The 10 Rules of Reliable Data Science

From alchemy to engineering in exploratory data workflows

DRIVENDATA

hello@drivendata.co | www.drivendata.co

The 10 Rules of Reliable Data Science

Introduction: The problem with data science work practices **2**

Rule 1: Start organized, stay organized **6**

Rule 2: Everything comes from somewhere and the raw data is immutable **8**

Rule 3: Version control is basic professionalism **9**

Rule 4: Notebooks are for exploration, source files are for repetition **12**

Rule 5: Tests and sanity checks prevent catastrophes **15**

Rule 6: Fail loudly, fail quickly **17**

Rule 7: Project runs are fully automated from raw data to final outputs **19**

Rule 8: Important parameters are extracted and centralized **22**

Rule 9: Project runs are verbose by default and result in tangible artifacts **25**

Rule 10: Start with the simplest possible end-to-end pipeline **27**

Conclusion **31**

Appendix A — Putting it into practice: software lessons learned the hard way **32**

Appendix B — Further readings on correctness and reproducibility **35**

Introduction: The problem with data science work practices

We heard this story firsthand from the data scientist in question.
We won't share the company name, but you have heard of it.

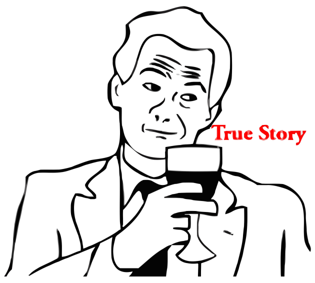
In 2017, a lead data scientist at a global energy company with tens of thousands of employees was tapped to take over a late-stage data science project and lead it to the finish. He was excited to be productionizing a major analysis that one of their PhD hires had been working on in secret for the past six months.

Using the company's existing sensor and geological survey data, the analysis suggested an efficient way to harness pre-existing capital infrastructure to get more out of existing fuel deposits. The company's planning staff had already committed time and resources in the next fiscal year to exploit this unexpected windfall, and senior leadership in the business unit was gleeful.

When the lead data scientist took over the project, it was the usual tangled web that he had come to expect from "research code"—scripts here, interactive notebooks there, data pulled from a variety of sources, code copied and reused with small tweaks in various places. This sort of hodgepodge is fairly common in data science work, so he was not initially worried. But soon he found cause for concern: after refactoring (cleaning up) some of the data loading code he could no longer recreate the original, interesting results.

Nervously, he pulled in the PhD and they began to comb through the code bit by bit.

At a certain point, they both arrived at a realization that made their hearts sink: somewhere in the original data cleaning code, arrays representing fuel yield were joined multiple times, inflating the apparent value of the land tracts being analyzed.



After months of work and millions of dollars earmarked, the promise of the analysis evaporated.

What part of correctness is still difficult for data science teams?

> "It is remarkably easy to incur massive ongoing maintenance costs at the system level when applying machine learning."
>
> — Sculley et al, *Machine Learning: The High Interest Credit Card of Technical Debt*

Organizations have realized that data science provides a powerful toolbox for increasing efficiency, driving growth, automating expensive processes, and making better decisions. **That's a good thing.**

But data science as a professional discipline is still in its infancy. Too many real-world projects are a chaotic stew of hand-tweaked algorithms, cherry picked examples, and brittle, undocumented, untested research code. **That's a bad thing.**

The business case for investing in data science work depends heavily on the reports, conclusions, and recommendations actually being correct.

We believe that some the main bottlenecks in exploratory data science workflows are no longer compute power or sophisticated algorithms, but rather *craftsmanship*, *communication*, and *process*.

Experience shows that it is hard enough to read even a small spreadsheet and understand what the columns mean or how the formulas work. Once programmatic data manipulation and probabilistic methods are involved, the complexity increases beyond the capacity of human teams to fully understand all of the moving pieces or recognize all of the possible points of failure.

The fact is that "quality" in data science work means drawing correct inferences about the world and making accurate predictions about the future. **Shoddy, disorganized analysis is bad analysis, and bad analysis costs money and wastes time.**

Sometimes, the conclusions of disorganized analysis are simply wrong. More frequently, they are "right-ish" but could have been more effective if the R&D work had been approached more systematically.

Toward greater reliability in data science work

> "The central enemy of reliability is complexity. Complex systems tend to not be entirely understood by anyone. If no one can understand more than a fraction of a complex system, then, no one can predict all the ways that system could be compromised."
>
> — Geer et al, *CyberInsecurity (2003)*

Our argument has three basic premises:

- Data science work is a kind of software;
- The correctness and reliability of software depends on development practices;
- Therefore, data science quality depends on software engineering quality.

If you accept these premises, you will agree that data science practitioners and managers need to understand and promote the processes, norms, and tools that result in quality data science software.

We've distilled our perspective on this issue into 10 guidelines that teams can consider adopting to build trust in their work products and encourage professionalism along the way.

What this is, what this is not

Our perspective has been shaped by reviewing a large number of data science work products, and **we are trying to articulate some opinionated best practices** given this empirical view of what practitioners are actually doing within companies and in their open source and data competition projects.

If your organization already does all of these things or has consciously made different choices, that's great! **We are not proposing a one-size-fits-all approach** to data science work, and we encourage teams to adapt the principles they find most relevant to their work.

It's also true that some of the practices we recommend are addressed by commercial products aiming to instill more opinionated and organized workflows onto data science work, or tools focused on orchestrating and monitoring code "in production." While we applaud any attempts to instill more structure and rigor, it is our observation that most working data scientists still do all (or significant parts) of their research work in folder- and file-based projects, using notebooks running locally rather than in specialized cloud environments or MLOps products.

For that reason, **this advice is primarily tailored to “R&D”-style exploratory analysis and modeling** rather than to code running in production or work being done in proprietary environments. Still, many of the practices described apply similarly even when using commercial data science products.

It's not just about preventing errors

To be sure, a large part of engineering discipline is targeted towards preventing mistakes, and we discuss many norms and practices that can help. But reproducibility and craftsmanship go beyond assuring others that the work is not wrong.

As practitioners, we should aim for work that is accurate and correct, but also work that *can be understood* without heroic efforts, work that others can collaborate on, and that can be improved and built upon in the future even if the original contributors have stepped away.

Reproducibile and reliabile work practices enable the type of returns to data R&D that people are looking for, and supports practitioners so they are better able to work together, share with colleagues, and build on past work.

Rule 1: Start organized, stay organized

"Pipeline jungles often appear in data preparation. These can evolve organically, as new signals are identified and new information sources added. Without care, the resulting system for preparing data in an ML-friendly format may become a jungle of scrapes, joins, and sampling steps, often with intermediate files output. Managing these pipelines, detecting errors and recovering from failures are all difficult and costly. ... All of this adds to the technical debt of a system and makes further innovation more costly."

— Sculley et al, "Machine Learning: The High Interest Credit Card of Technical Debt" (2014)

It's no secret that good analyses are often the result of scattershot and serendipitous explorations. Tentative experiments and trying out approaches that might not work are all part of the process for getting to the good stuff, and there is no magic bullet to turn data work into a simple, linear progression.

That said, once started this is not a process that lends itself to thinking carefully about the structure of the code or project layout. It is best to start with a clean, logical structure and stick to it.

So how should a project be structured?

We may be biased, but we think our Cookiecutter Data Science project template is an effective structure for starting an organized project and encouraging teams to stay on the right track.

In this template, for example, data is always in `data/`, with the original data in `data/raw/` and the final, cleaned version used for analysis in `data/processed/`. Notebooks are in `notebooks/`, and we encourage a numbering scheme to give a sense of order. Project-wide code is placed in `src/` and it can be imported from the notebooks to encourage deduplication and standardization. This sort of sensible and self-documenting structure allows others to understand, reproduce, and extend your analysis, and builds a sense of trust that the project was carried out prudently and professionally.

(Ours is far from the only project template out there, for example there is also Project Template which came out of the R data science ecosystem and shares many of the same values.)

The bottom line is that staying organized should not require deep reserves of discipline and willpower; the underlying project structure and tools should all combine to make the process of discovery easy and fun. If you are not already convinced that project organization is worth worrying about, here are two good reasons to give it higher priority.

Other people will thank you. A well-defined, standardized project structure means that a newcomer can begin to understand an analysis without sifting through extensive documentation. It also means that they don't have to read 100% of the code before knowing where to look for specific things. Every data scientist on our team can open any historical project we have worked on, and immediately know where to find the inputs, the output, the exploration, the final models, and any reports that were generated.

Well organized code is self-documenting in the sense that the placement of well named files itself provides intuitive context for project contents. People will thank you for this because they can:

- Collaborate more easily with you on this analysis
- Learn from your analysis about the discovery process and the underlying domain
- Feel confident in the conclusions at which the analysis arrives

You will thank yourself. Have you ever tried to reproduce an analysis that you did a few years or even a few months ago? You may have written the code, but it's now impossible to decipher whether you should use `make_figures.py.old`, `make_figures_working.py` or `new_make_figures01.py` to get things done. Here are some questions we've learned to ask with a sense of existential dread:

- Are we supposed to go in and join column X to the data before we get started or did that come from one of the notebooks?
- Come to think of it, which notebook do we have to run first before running the plotting code: was it "process data" or "clean data"?
- Where did the shapefiles get downloaded from for the geographic plots?
- *(Et cetera, times infinity.)*

These types of questions are painful and are symptoms of a disorganized project. Good project structures encourage sound practices that make it easier to come back to old work.

Rule 2: Everything comes from somewhere and the raw data is immutable

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

— Andy Hunt and Dave Thomas, *The Pragmatic Programmer*

Let's be grandiose and call this **"the fundamental theorem of reproducibility": every conclusion drawn in an analysis must come from somewhere.**

In a strong evidentiary argument, each claim is supported by one or more claims or axioms further upstream. Similarly, in a valid data analysis project, each intermediate value leading to the final product is either data from an external system of record or is the result of such data passing through a piece of code.

Every piece of data or work product in an analysis tree should be the result of a dependency graph that can be **traced backwards** to examine what combination of code and data it came from or **run forwards** to recreate any artifact of the analysis. To put this in terms of data structures, a proper data pipeline is a directed acyclic graph (DAG).

If you trace any end product far enough upstream, you should eventually end up at one or more scripts that extract raw data from a system of record or a pre-extracted raw dataset if extraction is beyond the scope of the project.

Having a miscellaneous data file in the project folder should be a method of last resort: where the heck did `census_shapefile_v2_NEW.shp` come from anyway? How old is it? How reliable is it? Where do we get a new copy when this one is out of date? How do we even know when it's out of date? If we need to do this because the file can't be downloaded from somewhere, the steps to acquire the file should at least be written down, for example in the `README` for the repo.

Practical tip

There are many tools for and implementations of data pipelines. We discuss some of these options and expand on this idea further in rule 7.

Rule 3: Version control is basic professionalism

"If you don't have source control, you're going to stress out trying to get programmers to work together. Programmers have no way to know what other people did. Mistakes can't be rolled back easily."

— Joel Spolsky, "The Joel Test: 12 Steps to Better Code"

To say that this rule is the consensus among software professionals is a gross understatement. By now, version control is such an ingrained cultural norm on modern software development teams that the lack thereof is widely seen as a serious red flag suggesting the company or team should be avoided.

As data science becomes more collaborative and more ingrained in the engineering operations of organizations, it is increasingly necessary for data teams to use version control systems like git. As we've already seen from decades of software development, those that don't will suffer the consequences. At the same time, there are a few considerations specific to data pipelines which bear discussion.

Data should (mostly) not be kept in source control

First of all, unless computation time is prohibitive, **it is not usually a good practice to store intermediate or cleaned data products**. As discussed above, the whole idea of reproducible pipelines is that everything should be obtainable with clear provenance from the original, raw dataset. The less often this pipeline is actually invoked, the more shrouded it becomes in doubt and mystery. (Consider the related concept from the world of IT operations that data backup recovery plans which are not often tested frequently fail to work.)

In addition to the simple impracticality of storage when data is a certain size (git will start to choke at the order of 100s of megabytes), even if you could check the new data into the repository every day this would only serve to clutter up the commit history and inflate the repository size on disk. The churn in change tracking if data changes were captured alongside code changes would become unbearable, and would discourage people from using the tools.

And it isn't just inconvenient and noisy: **putting data into a code tracker is a conceptual mismatch**. While code changing is a notable historical event, in data science workflows we typically expect the raw data to change quite a bit over time—for example, a daily report pipeline would run on a new or updated data set every day.

If your data is properly stored in a system of record like a relational database, flat files, or any kind of data warehouse, then "versioning" is a separate concern that is properly handled elsewhere (e.g. timestamps of events, unique identifiers, etc). Instead, raw data can be archived and shared using basic storage (e.g. files on a network drive, ordinary hard drive, or an Amazon S3 bucket). For example, in our exploratory projects we have simple scripts that allow a data scientist to sync the whole `data/` directory up to a cloud storage location, or pull it down for local use.

There are a few common exceptions to this rule: (1) when the files are small and rarely change (e.g., a lookup table of country codes), especially (2) with text-formatted data where looking at a "diff" would actually be helpful, and (3) for change management purposes, for example to avoid breaking changes from external dataset updates without a thoughtful and intentional project update.

Practical tip

For more in-depth collaboration and change tracking, teams might consider using one of the more complete data versioning tools like git-lfs, dvc, or dat. Teams with sufficient resources may also consider one of the many cloud data science environments, which typically provide both compute resources, notebook versioning, and dataset management tools.

Source control enables code review

Code review is the most effective tool for maturing data science teams, but adoption of this practice varies widely from team to team. Most effective software teams have a code review process where, in order for code to be accepted into the code base, it is read by another human.

To paraphrase a common maxim, code should primarily be written for humans to read and only secondarily to be run by computers. If your code does not clearly communicate its purpose to collaborators while you are developing it, then that code will be nearly impossible to debug or maintain.

One of the great benefits of source control is that it lets you look at a "diff" that highlights just the code that changed at a certain point in time. Even before others review your code, this helps data scientists spot unintended changes, extra code that was added for debugging, and lurking `TODO`s that did not get done.

Another benefit is that data science code often accretes many small decisions and assumptions. It is hard to track each of these individually and get feedback on the mathematical validity, statistical relevance, and implementation of all of these decisions without code review.

Otherwise, these decisions, which can have an enormous impact, have a tendency to slip through the cracks. Data science is a wide and diverse field and very few data scientists have exactly the same expertise, so code review can catch mistakes that others may have experienced before. Especially when paired with a collaboration tool like pull requests, code review helps highlight each of the newly added analysis decisions in a scope that is manageable for a second set of eyes to check without having to read the entire code base end-to-end.

Finally, code review is one of the best ways to promote sharing knowledge among colleagues. From simply learning helpful tricks to explaining complex statistical logic, code review enables the author and the reviewer to learn from each other through concrete examples.

Practical tip

For teams using Jupyter notebooks, you may have noticed that the tooling is still rough for reviewing and commenting on colleagues' work. We created a Python package that automatically exports Jupyter notebooks to `.py` files upon save. We make it a habit to commit these up-to-date text exports whenever their corresponding notebook changes, and leave pull request comments on line of the text files in any code reviews.

Rule 4: Notebooks are for exploration, source files are for repetition

"The majority of the complaints I hear about notebooks I think come from a misunderstanding of what they're supposed to be. ... Just like with a paper, you can present scientific or mathematical ideas with accompanying visualizations or simulations. ... It's decidedly *not* there for you to type all your code in like an editor and make a huge mess."

— Mali Akmanalp, @makmanalp

Especially for tasks like exploratory data analysis, data scientists have embraced interactive environments such as Jupyter, R Studio, and other literate programming frameworks. As tools for rapid, iterative, and serendipitous explorations, notebooks give data scientists a tight feedback loop showing the immediate results of each change. They are also invaluable as artifacts for communicating and explaining analyses.

But practitioners must also be clear-eyed about their limitations. These tools can prove problematic for reproducing an entire analysis or serving as commonly used components in a larger pipeline.

For anything other than tiny projects, you ordinarily don't want to squeeze the entire data pipeline, multiple experiments, and output generation into one monolithic notebook. When notebooks get that large and unruly, clarity and separation of concerns are often sacrificed in order to pack the entire analysis into one file.

The problem with sacrificing clarity to get "One Notebook To Run Them All" is that **notebooks are almost always the wrong tool for repeated processes**, for several reasons:

1. Having a human manually open up a notebook and run the cells is not automation. (In fact, if it is considered important for a person to be watching the results of this process as it is run then that is a potential red flag for reproducibility.)

2. Even though the popular notebook environments contain rudimentary support for being run as if they were a script, the implementations are hard to test and often introduce complications which go against the grain of common system conventions such as how logging works, where output is directed (e.g. `STDERR` vs `STDOUT`), the return of error codes and what process fails on uncaught exceptions, working directories, and so forth.

3. Since notebooks are challenging objects for source control (e.g., diffs of the raw files are often not human-readable and merging is nearly impossible), we recommend not collaborating directly with others by editing the same notebooks. In addition to merging conflicts and other headaches, collaborating on notebooks significantly reduces the effectiveness of code review, which is critical for data projects.

Practical tip

As an aside, for collaborating on a project with notebooks we highly recommend that teams adopt a naming convention that shows the "owner" and gives a sense of the order the analysis was done in. For example, we use the format `<step>-<initials>-<description>.ipynb` (e.g., `3.1-pjb-visualize-distributions.ipynb`) where 'step' is a loose idea of where in the end-to-end workflow this notebook falls.

So how do we harness the exploratory power of notebooks without giving up on reliable and automated pipelines?

The answer is by **continuously extracting common building blocks out of the notebook and into source files that can be centrally tracked and used from any other notebook.** Standardizing such code—e.g. utility functions and extract-transform-load (ETL) steps—in a central location does a number of amazing things for a project:

- Separates the multiple concerns into logical units, so that the data layer (ETL code) is not intermingled with the modeling and experimentation (where notebooks shine), and that neither of those are deeply entangled with the output layer (final results of the workflow, whether that means predictions, reports, graphics, or values sent to a database).

- Prevents duplication of code across multiple notebooks—which is a common way for errors to creep in—and allows the pipeline code to be called *either from notebooks or programmatically* from other automated scripts. When writing all our code in notebooks, we are stuck with only notebooks. When we split reusable notebook code into modules, changes to the commonly used code propagate automatically across all of our notebooks and wherever else the code is used.

- Allows *pieces of the pipeline* to run without running the entire pipeline, so that changing a few hyperparameters and refitting a model doesn't force the choice between (a) sitting through another long-running data extraction task or (b) choosing some miscellaneous cell in the notebook to back up to while hoping that no other state from later in the program has muddied the waters.

- Enables us write tests that verify both functionality and correctness for the most important parts of an exploratory analysis.
- Readily shows code changes across commits in version control, which also means that significant changes to how data gets extracted and cleaned are obvious and centrally updated. This in turn allows code review tools to work as designed on meaningful diffs in plain text source files.

This kind of refactoring is a powerful technique that sounds like a lot of work but actually speeds up analysis instead of slowing it down.

Once a team starts to embrace refactoring, it becomes second nature to write utility code in commonly accessible modules, maybe write a couple of quick tests to make sure it's correct, and then import it into whichever notebooks are in use. This process pays dividends in terms of code quality, reliability, and productionizability.

One caveat: Netflix has famously taken things in the opposite direction and decided to centralize their data science pipeline around running notebooks. That's fine, but now's a good time to remember that most organizations are not Netflix and probably don't have the same resources to throw at the tooling and work practices necessary to make this work well.

Practical tip

When code should be moved out of the notebook is a matter of judgment, taste, and pragmatism. We think it helps to weigh the special advantages of notebooks over text files: *demonstration and experimentation*.

If a piece of code is the focus of the work at hand or solely designed for this unit of analysis, it may make sense to leave it in; if it is necessary building block but not worth showing especially, it may be better to move to a separate file that can be tracked in version control and tested as discussed in the next section.

Rule 5: Tests and sanity checks prevent catastrophes

"Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse."

— Michael Feathers, Working Effectively with Legacy Code

Data scientists ought to take note of how transformational the professional norm of writing tests has been in modern software development. At the same time, testing data science code has additional challenges compared to testing ordinary software.

Common difficulties include:

- Tests on large datasets make for long-running test suites
- We actually expect data to change over time, so downstream values fluctuate
- We often use randomness on purpose (e.g. to sample, or in model fitting), meaning it is hard to assert whether changes are meaningful or due to expected variation
- Visualizations are challenging to test
- Notebook environments do not have the same level of tooling for test discovery and running

These factors all make testing harder, and we will not presume to argue that data science projects should have the sort of extensive test coverage found in production software projects. This rule is an excellent example of "a little goes a long way" rather than "all-or-nothing."

So what data science code should be tested?

Instead of the sort of large integration tests commonly written for user-facing software, we believe that the right tool for probing data correctness in most exploratory code is the sanity check and smoke test" (see rule 6). This is especially helpful when the tested examples exercise possible edge cases such as null values, zeros, shape mismatches, or whatever is likely to cause trouble. In particular, tests that operate on "toy examples" with tiny amounts of data or extremely small arrays make it clear to a human reader what is being tested and what values are expected, and aid in debugging if errors are detected.

Meanwhile, the right tool for verifying processing or mathematical code is the unit test focused on a specific operation being correct in isolation. This typically involves taking a sensible number of expected input-output pairs—perhaps taken from a reference, calculated using an alternative package, or worked out by hand—and demonstrating that the new code produces the expected output.

We do not endorse any metric like code coverage percentage or number of tests—at best these metrics provide only a false sense of security. We strongly recommend: (1) writing tests for any code that is refactored out of notebooks, and (2) writing tests with sample data to confirm that the logic works as expected.

Practical tip

As noted above, writing and running tests within notebooks is not a seamless experience by default. There are packages that attempt to make this easier, but in our view it actually bolsters the advice in rule 4. it is a good practice to extract non-exploration code to source files which can be tested in the customary manner.

Rule 6: Fail loudly, fail quickly

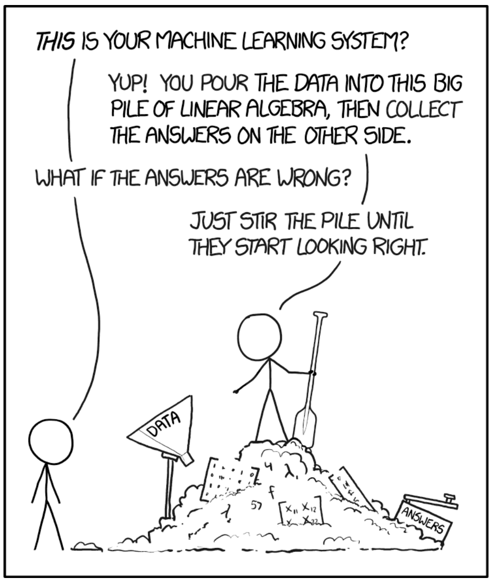
"This is a problem that occurs more for machine learning systems than for other kinds of systems. Suppose that a particular table that is being joined is no longer being updated. The machine learning system will adjust, and behavior will continue to be reasonably good, decaying gradually. Sometimes tables are found that were months out of date, and a simple refresh improved performance more than any other launch that quarter!"

— Martin Zinkevich, "Rules of Machine Learning"

Machine learning is great because we can often make good predictions and inferences despite the presence of noisy data.

Machine learning is dangerous because sometimes we unwittingly cause that noise through buggy code and still get reasonable-looking outputs.

This robustness to human error is one of the most dangerous properties of probabilistic models. In other words, **our models can often appear to give reasonable predictions despite serious programming errors or data quality issues.**


THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

XKCD #1838, "Machine Learning"

Data scientists can prevent many errors like this with the application of some defensive programming principles designed to stop invalid or unexpected states from proceeding to an unexpected and potentially flawed result. In their internal R training the United States Geological Survey lays out the principles of defensive programming as follows, arguing that errors should be:

> **Conspicuous** — the worst failure is a silent one. **Fast** — if a function is going to fail eventually, it might as well fail right now. **Informative** — provide messages and context that help the user understand and/or correct the problem.

For example, let's imagine we have a model-fitting pipeline where one of the features in the clean data is the mean value of another data column in the raw data. Let us also assume that the column in the raw data is not generally expected to contain missing values and we write our data pipeline accordingly.

Consider that the `mean()` function in many statistical packages ignores nulls without complaint; one commonly seen and potentially devastating failure mode is if the raw data column silently starts to have non-random null values—perhaps from a damaged sensor or a buggy code change in the database or application layers. Without at any point failing to be a valid floating point number, the new mean value of this nullable column could be strongly biased and lead to bad predictions with which we will make potentially costly business decisions.

That is a *terrible* property of systems that are trying to generate meaningful outputs!

An easy way around this whole category of errors is simply to make our assumptions concrete and enforce them at runtime, and to bail out immediately and loudly if the assumptions are violated. In the example described, we might check the input column before applying the transformation, immediately log that the non-null assumption was violated for the column in question, and then halt the processing script with a failure exit status.

Using the `mean()` example above, if you fully intend to ignore null values in a column, that is of course fine. But the pipeline should not be more permissive than necessary, so teams should set a norm thinking explicitly about where to be permissive and where to be strict. Decorating that processing step with an assertion that values can be floating point values or null is yet another way to "point and call" sources of possible error in an extremely intentional way.

Practical tip

Defensive assumption-checking is such a common and recommended pattern that packages like bulwark (for Python) and assertthat (for R) have been created to help, establishing shortcuts to enforce assertions about scalars, vectors, and data frames. Code reviews can be an excellent time for data scientists to step back and proactively anticipate sources of error and document assumptions directly in the code under consideration such that any violation of assumptions will directly trigger human intervention at run time.

Rule 7: Project runs are fully automated from raw data to final outputs

"People can lull themselves into skipping steps even when they remember them. In complex processes, after all, certain steps don't always matter. ... 'This has never been a problem before,' people say. Until one day it is."

— Atul Gawande, The Checklist Manifesto

Decentralization and anonymity are desirable in cryptocurrencies, not data pipelines. It should be staggeringly obvious to anybody looking at a project how to go from raw data to finished products, and **they should be able to initiate this process with one interaction**.

Instructions in a `README` file are simply not enough, and we have run across plenty of well meaning `README`s that accidentally miss crucial steps, are too vague, or have not been updated as other pieces of the project change. When data pipelines are not automated, they are not reproducible.

Use a build tool

Here's a good rule of thumb: **keep your pipeline working and run it every time you make a non-trivial change**. In modern software development, we can look to the practice of Continuous Integration and Continuous Delivery (CI/CD). The fundamental idea of CI/CD is that the mainline branch of the project is **always in a runnable state** and in fact **has its tests run every time a change is committed** to version control.

In an ordinary software project, this usually means that a developer can deterministically build the same end product by running a single command on a given version of the codebase. In the development lifecycle, this sort of reproducibility and centralization allows a suite of automated tests to be run every time changes are committed. The new version can then be deployed to production if successful, which is the "delivery" part of Continuous Delivery.

For a data project, this means that a person trying to reproduce the results of a project should be able to run a "default" pipeline without typing out or understanding all of the various knobs and settings available. (Here "default" might be something context-specific like the specific parameters used to generate the paper, report, or predictions. In other cases, it may just mean falling back to some sensible defaults.)

Using a proper build tool, this can often be as intuitive as instructing the user to run a single command from the project directory.

Practical tip

There are many heavy-duty DAG management tools like Apache Airflow that are geared toward pipeline orchestration in production environments. For research projects, our team prefers good old GNU Make, a ubiquitous and long-lived open source tool for managing build steps with interdependencies. Despite its age, a number of data people still swear by it primarily because it follows of the Unix philosophy of doing one thing well. Following the Make documentation can be a bit daunting, but teams can get far using a tiny subset of features starting with "how does file A get turned into file B?" Here is a good overview to get started.

Make the environment reproducible

Setting up a build tool is a necessary first step, but it's not sufficient to make a project easily reproducible. A colleague or client also needs the same interpreter and the same libraries with identical versions to ensure matching results. Most data science environments provide standard ways to document package dependencies.

For Python projects, this means at a minimum including a `requirements.txt` for pip (or the Anaconda equivalent). For R, tools like packrat help keep dependencies reproducible. If you have more complex requirements for recreating your environment, consider a containerization or virtual machine-based approach such as Docker or Vagrant. Both of these tools use text-based formats which describe how to create a virtual machine with the requirements you need and can easily be tracked in source control too.

Practical tip

Our rule of thumb is that analysis code should pin the specific versions of libraries that reproduce the results exactly. Library code may depend on at least a specific version (e.g., `>=1.1.0`) since it can be used across data science projects. There is a tradeoff here: if you pin everything, it might get more difficult to build the environment over time, and future users might be trapped in an obsolete environment.

Make the randomness reproducible

Finally, we strongly advise setting a random seed in a central location when possible. In many data science applications (e.g. sampling methods for model training, stochastic optimization, simulation) random values are used as part of the analysis. Reproducing an analysis using probabilistic methods means the pseudorandom number generators must be initialized with a known state to induce the same values each time.

In our projects, we typically prearrange a centralized random seed that is set wherever random numbers are use, but we allow this value to be overridden or unset by environment variables or command line arguments to the build tool (per rule 9).

Practical tip

Managing randomness is a danger zone where projects often introduce irreprodiciblity. For example, one common misstep is to forget that libraries don't necessarily obey each other's random seeds; `random` in the Python standard library, `numpy.random`, `torch.rand`, `jax.random` (and so on) each use their own seeding mechanisms. Relatedly, we prefer using explicit random number generator objects where we set the seed in one location and track its use—for example, explicitly creating and passing a `numpy.RandomState` object wherever randomness is used) rather than simply setting the seed at the library level (e.g. a one-off call to `numpy.seed` somewhere).

Rule 8: Important parameters are extracted and centralized

"Explicit is better than implicit."

— Tim Peters, "The Zen of Python"

In human language, the phrase *magic numbers* might sound mysterious and exciting, but in modern software development the phrase is a disapproving reference to the illicit practice of sprinkling throughout the codebase critically important values which affect the program's behavior.

The difference between magic numbers scattered all over a codebase as opposed to organized constants within source files generalizes to a much broader question of whether a project's configuration is centralized and legible or whether important settings are carelessly dispersed through multiple layers of abstraction.

The problem of parameters

Data projects often end up with an excessive number of ways to parameterize scripts and functions. For example, we have seen instances where there are no fewer than six places where important configuration values such as model parameters can live, and can only be run in a fully specified way using an intricate combination of environment variables, `Make` variables, command-line arguments, constants defined in a central location, constants defined at the top of files, and default keyword arguments in user-defined functions.

Forking paths and exponential complexity make highly parameterized code difficult to understand or trust, and worse yet, mean that **you can't reproduce a project's results from one run to another without exactly mirroring the developer's environment, run command, and configuration files at one instant in time**!

Consider a model training pipeline that cleans some raw sales data, fits several ensemble models, and then outputs predictions for next quarter. All of the model parameters multiply possibilities with other "meta" choices like train/test ratio, cross validation parameters, and ensemble voting parameters to make a combinatorially overwhelming universe of possible inputs for a single run of this project.

Let's look at the function signature for a single, arbitrarily chosen, widely used ensemble model:

```
class xgboost.XGBRegressor(max_depth=3, learning_rate=0.1, n_estimators=100,
 ↪  silent=True, objective='reg:linear', booster='gbtree', n_jobs=1,
 ↪  nthread=None, gamma=0, min_child_weight=1, max_delta_step=0, subsample=1,
 ↪  colsample_bytree=1, colsample_bylevel=1, reg_alpha=0, reg_lambda=1,
 ↪  scale_pos_weight=1, base_score=0.5, random_state=0, seed=None,
 ↪  missing=None, **kwargs)
```

This is not an unusually parameterized model, but already there are more than ten choices which can affect model performance. So assuming we don't always use all of the default values, **where should these choices live**?

The naive strategy is to make all such decisions immediately at points in the code where values have to be passed to functions. Trying different configurations of parameters then involves finding all the places where they need to be changed.

For the very earliest phases of exploratory analysis this might be fine. But once a project turns the corner from Messing Around™ to Serious Business™ it often makes more sense to extract the configuration from the code.

"But wait," you object, "my exploratory process needs to be fluid and dynamic! I don't have time to separate every little thing into a configuration file!"

In our view, that's exactly where the upfront effort in extracting configuration really starts to pay off, because making changes is not the only thing we care about: we also want to know how our changes affect outcomes.

Pulling it all together

The pinnacle of pipeline elegance is running another experiment just by changing some settings in a central configuration file, getting a cup of coffee, and then comparing the most recent log to past experiments. This has the added benefit of documenting the changes with the output so you don't have to just remember it, and your colleagues don't need to do an archaeology project to figure out what you were doing.

Using the example of sales forecasting mentioned above, let's say we are doing an analysis where we intend to fit three completely different models, and for each model we wish to (at the very least) test out various settings and hyperparameters to improve performance, not to mention any other pieces that can change (e.g. feature extraction and manipulation, grid search, cross validation, train/test splits, etc.)

If this exploration is done by changing hard-coded steps, data scientists often default to using one notebook for each model then copying and pasting important setup code. Even worse, expedience often tempts data scientists to reuse a notebook for many different models and parameters by changing them by hand one setting at a time and eyeballing the results. If done that way, in order to go back and examine a specific permutation that produced a certain output you must collate all of the corresponding parameters by looking at the version history of the code and then make sure each file is in exactly the right state for that run.

For the example given above, you could imagine an authoritative `config.yml` file with the important inputs centralized:

```yaml
n_threads: 4
random_seed: 42
train_ratio: 0.5
log_level: debug
features:
  use_log_scale: true
  n_principal_components: 4
models:
  xgboost:
    max_depth: [2, 5, 10]
    n_estimators: [50, 100, 150, 200]
  random_forest:
    criterion: ["gini", "entropy"]
ensembling:
  voting: ["soft", "hard"]
```

Doing so lets colleagues feel confident that most if not all of the relevant choices and settings are together in one location, explicitly laid out per experiment *in a format easily tracked in version control*.

Rule 9: Project runs are verbose by default and result in tangible artifacts

Just as magic numbers and unseparated configuration described above make it difficult to reason about the entire program, failing to produce and capture useful output during data pipeline runs makes it painful to figure out where results came from in the future. When questions arise or when it comes time to continue work on the model, it prevents us from easily looking back or picking up where the last data scientist left off if configuration and run history are not tracked along with pipeline outputs (e.g. visualizations, predictions, or fitted models),

Consider the sales forecasting example in rule 8 where a declarative settings file lays out the configuration for all of the modeling choices at once. This series of experiments can ideally be run from a single interaction (rule 7 resulting in a timestamped log file—an *artifact*—showing what happened and when, all stored in version control for posterity.

For example, a run of the experiment codified in the configuration file above might result in a `log_2022-11-01_14-10-02.txt` like this:

```
2022-11-01 14:10:02 INFO starting run on branch master (HEAD @ 37c02ab)
2022-11-01 14:10:02 DEBUG set random seed of 42
2022-11-01 14:10:02 DEBUG reading config file
2022-11-01 14:10:02 DEBUG … settings: {"n_threads": 4, <...snip...>}
2022-11-01 14:10:03 INFO reading in and merging data files
2022-11-01 14:10:39 INFO finished loading data: 2,835,824 rows
2022-11-01 14:10:40 WARNING … dropped 126,664 rows where ID was duplicated (4.47%)
2022-11-01 14:10:41 WARNING … dropped 2,706 rows where column `total` was null (0.09%)
2022-11-01 14:10:41 INFO creating train/test split
2022-11-01 14:10:41 INFO … train: 0.5 [1,353,227 rows]
2022-11-01 14:10:41 INFO … test: 0.5 [1,353,227 rows]
2022-11-01 14:10:42 INFO starting grid search cross validation ...
2022-11-01 14:21:01 DEBUG … 30/120
2022-11-01 14:31:46 DEBUG … 60/120
2022-11-01 14:42:31 DEBUG … 90/120
2022-11-01 14:53:03 DEBUG … 120/120
2022-11-01 14:53:03 INFO finished cross validation, writing best parameters
2022-11-01 14:53:03 DEBUG … runs/2022-11-01_14-10-02/parameters/xgboost.yml
2022-11-01 14:53:03 DEBUG … runs/2022-11-01_14-10-02/parameters/random_forest.yml
2022-11-01 14:53:03 DEBUG … runs/2022-11-01_14-10-02/parameters/adaboost.yml
2022-11-01 14:53:03 INFO training voting classifier on ensemble of 3 best models…
2022-11-01 14:53:19 INFO making predictions
2022-11-01 14:53:22 INFO writing results to runs/2022-11-01_14-10-02/results.yml
2022-11-01 14:53:22 DEBUG … results: precision=0.9624 recall=0.9388 f1=0.9514
2022-11-01 14:53:23 INFO writing predictions to runs/2022-11-01_14-10-02/predictions.csv
```

In this notional log file, we can immediately see some important history about the model fitting, such as how many rows were dropped in various filtering steps, what the train/test split was, how long it took to run, what the performance was of entire ensemble, and what the best parameters were from each model.

Not only are we storing a narrative of the experiment, but many ephemeral values that would otherwise be lost are permanently captured as *artifacts* at that point in time—for example, the best parameters found for each model are stored as computer-readable serialized files like JSON or YAML, and the performance metrics we care about captured in an actual file that could be committed to version control and compared in the future.

Imagine that we decide to introduce a new feature into the training data, and want to see what impact that has on overall performance as well as processing time. Instead of making the change and eyeballing the results or trying to remember past configurations, we can simply look at the result data files from any number of past experiments.

When data scientists and their colleagues can rapidly iterate through experimental settings and retry the whole trusted pipeline with a single command, teams can worry less about missing settings or performance backsliding from miscellaneous changes, and instead focus more time on the actual modeling subject matter and business case.

Particularly in business settings where results are going to be used to make decisions, we have found that the ability to audit *how* results came about—and therefore the ease of detecting and fixing any downstream problems—is well worth the upfront engineering investment for any analyses more serious than a throwaway exploration.

Rule 10: Start with the simplest possible end-to-end pipeline

"A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system."

— Brian Kernighan and John Gall, Systemantics: How Systems Really Work and How They Fail (1975)

Simplicity is an often lauded virtue in technical work but is seldom explicitly described. Here we attempt to distill three aspects of what simplicity really means in data science projects, where the temptation to introduce the complexity of a state-of-the-art model is ever present.

Start with proper form, fill in proper function

> "Suppose you have something concrete that you can point to, and say 'It's going to look like this.' If you can show it to trusted users to get their reactions, you may learn that your design was on target. More often than not, your prototype will come back shot full of holes. That's OK, though. You have gained valuable design information. ... For every correct design, there are hundreds of incorrect ones. By knocking out a few of the bad ones early, you begin a process of elimination that invariably brings you closer to a quality finished product."
>
> — Mike Gancarz, *Linux and The Unix Philosophy 2E*

It often makes sense to work a data science problem through roughly from raw data all the way to the form of the finished product before going back and trying to improve all of the pieces. If time were not a factor, it might be nice to start at the beginning and end at the end, polishing each conjecture and experiment to a perfect diamond of analysis before moving on.

However, most teams are not operating without time constraints so it is often better to get the whole pipeline glued together even if some parts are painfully basic or even faked and left as stubs. Then, even if very few features are cleaned from the raw data, even if we use a naive model that gets poor accuracy, and even if the outputs are laughably rough, at least we will have assembled an automated pipeline that can then be worked on piece by piece.

There can be a powerful peace of mind that comes from being able to say "we still need to improve X, replace Y, tweak Z, and retrain using all the data but you can see the final deliverable format if you run the pipeline now."

Start with basic tools and simple models

> "Always implement things when you actually need them, never when you just foresee that you need them"
>
> — Brian Kernighan, *The Elements of Programming Style (1978)*

When trying to identify objects in images or video it may be necessary to use a complicated and computationally expensive deep learning model. However, when making numeric predictions from a low-dimensional tabular data set the potential improvement in accuracy may not justify the added complexity and decreased interpretability compared to plain old regression models or decision trees.

The decision to take on additional complexity in a data science project should not be made lightly or by default, but delayed until the last responsible moment when it becomes clear that the tradeoff is likely to pay dividends in the form of increased prediction accuracy or clearer inference.

This logic extends far beyond the choice of statistical models to virtually every engineering decision in a project.

For example, as discussed above there are numerous powerful libraries for orchestrating the pipeline from raw data to clean data to trained models to new predictions. But before selecting a large customizable framework, it is worth considering whether an old and boring (read: well understood) tool like `Make` will do the job.

As another example, it can be tempting to try to perfectly handle every quirk in a dataset. (What if a value is missing in one of the rows? What if one of the categorical rows has an unexpected value?) But often it makes sense to start building with the "happy path" in mind.

Often this means purposefully declining to handle edge cases, for example by *failing fast* on bad inputs (per rule 6) or by *loudly filtering* unhandled cases (per rule 9). Always these decisions can be responsibly documented either as an open issue on the project tracker or at least a line in the `README` file.

Practical tip

For example, let's say the raw data has a string column with values like `"12.10 USD"` representing amounts of money. If we want to create a feature column with the amount, it is tempting to shave this yak by doing something clever like creating a function that can take any currency abbreviation and render a custom struct with an enumerated data type field representing which currency is in use and a decimal field representing the amount converted to USD.

However, if you are writing software to analyze stocks on the NYSE, you'll only ever see `"USD"` and this would be (1) a waste of time, (2) harder for future data scientists to fully comprehend, and (3) a great way to create more real estate for bugs to live in. Instead we can get away with just splitting the field on the space character, asserting that the second value is `"USD"` as expected (see rule 5 on defensive programming), and parsing the first value only

You are not required to use all of the data all of the time

A friendly reminder for keeping the process of discovery fast and iterative: you probably don't have to process all 50 exabytes of data when you're still trying things out in an exploratory mode.

Perhaps the central reason that notebooks are so popular in data science work is that a rapid, iterative feedback loop makes all the difference when trying to produce quantitative insights. But many data scientists abandon this fast feedback loop when working with large data sets because it somehow feels wrong not to use all available data.

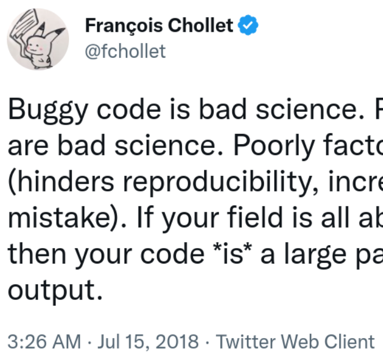
It is an unfortunate truth that many of our most powerful models and preprocessing techniques scale poorly with the size of the data in terms of speed, memory, or disk usage (and often all three). Luckily, *exploratory analysis* and *scaling an analysis* are two different chores, and can often be tackled separately as long as the bigger picture is kept in mind.

For example, even if an analysis will end up needing to use all of the data to generate the final reports or predictions, we can often keep the exploratory feedback loop fast by working with representative samples. And just because we are going to train a certain model using a Spark cluster doesn't mean we have to submit a job on all of the data every time we want to make some tweaks.

In fact, sometimes it makes sense to save all of the parallelization for later, working with smaller chunks of the data using the same model but in RStudio or a Jupyter notebook before productionizing it into a job that will run on a cluster. Once we feel confident about the modeling assumptions and data pipeline, they can then be translated into the appropriate idioms for parallelization.

We aren't trying to minimize the difficulty of choosing how to sample, which is a deep topic. Sometimes even a naive sample is good enough for the task at hand, such as getting all the data cleaning code in order or making sure that a preprocessing step results in useful features.

Conclusion


François Chollet
@fchollet

Buggy code is bad science. Poorly tuned benchmarks are bad science. Poorly factored code is bad science (hinders reproducibility, increases chances of a mistake). If your field is all about empirical validation, then your code *is* a large part of your scientific output.

3:26 AM · Jul 15, 2018 · Twitter Web Client

Perhaps of most interest to the working data scientist, **it is simply more fun and less stressful to work on a tidy project** than to feel like the work is turning into a big ball of mud. (We have seen and created both kinds of projects—we suspect most others have.)

Like working at a messy desk littered with stacks of papers and debris, it can be demotivating to work on a project that feels disorganized or out of control. Data scientists who spend too long working on projects in this condition tend to look for new jobs.

But enjoyment and retention are not the only reasons for us to step up our game as a field.

Without being alarmist, it has become clear that quantitative reproducibility is a problem in both science and industry. Ultimately, **we want to see our profession produce sound results and earn the trust of others who depend on our work.** For that to happen, we must try to be organized and methodical enough to catch possible sources of error early and counteract our biases and shortcomings.

Many of these rules have proved helpful in our work, and we hope that they prove helpful in yours.

Thank you for reading!

Thoughts? Get in touch!

We want this to be a living document, so we hope you will share your feedback, anecdotes, and recommendations with us.

Appendix A — Putting it into practice: software lessons learned the hard way

"In a competitive market for development work and staff, paying people to learn from mistakes that have already been made by many others is an unaffordable luxury; an engineering approach, derived from evidence, is a lot more cost-effective than craft development."

— Derek Jones, Evidence based software engineering (2020)

It is beyond the scope of this paper to list every lesson learned about software quality, but here we'll quickly summarize in no particular order selected principles that are particularly relevant to the data science practitioner.

Version control is a must. Learning a tool like git is admittedly a painful experience. That said, the basics can be picked up in an afternoon and there is simply no excuse for professional data scientists not to use a modern version control tool. (Adopted directly as rule 3

Keep it simple, stupid (KISS). This engineering maxim comes to software by way of the famous Lockheed Skunkworks in the 70s. Simple solutions are easier to reason about, easier to debug, and easier for new collaborators to understand. Like a corporation taking on debt to gain liquidity, it is sometimes reasonable to incur complexity when it serves the ultimate goal of the project, but only after careful consideration. (Relevant to all rules, but primarily rule 10

Separation of concerns. Experienced developers spend a lot of time thinking about how to minimize the amount each component of a program needs to "know" about other pieces. For example, it is a truth universally acknowledged that the presentation layer of an application shouldn't need to "know" the details of the lower level business or data layers. Also referred to as modular design or loose coupling, this principle also tends to make code easier to understand since a function in one file will not be tangled up in every other piece of the codebase. (Relevant to rules 1, 4, 9, and 10.)

Separate configuration from code. Relatedly, to understand a program one needs to know what decisions have been made for all of the meaningful settings and parameters. When these settings are dispersed throughout the entire codebase, any person trying to understand the logical flow must "load" all of the code that can possibly affect program setup and mentally "run" it to figure out what is actually happening. Important settings can be centralized and codified to make the implications of their values on outputs far more obvious. (Relevant primarily to rule 9.)

You aren't gonna need it (YAGNI). Programmers tend to have an intense desire to write data abstractions and generalizable functions even on their first encounter with a problem. YAGNI tells us that this is premature optimization at best and dangerous folly at worst. Abstraction is both powerful and costly—software developers have realized through hard-won experience that over-engineering and early abstraction can be crushing wastes of time. The takeaway is to start with a concrete case and only generalize when needed. Software developers refer to people who start projects with overly grand, generalized visions as "architecture astronauts." (Relevant primarily to rule 10.)

Premature optimization is the root of all evil. The rule of thumb is to "make it work, make it right, make it fast." Related to YAGNI, this lesson emphasizes the importance of focusing on getting the job done first before trying to cover all possible edge cases and before trying to speed the program up. For data work, this often comes to the fore when early project stages get bogged down trying to deal with massive data sets and clusters before the problem and the plan are really understood. (Relevant primarily to rule 10.)

Don't repeat yourself (DRY). Having a few instances of similar code is not a problem. But when functionality starts to get copied and pasted all over a program, that should be cause for concern because modifications to the functionality need to be repeated and those locations remembered. In examining how this applies to data science, we will talk about the importance of refactoring notebook code to "real" modules instead of duplicating code in many locations. (Relevant to rules 4 and 9.)

Composability. The Art of Unix Programming lays out a set of ideas referred to as the Unix Philosophy; these principles dictate that we "should write programs that can communicate easily with other programs. This rule aims to allow developers to break down projects into small, simple programs rather than overly complex monolithic programs." For example, instead of having a massive graphing function that reads in data, reshapes the data, and then plots it, we should prefer separate functions that read in the data, assemble the data to be presented, and are then composed together with a function that graphs the result when desired. This way, the data import and reshaping can be used in other parts of the program without resulting in an unwanted graph—not coincidentally also respecting the related principles of separation of concerns and DRY. (Relevant to rules 4, 7, 9, and 10.)

Test the critical bits. Tests can't prove that code is bug free, but they can prevent some of the most common categories of errors, increase confidence about program correctness, and help already-fixed bugs from creeping back in. Even very simple and basic tests can sometimes act as a strong bulwark against catastrophic bugs. Data work is much more exploratory than backend software engineering, for example, so we don't go so far as to recommend an approach like test driven development (TDD). That said, we do discuss why simple tests and sanity checks are crucially important for data projects. (Adopted directly as rule 5.)

Fail fast and loudly. In software development, few things are more alarming than a program that has fallen into an invalid logical state but continues to do stuff with production data. This idea has several implications for software design, encouraging a defensive (some might say paranoid) way of thinking about error cases and a focus on bailing out early if unanticipated errors happen. It also has massive corollaries in the world of data science that are discussed at length in this document. (Adopted directly as rule 6.)

Appendix B — Further readings on correctness and reproducibility

1. Helmreich, Robert L., Ashleigh C. Merritt, and John A. Wilhelm. 1999. "The Evolution of Crew Resource Management Training in Commercial Aviation." The International Journal of Aviation Psychology 9 (1): 19–32. https://doi.org/10.1207/s15327108ijap0901_2.

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

3. Knuth, Donald E. 1974. "Computer Programming As an Art." Commun. ACM 17 (12): 667–673. https://doi.org/10.1145/361604.361612.

4. Netflix. "Beyond Interactive: Notebook Innovation at Netflix." Netflix TechBlog (blog). August 16, 2018. https://medium.com/netflix-techblog/notebook-innovation-591ee3221233.

5. Noble, William Stafford. 2009. "A Quick Guide to Organizing Computational Biology Projects." PLOS Computational Biology 5 (7): e1000424. https://doi.org/10.1371/journal.pcbi.1000424.

6. Randall, D., and Welser, C. "The Irreproducibility Crisis of Modern Science: Causes, Consequences, and the Road to Reform," National Association of Scholars. 2018.

7. Sculley, D., Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. "Machine Learning: The High Interest Credit Card of Technical Debt." https://research.google.com/pubs/pub43146.html.

8. Stripe. "Reproducible Research: Stripe's Approach to Data Science." https://stripe.com/blog/reproducible-research.

9. U.S. Geological Survey. "Defensive Programming - R Curriculum." https://owi.usgs.gov/R/training-curriculum/r-package-dev/defense/.

10. Wilson, Greg, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K. Teal. 2016. "Good Enough Practices in Scientific Computing." ArXiv:1609.00037 [Cs], August. https://arxiv.org/abs/1609.00037.

DRIVENDATA

Building data solutions for problems that matter

DriveNData brings cutting-edge practices in data science and crowdsourcing to some of the world's biggest social challenges and the mission-driven organizations taking them on.

Think we can help? Get in touch!

Contact us

Website	www.drivendata.co
Email	hello@drivendata.co
GitHub	@drivendataorg
Twitter	@drivendataorg
LinkedIn	@drivendata-org

Cover image adapted from:
The Alchemist, after 1558,
The Metropolitan Museum of Art