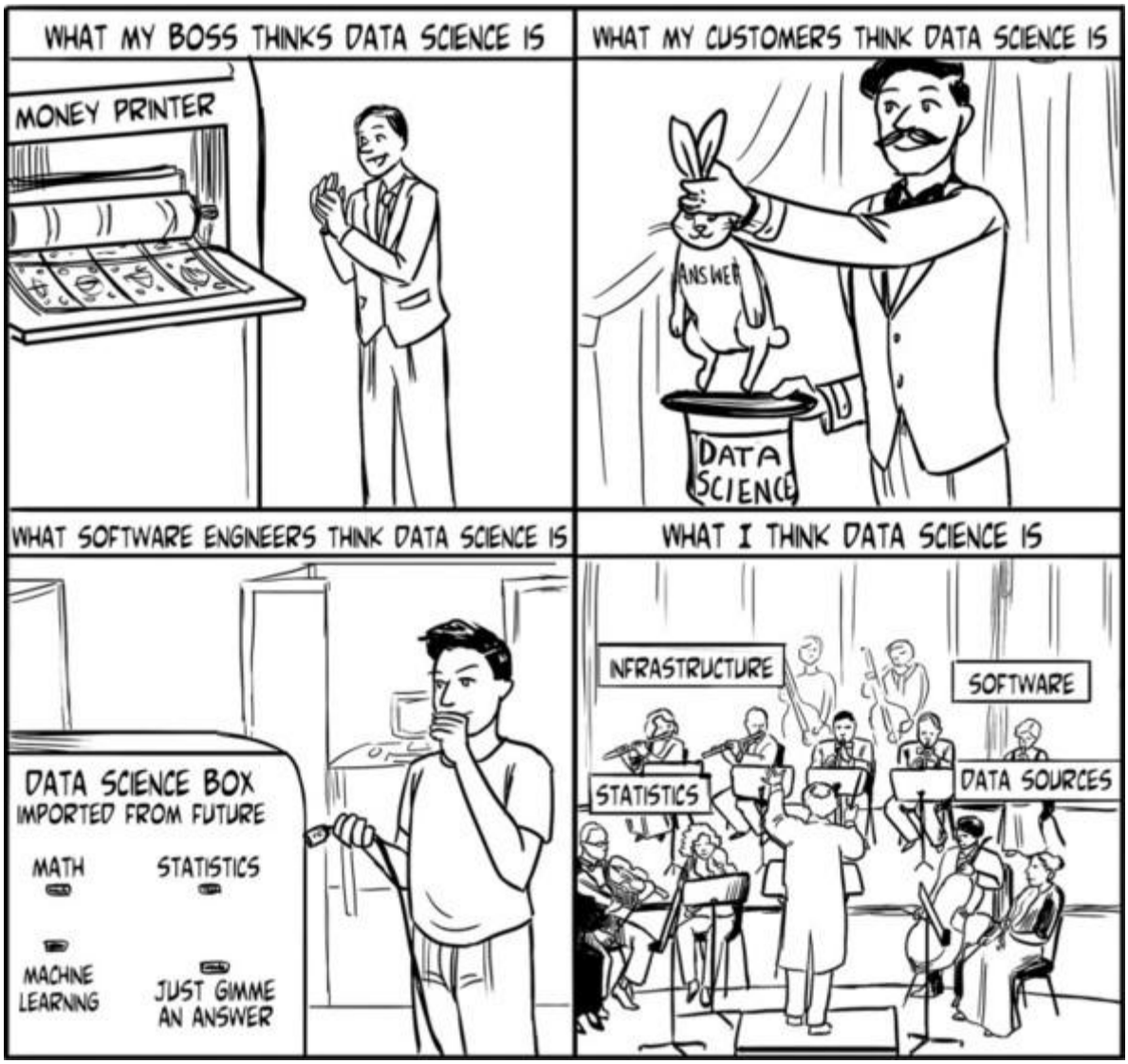
Chapter 1. Philosophies of data science

This chapter covers

- The role of a data scientist and how it's different from that of a software developer
- The greatest asset of a data scientist, awareness, particularly in the presence of significant uncertainties
- Prerequisites for reading this book: basic knowledge of software development and statistics
- Setting priorities for a project while keeping the big picture in mind
- Best practices: tips that can make life easier during a project

In the following pages, I introduce data science as a set of processes and concepts that act as a guide for making progress and decisions within a data-centric project. This contrasts with the view of data science as a set of statistical and software tools and the knowledge to use them, which in my experience is the far more popular perspective taken in conversations and texts on data science (see figure 1.1 for a humorous take on perspectives of data science). I don't mean to say that these two perspectives contradict each other; they're complementary. But to neglect one in favor of the other would be foolish, and so in this book I address the less-discussed side: process, both in practice and in thought.

Figure 1.1. Some stereotypical perspectives on data science



To compare with carpentry, knowing how to use hammers, drills, and saws isn't the same as knowing how to build a chair. Likewise, if you know the process of building a chair, that doesn't mean you're any good with the hammers, drills, and saws that might be used in the process. To build a good chair, you have to know how to use the tools as well as what, specifically, to do with them, step by step. Throughout this book, I try to discuss tools enough to establish an understanding of how they work, but I focus far more on when they should be used and how and why. I perpetually ask and answer the question: what should be done next?

In this chapter, using relatively high-level descriptions and examples, I discuss how the thought processes of a data scientist can be more important than the specific tools used and how certain concepts pervade nearly all aspects of work in data science.

1.1. DATA SCIENCE AND THIS BOOK

The origins of data science as a field of study or vocational pursuit lie somewhere between statistics and software development. Statistics can be thought of as the schematic drawing and software as the machine. Data flows through both, either conceptually or actually, and perhaps it was only in recent years that practitioners began to give data top billing, though data science owes much to any number of older fields that combine statistics and software, such as operations research, analytics, and decision science.

In addition to statistics and software, many folks say that data science has a third major component, which is something along the lines of subject matter expertise or domain knowledge. Although it certainly is important to understand a problem before you try to solve it, a good data scientist can switch domains and begin contributing relatively soon. Just as a good accountant can quickly learn the financial nuances of a new industry, and a good engineer can pick up the specifics of designing various types of products, a good data scientist can switch to a completely new domain and begin to contribute within a short time. That is not to say that domain knowledge has little value, but compared to software development and statistics, domain-specific knowledge usually takes the least time to learn well enough to help solve problems involving data. It's also the one interchangeable component of the three. If you can do data science, you can walk into a planning meeting for a brand-new data-centric project, and almost everyone else in the room will have the domain knowledge you need, whereas almost no one else will have the skills to write good analytic software that works.

Throughout this book—perhaps you've noticed already—I choose to use the term *data-centric* instead of the more popular *data-driven* when describing software, projects, and problems, because I find the idea of data *driving* any of these to be a misleading concept. Data should drive software only when that software is being built expressly for moving, storing, or otherwise handing the data. Software that's intended to address project or business goals should not be driven by data. That would be putting the cart before the horse. Problems and goals exist independently of any data, software, or other resources, but those resources may serve to solve the problems and to achieve the goals. The term *data-centric* reflects that data is an integral part of the solution, and I believe that using it instead of *data-driven* admits that we need to view the problems not from the perspective of the data but from the perspective of the goals and problems that data can help us address.

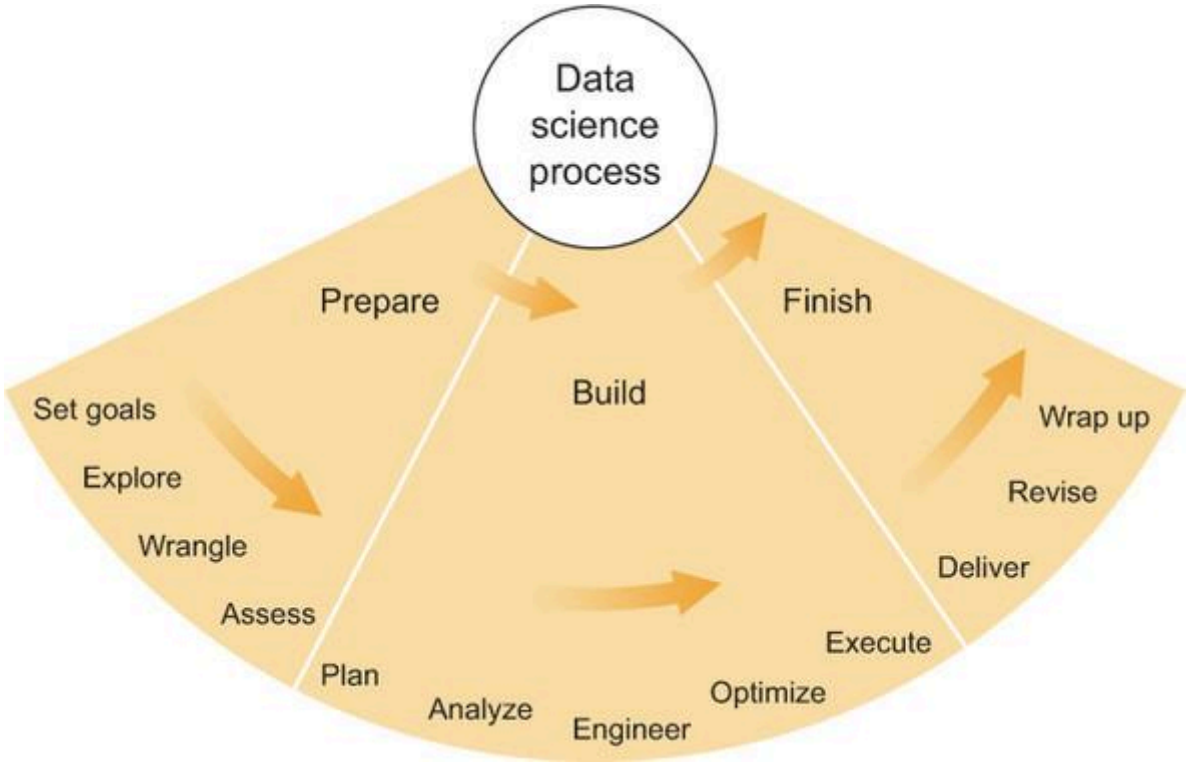
Such statements about proper perspective are common in this book. In every chapter I try to maintain the reader's focus on the most important things, and in times of uncertainty about project outcomes, I try to give guidelines that help you decide which are the most important things. In some ways, I think that locating and maintaining focus on the most important aspects of a project is one of the most valuable skills that I attempt to instruct within these pages. Data scientists must have many hard skills—knowledge of software development and statistics among them—but I've found this soft skill of maintaining appropriate perspective and awareness of the many moving parts in any data-centric problem to be very difficult yet very rewarding for most data scientists I know.

Sometimes data quality becomes an important issue; sometimes the major issue is data volume, processing speed, parameters of an algorithm, interpretability of results, or any of the many other aspects of the problem. Ignoring any of these at the moment it becomes important can compromise or entirely invalidate subsequent results. As a data scientist, I have as my goal to make sure that no important aspect of a project goes awry unnoticed. When something goes wrong—and something will—I want to notice it so that I can fix it. Throughout this chapter and the entire book, I

will continue to stress the importance of maintaining awareness of all aspects of a project, particularly those in which there is uncertainty about potential outcomes.

The lifecycle of a data science project can be divided into three phases, as illustrated in figure 1.2. This book is organized around these phases. The first part covers preparation, emphasizing that a bit of time and effort spent gathering information at the beginning of the project can spare you from big headaches later. The second part covers building a product for the customer, from planning to execution, using what you've learned from the first section as well as all of the tools that statistics and software can provide. The third and final part covers finishing a project: delivering the product, getting feedback, making revisions, supporting the product, and wrapping up a project neatly. While discussing each phase, this book includes some self-reflection, in that it regularly asks you, the reader, to reconsider what you've done in previous steps, with the possibility of redoing them in some other way if it seems like a good idea. By the end of the book, you'll hopefully have a firm grasp of these thought processes and considerations when making decisions as a data scientist who wants to use data to get valuable results.

Figure 1.2. The data science process



1.2. AWARENESS IS VALUABLE

If I had a dollar for every time a software developer told me that an analytic software tool "doesn't work," I'd be a wealthy man. That's not to say that I think all analytic software tools work well or at all—that most certainly is not the case—but I think it motivates a discussion of one of the most pervasive discrepancies between the perspective of a data scientist and that of what I would call a "pure" software developer—one who doesn't normally interact with raw or "unwrangled" data.

A good example of this discrepancy occurred when a budding startup founder approached me with a problem he was having. The task was to extract names, places, dates, and other key information from emails related to upcoming travel so that this data could be used in a mobile application that would keep track of the user's travel plans. The problem the founder was having is a common one: emails and other documents come in all shapes and sizes, and parsing them for useful information is a challenge.

It's difficult to extract this specific travel-related data when emails from different airlines, hotels, booking websites, and so on have different formats, not to mention that these formats change quite frequently. Google and others seem to have good tools for extracting such data within their own apps, but these tools generally aren't made available to external developers.

Both the founder and I were aware that there are, as usual, two main strategies for addressing this challenge: manual brute force and scripting. We could also use some mixture of the two. Given that brute force would entail creating a template for each email format as well as a new template every time the format changed, neither of us wanted to follow that path. A script that could parse any email and extract the relevant information sounded great, but it also sounded extremely complex and almost impossible to write. A compromise between the two extreme approaches seemed best, as it usually does.

While speaking with both the founder and the lead software developer, I suggested that they forge a compromise between brute force and pure scripting: develop some simple templates for the most common formats, check for similarities and common structural patterns, and then write a simple script that could match chunks of familiar template HTML or text within new emails and extract data from known positions within those chunks. I called this *algorithmic templating* at the time, for better or for worse. This suggestion obviously wouldn't solve the problem entirely, but it would make some progress in the right direction, and, more importantly, it would give some insight into the common structural patterns within the most common formats and highlight specific challenges that were yet unknown but possibly easy to solve.

The software developer mentioned that he had begun building a solution using a popular tool for natural language processing (NLP) that could recognize and extract dates, names, and places. He then said that he still thought the NLP tool would solve the problem and that he would let me know after he had implemented it fully. I told him that natural language is

notoriously tricky to parse and analyze and that I had less confidence in NLP tools than he did but I hoped he was right.

A couple of weeks later, I spoke again with the founder and the software developer, was told that the NLP tool didn't work, and was asked again for help. The NLP tool could recognize most dates and locations, but, to paraphrase one issue, "Most of the time, in emails concerning flight reservations, the booking date appears first in the email, then the departure date, the arrival date, and then possibly the dates for the return flight. But in some HTML email formats, the booking date appears between the departure and arrival dates. What should we do then?"

That the NLP tool doesn't work to solve 100% of the problem is clear. But it did solve some intermediate problems, such as recognizing names and dates, even if it couldn't place them precisely within the travel plan itself. I don't want to stretch the developer's words or take them out of context; this is a tough problem for data scientists and a *very* tough problem for others. Failing to solve the problem on the first try is hardly a total failure. But this part of the project was stalled for a few weeks while the three of us tried to find an experienced data scientist with enough time to try to help overcome this specific problem. Such a delay is costly to a startup—or any company for that matter.

The lesson I've learned through experiences like these is that awareness is incredibly valuable when working on problems involving data. A good developer using good tools to address what seems like a very tractable problem can run into trouble if they haven't considered the many possibilities that can happen when code begins to process data.

Uncertainty is an adversary of coldly logical algorithms, and being aware of how those algorithms might break down in unusual circumstances expedites the process of fixing problems when they occur—and they will occur. A data scientist's main responsibility is to try to imagine all of the possibilities, address the ones that matter, and reevaluate them all as successes and failures happen. That is why—no matter how much code I

write—awareness and familiarity with uncertainty are the most valuable things I can offer as a data scientist. Some people might tell you not to daydream at work, but an imagination can be a data scientist's best friend if you can use it to prepare yourself for the certainty that something will go wrong.

1.3. DEVELOPER VS. DATA SCIENTIST

A good software developer (or engineer) and a good data scientist have several traits in common. Both are good at designing and building complex systems with many interconnected parts; both are familiar with many different tools and frameworks for building these systems; both are adept at foreseeing potential problems in those systems before they're actualized. But in general, software developers design systems consisting of many well-defined components, whereas data scientists work with systems wherein at least one of the components isn't well defined prior to being built, and that component is usually closely involved with data processing or analysis.

The systems of software developers and those of data scientists can be compared with the mathematical concepts of logic and probability, respectively. The logical statement "if A, then B" can be coded easily in any programming language, and in some sense every computer program consists of a very large number of such statements within various contexts. The probabilistic statement "if A, then *probably* B" isn't nearly as straightforward. Any good data-centric application contains many such statements—consider the Google search engine ("These are *probably* the most relevant pages"), product recommendations on Amazon.com ("We *think* you'll *probably* like these things"), website analytics ("Your site visitors are *probably* from North America and each views *about* three pages").

Data scientists specialize in creating systems that rely on probabilistic statements about data and results. In the previous case of a system that

finds travel information within an email, we can make a statement such as "If we know the email contains a departure date, the NLP tool can *probably* extract it." For a good NLP tool, with a little fiddling, this statement is likely true. But if we become overconfident and reformulate the statement without the word *probably*, this new statement is much less likely to be true. It might be true some of the time, but it certainly won't be true all of the time. This confusion of probability for certainty is precisely the challenge that most software developers must overcome when they begin a project in data science.

When, as a software developer, you come from a world of software specifications, well-documented or open-source code libraries, and product features that either work or they don't ("Report a bug!"), the concept of uncertainty in software may seem foreign. Software can be compared to a car: loosely speaking, if you have all of the right pieces, and you put them together in the right way, the car works, and it will take you where you want it to go if you operate it according to the manual. If the car isn't working correctly, then quite literally something is broken and can be fixed. This, to me, is directly analogous to pure software development. Building a self-driving car to race autonomously across a desert, on the other hand, is more like data science. I don't mean to say that data science is as outrageously cool as an autonomous desert-racing vehicle but that you're never sure your car is even going to make it to the finish line or if the task is even possible. So many unknown and random variables are in play that there's absolutely no guarantee where the car will end up, and there's not even a guarantee that *any* car will ever finish a race—until a car does it.

If a self-driving car makes it 90% of the way to the finish line but is washed into a ditch by a rainstorm, it would hardly be appropriate to say that the autonomous car doesn't work. Likewise if the car didn't technically cross the finish line but veered around it and continued for another 100 miles. Furthermore, it wouldn't be appropriate to enter a self-driving sedan, built for roads, into a desert race and to subsequently proclaim that the car doesn't work when it gets stuck on a sand dune. That's precisely how I feel

when someone applies a purpose-built data-centric tool to a different purpose; they get bad results, and they proclaim that it doesn't work.

For a more concrete example, suppose you've been told by a website owner, "The typical user visits four pages of our site before leaving." Suppose you do an analysis of a new data set of site usage and find that the average user is visiting eight pages before leaving. Does that mean there's an error? Are you using the mean user when you should be using the median user? Does this new data include a different type of user or usage? These are questions that a data scientist, not a software developer, typically answers, because they involve data exploration and uncertainty. Implementing a software solution based on these questions and their answers can certainly benefit from the expertise of a software developer, but the exploration itself—necessarily involving statistics—falls squarely within the realm of a data scientist. In chapter 5, we'll look at data assessment and evaluation as a useful tool for preventing and diagnosing problems and for helping avoid the case where a seemingly finished software product fails in some way.

It's worth noting that, though I've seemed to pit data scientists and software developers against each other, this conflict (if I can call it that) can also be internal to a single person. While working on data science projects, I often find myself trading my data scientist hat for that of a software developer, particularly when writing production code. The reason I conceive of them as two different hats is that there can be conflicts of interest at times, because priorities can differ between the two. Openly discussing these conflicts, as I do in this book, can be helpful in illustrating the resolution of these differences, whether they occur between two or more people or within an individual who may wear either hat.

1.4. DO I NEED TO BE A SOFTWARE DEVELOPER?

Earlier, I discussed the difference between data scientists and software developers, often as if those are the only two options. Certainly, they are

not. And you don't need to be either in order to gain something from this book.

Knowledge of a statistical software tool is a prerequisite for doing practical data science, but this can be as simple as a common spreadsheet program (for example, the divisive but near-ubiquitous Microsoft Excel). In theory, someone could be a data scientist without ever touching a computer or other device. Understanding the problem, the data, and relevant statistical methods could be enough, as long as someone else can follow your intentions and write the code. In practice, this doesn't happen often.

Alternatively, you may be someone who often works with data scientists, and you'd like to understand the process without necessarily understanding the technology. In this case, there's also something in this book for you. One of my primary goals is to enumerate the many considerations that must be taken into account when solving a data-centric problem. In many cases, I'll be directing explanations in this book toward some semi-fictionalized colleagues from my past and present: biologists, finance executives, product owners, managers, or others who may have given me data and asked me a single question: "Can you analyze this for me?" For that last case, perhaps if I write it down here, in detail and with plenty of examples, I won't have to repeat (yet again) that it's never that simple. An analysis demands a question, and I'll discuss both of those in depth on these pages.

This book is about the process of thinking about and doing data science, but clearly software can't be ignored. Software—as an industry and its products—is the data scientist's toolbox. The tools of the craft are the enablers of work that's beyond the capabilities of the human mind and body alone. But in this book, I'll cover software only as much as is necessary to explore existing strengths and limitations of the software tools and to provide concrete examples of their use for clarification. Otherwise, I'll try to write abstractly about software—without being impractical—so that the explanations are accessible by as many people as possible, technical or not,

and years from now, the explanations may still be valuable, even after we've moved on to newer (and better?) software languages and products.

1.5. DO I NEED TO KNOW STATISTICS?

As with software, expert knowledge of statistics certainly helps but isn't necessary. At my core, I'm a mathematician and statistician, and so I'm most likely to veer into an overly technical tangent in these fields. But I despise jargon and presumed knowledge more than most, and so I'll try hard to include accessible conceptual explanations of statistical concepts; hopefully, these are sufficient to any reader with a little imagination and perseverance. Where I fall short, I'll try to direct you to some resources with more thorough explanations. As always, I'm an advocate of using web searches to find more information on topics that interest you, but at least in some cases, it may be better to bear with me for a few pages before heading down a rabbit hole of web pages about statistics.

In the meantime, to get you started conceptually, consider the field of statistics as the theoretical embodiment of the processes that generate the data you encounter on a daily basis. An anonymous website user is a random variable who might click any number of things depending on what's going on in their head. Social media data reflects the thoughts and concerns of the populace. Purchases of consumer goods depend on both the needs of the consumers as well as marketing campaigns for the goods. In each of these cases, you must theorize about how intangible thoughts, needs, and reactions are eventually translated into measurable actions that create data. Statistics provides a framework for this theorizing. This book will spend less time on complex theoretical justifications for statistical models and more on formulating mental models of data-generating processes and translating those mental models into statistical terminology, equations, and, ultimately, code.

1.6. PRIORITIES: KNOWLEDGE FIRST, TECHNOLOGY SECOND, OPINIONS THIRD

This section title is an adage of mine. I use it to help settle disputes in the never-ending battle between the various concerns of every data science project—for example, software versus statistics, changing business need versus project timeline, data quality versus accuracy of results. Each individual concern pushes and pulls on the others as a project progresses, and we're forced to make choices whenever two of them disagree on a course of action. I've developed a simple framework to help with that.

Knowledge, technology, and opinions are typically what you have at the beginning of any project; they are the three things that turn data into answers. *Knowledge* is what you know for a fact. *Technology* is the set of tools you have at your disposal. *Opinions* are those little almost facts you want to consider true but shouldn't quite yet. It's important to establish a hierarchy for your thought processes so that less-important things don't steamroll more-important ones because they're easier or more popular or because someone has a hunch.

In practice, the hierarchy looks like this:

- ***Knowledge first—*** Get to know your problem, your data, your approach, and your goal before you do anything else, and keep those at the forefront of your mind.
- ***Technology second—*** Software is a tool that serves you. It both enables and constrains you. It shouldn't dictate your approach to the problem except in extenuating circumstances.
- ***Opinions third—*** Opinions, intuition, and wishful thinking are to be used only as guides toward theories that can be proven correct and not as the focus of any project.

I'm not advocating that knowledge should always take precedence over technology in every decision—and likewise for technology over opinion—but if the hierarchy is to be turned upside down, you should be

doing it deliberately and for a very good reason. For instance, suppose you have a large amount of data and a statistical model for which you would like to estimate parameters. Furthermore, you already have a tool for loading that data into a system that performs a type of statistical parameter optimization called *maximum likelihood estimation* (MLE). You know that your data and statistical model are complex enough, possibly, to generate many reasonable parameter values, and so using MLE to find a single most likely parameter value might give unpredictable results. There exist more robust alternatives, but you don't currently have one implemented. You have two options:

1. Build a new tool that can do robust parameter estimation.
2. Use the tool you have to do MLE.

Your knowledge says you should do A, if you had unlimited time and resources, but the technology indicates that you should do B because A requires a tremendous outlay of resources. The pragmatic decision is probably B, but that inverts the hierarchy. As mentioned earlier, you can do this but only deliberately and for a very good reason. The good reason might be the difference in time and money that you'll need to spend on A and B. By *deliberately*, I mean that you should not make the decision lightly and you should not forget it. If you choose B, you should pass along with any results the knowledge that you made a sacrifice in quality in the interest of a cost savings, and you should make note of this in documentation and technical reports. You should appropriately intensify quality control and perform tests that check specifically for the types of optimization errors/biases to which MLE is prone. Making the decision and then forgetting the reasoning is a path to underwhelming or misleading results.

Opinion presents an even fuzzier challenge. Sometimes people are blinded by the potential of finding truly amazing results and forget to consider what might happen if those results aren't evident in the data. In the heyday of big data, any number of software startups attempted to exploit social media—particularly Twitter and its "firehose"—to determine trends within

various business markets, and they often ran into obstacles much larger than they expected. Scale of computation and data, parsing of natural language in only 140 characters, and inferring random variables on messy data are often involved. Only the very best of these firms were able to extract meaningful, significant knowledge from that data and earn a profit with it. The rest were forced to give it up or change their focus. Each of these startups, at some point, had to decide whether they wanted to spend even more time and money chasing a goal that was based mainly on hope and not on evidence. I'm sure many of them regretted how far they'd gone and how much money they'd spent when they did decide to pack it in.

Often people are blinded by what they think is possible, and they forget to consider that it might not be possible or that it might be much more expensive than estimated. These are opinions—guesses—not knowledge, and they shouldn't play a primary role in data analysis and product development. Goals are not certain to be achieved but are required for any project, so it's imperative not to take the goal and its attainability for granted. You should always consider current knowledge first and seek to expand that knowledge incrementally until you either achieve the goal or are forced to abandon it. I've mentioned this uncertainty of attainability as a particularly stark difference between the philosophies of software development and data science. In data science, a goal is much less likely to be achievable in exactly its original form. In a room full of software developers or inexperienced data scientists, be particularly wary of anyone presupposing without evidence that a goal is 100% achievable.

Remember: knowledge first, then technology, and then opinion. It's not a perfect framework, but I've found it helpful.

1.7. BEST PRACTICES

In my years of analyzing data and writing code as an applied mathematician, PhD student researcher, bioinformatics software engineer, data scientist, or any of the other titles I've had, I've run into a few

problems involving poor project management on my part. When I worked for years on my own research projects, which no one else touched or looked at, I frequently managed to set poorly documented code aside for long enough that I forgot how it worked. I'd also forgotten which version of the results was the most recent. I'd managed to make it nearly impossible for anyone else to take up my projects after I left that position. None of this was intentional; it was largely negligence but also ignorance of the usual ways in which people ensure that their project's materials and code can survive a long while on the shelf or in another's hands.

When working on a team—in particular, a team of experienced software developers—someone has usually established a set of best practices for documentation and preservation of project materials and code. It's usually important that everyone on the team abides by the team's strategies for these things, but in the absence of a team strategy, or if you're working by yourself, you can do a few things to make your life as a data scientist easier in the long run. The following subsections cover a few of my favorite ways to stay organized.

1.7.1. Documentation

Can you imagine what one of your peers might have to go through to take over your project if you left suddenly? Would taking over your project be a horrible experience? If you answer yes, please do these future peers—and yourself—a favor by staying current on your documentation. Here are some tips:

- Comment your code so that a peer unfamiliar with your work can understand what the code does.
- For a finished piece of software—even a simple script—write a short description of how to use it and put this in the same place (for example, the same file folder) as the code.
- Make sure everything—files, folders, documents, and so on—has a meaningful name.

1.7.2. Code repositories and versioning

Some software products have been built specifically to contain and manage source code for software. These are called *source code repositories* (or *repos*), and they can help you immensely, for a number of reasons.

First, most modern repos are based on versioning systems, which are also great. A versioning system tracks the changes you make in your code, allows you to create and compare different versions of your code, and generally makes the process of writing and modifying code much more pleasant once you get used to it. The drawback of repos and versioning is that they take time to learn and incorporate into your normal workflow. They're both worth the time, however. At the time of this writing, both Bitbucket.org and GitHub.com provide free web hosting of code repos, although both websites host both public and private repos, so make sure you don't accidentally make all of your source code public. Git is currently the most popular versioning system, and it incorporates nicely into both repo hosts mentioned. You can find tutorials on how to get started on the hosts' web pages.

Another reason why I find a remote repo-hosting service nearly indispensable is that it functions as a backup. My code will be safe even if my computer is accidentally crushed by an autonomous desert-race vehicle (though that hasn't happened to me yet). I make it a habit to *push* (send or upload) my latest code changes to the remote host almost every day, or about as often as I might schedule an automatic backup on a standard data-backup service.

Some code-hosting services have great web interfaces for looking through code history, various versions, development status, and the like, which has become standard practice for collaborating on teams and large projects. It's helpful for individuals and small projects as well, particularly when returning to an old project or trying to figure out which changes you made since a particular time in the past.

Finally, remote repos let you access your code from any place with web access. You don't need a computer with the appropriate editor and environment to browse through code. No, you normally can't do much except browse code (and maybe do simple editing) from these web interfaces, but the best ones have good language-specific code highlighting and a few other features that make browsing easy and useful.

Here are some tips on repos and versioning:

- Using a remote source code repo is now standard practice for most groups that write code; use one!
- It's absolutely worth the effort to learn Git or another versioning system; learn it!
- Commit your code changes to the repo often, perhaps daily or whenever you finish a specific task. Push those changes to a remote repo, so they're backed up and shared with others on your team.
- If your next code changes will break something, do the work in a location that won't affect the production version or the development by other team members. For example, you could create a new branch in Git.
- Use versioning, branching, and forking (tutorials abound on the internet) instead of copying and pasting code from one place to another and therefore having to maintain/modify the same code in multiple places.
- Most software teams have a Git guru. Ask them whenever you have questions about best practices; the time spent learning now will pay off in the end.

1.7.3. Code organization

Many books detail good coding practices, and I don't intend to replicate or replace them here. But a few guidelines can be very helpful, particularly when you try to share or reuse code. Most people who have been programming for some time will be familiar with these, but I've found that many people—particularly in academia and other work environments

without much coding collaboration—don't always know about them or adhere to them.

Here are the guidelines:

- Try to use coding patterns that are common for the particular programming language. Python, R, and Java, for instance, all have significant differences in the way developers typically organize their code. Any popular resource for the language contains examples and guidelines for such coding patterns.
- Use meaningful names for variables and other objects. This makes your code much more understandable for new collaborators and your future self.
- Use plenty of informative comments, for the reasons just mentioned.
- Don't copy and paste code. Having the same code active in two places means twice as much work for you when you want to change it. Encapsulating that code in a function, method, object, or library makes sure that later modifications happen in only one place.
- Try to code in chunks with specific functionalities. For scripts, this means having well-commented snippets (optionally in separate files or libraries) that each accomplish a specific task. For applications, this means having relatively simple functions, objects, and methods so that each does specific things. A good general rule is if you can't name your snippet, function, or method in such a way that the name generally describes everything the chunk of code accomplishes, you should probably break the chunk into smaller, simpler chunks.
- Don't optimize prematurely. This is a common mantra for programmers everywhere. Make sure you code your algorithm in a logical, coherent fashion, and only if you find out later that your implementation is inefficient should you try to shave off compute cycles and bytes of memory.
- Pretend that someone else will, at some point, join your project. Ask yourself, "Could they read my code and figure out what it does?" If not, spend a little time organizing and commenting sooner rather than later.

1.7.4. Ask questions

This may sound obvious or trivial, but it's so important that I include it here. I've already discussed how awareness is one of the greatest strengths of a good data scientist; likewise, the unwillingness to gain awareness via any and all means can be a great weakness. The stereotype of an introverted academic being too shy to ask for help may be familiar to you, but have you heard of the know-it-all PhD data scientist who was too proud to admit he didn't know something? I certainly have. Pride is perhaps a bigger pitfall to data scientists these days than shyness, but you should be wary of both.

Software engineers, business strategists, sales executives, marketers, researchers, and other data scientists all know more than you do about their particular domains or projects, and it would be a shame to ignore the wealth of knowledge they possess, due to shyness, pride, or any other reason. A business setting wherein nearly everyone else has a role different from yours provides ample opportunity to learn about the company and the industry. This is the subject matter expertise, or domain knowledge, that I've mentioned already. Nontechnical business folks, in my experience, have a tendency to treat you, the data scientist, as the smart one in the conversation, but don't forget that they tend to know a lot more than you about project goals and business problems, two extremely important concepts. Never hesitate to have a lengthy discussion with someone who knows the business side of the project and problems you're working on. I often find that such conversations illuminate projects in new ways, sometimes causing strategy changes but always contributing to the domain knowledge that's necessary for me to finish a project successfully.

1.7.5. Stay close to the data

Being "close to the data" means making sure that the methods and algorithms that you're applying to the data aren't so dense as to obscure it. Another way to phrase this would be "don't use methods that are more complex than needed, and always be conscious of the possibility of mistakes."

Many people will argue with this piece of advice, and I agree with these detractors that many complex methods have proven their worth. The field of machine learning is a perfect example of a source of such methods. In the cases where complex methods (*black box* methods, in some cases) give considerable advantage, the concept of being close to the data can be adapted: make sure that some results from complex methods can be verified, justified, or supported by simpler methods that are close to the data. This could include a glorified form of spot-checking, whereby you can pick some results at random, extract the raw data that's relevant to these results, and use logic and/or simple statistics to make sure that the results make sense intuitively. Straying too far from the data without a safety line can get you in trouble, because these problems are the hardest to diagnose. Throughout the book, and in each example, I'll return to this concept and give more specifics.

1.8. READING THIS BOOK: HOW I DISCUSS CONCEPTS

It has been my experience—over and over—that complex concepts appear vastly more complex at the point when you begin to learn about them, as compared to later, after you've begun to understand them. This is not only because all concepts seem less complex once you begin to understand them, but also because people, in general, who might be explaining a concept to you or who are writing about it revel in the complexities that they understand and often have little patience or sympathy for those who don't understand. For example, it's difficult for most statistics professors to explain a simple statistical test to a layperson. This inability to explain in simple terms extends across all fields and industries, I've found. Part of this problem is that most people love to use jargon, and they love to prove their knowledge of it. This is, perhaps, a fault of human nature. But from this I've learned to ignore my frustrations at the beginning of learning a new, complex concept and not to get hung up on any specific point until I've gone through the concept as a whole.

Throughout this book, I try to explain new concepts in simple terms before getting into specifics. This is the way I prefer to learn and the way I prefer to explain. Despite this, you'll still have moments when you get stuck. I implore you suspend your frustration, stick with it to the end of the chapter, and then review the concept as a whole. At that point, if something still doesn't make sense, perhaps rereading the paragraphs in question can help, and if not, feel free to consult other resources. I employ conceptual thought processes, and I intend to focus on the whole before the parts. Knowing this in advance may help you while reading.

Before diving into the practical steps of the data science process in the next chapter and beyond, I'd like to note that this book is intended to be accessible for non-experts in data science, software, and statistics. If you're not a beginner, you may occasionally find some sections that cover material you already know. I'd like to think that every section of this book provides a useful or even fresh perspective on its topic, even for experts, but if you're pressed for time, feel free to skip ahead to something of more use to you. On the other hand, I'd like to discourage skipping whole chapters of this book when reading it for the first time. Each chapter describes a step in the data science process, and skipping one of them could break the continuity of the process and how it's presented. Rather than skipping, it would be better to read at least the beginning and the end of the chapter and to skim section by section in order to gain important context for subsequent chapters.

As a final disclaimer, throughout the book I pull many practical examples from my own experience. Several of my early, formative data science projects were in the field of bioinformatics, and so sometimes I delve into discussions of genetics, RNA, and other biological concepts. This may seem heavy to some people, and that's OK. It's not necessary to understand the biology as long as you're able to understand the data-oriented aspects such as project goals, data sources, and statistical analysis. This is also true for examples from other fields, but I rely particularly heavily on bioinformatics because I can still remember well what it was like to encounter various challenges of data science at that early phase of my career. Furthermore, I

won’t shy away from using highly specified examples because it’s always good practice for a data scientist to learn the basics of a new field and attempt to apply experience and knowledge to problems therein. I do, however, try to present all examples in a way that anyone can understand.

SUMMARY

- Awareness is perhaps the biggest strength of a good data scientist, particularly in the face of uncertainty.
- Dealing with uncertainty is often what separates the role of a data scientist from that of a software developer.
- Setting priorities and balancing them with limitations and requirements can be done using a helpful framework that I’ve outlined.
- Using some of the best practices that I suggest can spare you from headaches later.
- I’m a conceptual learner and teacher who loves talking about things abstractly before delving into practical examples, so please keep that in mind throughout the book.