_run_trials.R
|    |   trading/
|    |   |   example_trial.R
|    |   |   load_slow_moding_data.R
|    |   |   trade.R
|    trials/
|    results/
```

Rather, these scripts follow a parent - child relationship. The parent script is `3_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.

5.5.1. Global parameters

The first section of the script sets parameters global to the session. These parameters are -

1. Mode
    - `run_mode` - BACKTEST or LIVE. LIVE mode not yet implemented.
    - `heartbeat_duration` - time between trade loops, in seconds. Typically between 600 and 3600.
2. Universe
    - `constituent_index` - index to backtest against.
    - `data_source` - data source to use.
    - `market_metrics` - market metrics to filter (defaults to all).

- `fundamental_metrics` - fundamental metrics to filter (defaults to all).

3. Timeframe
   - `start_backtest` - YYYYMMDD start of backtest.
   - `end_backtest` - YYYYMMDD end of backtest.
4. 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.
5. 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.

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 build in 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 seize 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 modeling 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 with the original results.

[FLOW CHART OF OPERATIONS INSIDE THE TRADE SECTION]

5.5.2. The `ticker_data` and `runtime_ticker_data` objects

The trading engine reuses several dataframes during the course of operation. 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 sourcing `load_slow_moving_data.R`, which expects to find `3_run_trials.R` parameters in the global environment.
    1. It reads in the `constituent.feather` and `metadata.feather` 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*, Tidty 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.

5.5.3. 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 fo 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 wont 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.

5.5.4. 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` by sourcing `load_slow_moving_data.R`
5. Check if `runtime_ticker_data` is less than 24 hours (in heartbeat time) old.
    - If not, 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.
6. Get the `transaction_log` and `trade_history` data objects and compute our current portfolio and cash positions.
7. Query latest prices for the positons 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 `hearbeat_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 avaiable 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 extenral 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 of size 1 for 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 thoughout 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 avaialble 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.

5.5.5. Backtest results

A well specified `compute_weights` function will result in a successful backtest. The results of each backtest run by `3_run_trials.R` will be placed in the `results` directory according to the naming convention covered earlier.

These `results` subdirectories contain five 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 fro 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.
5. A `summary_statistics.Rmd` file, which presents an historical summary of the strategy performance, including (but not limited to) sharpe ratio, annualized return over a range of horizons, maximum drawdown, turnover and total expense ratio.

5.7. Reporting

Must address:

1. Survivorship bias
2. Look ahead bias
3. Completeness: multi source
4. Retroactive corrections: dataset monitoring
5. Market impact
6. Transaction Costs
7. Portfolio bankruptcy
8. Backtest overfitting: pbo

Must try to:

1. Be convertalbe to live with minimal effort
2. Acommodate any sort of date
3. Be interoperable with other packages (xes. . .)
4. Use tidy data structures

Second-order challenges Compaction: dropping useless data Speed: training vs trading Speed: parallel processing Speed: read/write: feather Memory: filtering and subsetting

5.8. Features

- data versioning
- multi-source
- multi-index
- code version control
- clear procedural workflow
- event-driven
- tidy data
- pbo
- host of metrics
- bias avoidance

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 geniune 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 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 programatically 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 research 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.

Katzke (2017)

References

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. doi: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." doi: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." doi: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.

Hou, Kewei, Chen Xue, and Lu Zhang. 2017. "Replicating anomalies." *NBER Working Papers*, no. No. 23394. doi: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. doi:10.1371/journal.pmed.0020124.

Katzke, N F. 2017. "Texevier: Package to create Elsevier templates for Rmarkdown." Stellenbosch, South Africa.

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

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

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

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

Munafò, Marcus R, Brian A Nosek, Dorothy V.M. Bishop, Katherine S Button, Christopher D Cham-

bers, Nathalie Percie Du Sert, Uri Simonsohn, Eric Jan Wagenmakers, Jennifer J Ware, and John P.A. Ioannidis. 2017. "A manifesto for reproducible science." doi:10.1038/s41562-016-0021.

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

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

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

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

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

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. doi:10.1016/j.tet.2003.02.005.

Wickham, Hadley. 2014. "Tidy Data." *Journal of Statistical Software*. doi: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.