Building Intelligent Systems

A Guide to Machine Learning Engineering

—

Geoff Hulten

Apress®

Building Intelligent Systems

A Guide to Machine Learning Engineering

Geoff Hulten

Apress®

Building Intelligent Systems: A Guide to Machine Learning Engineering

Geoff Hulten
Lynnwood, Washington, USA

ISBN-13 (pbk): 978-1-4842-3431-0 ISBN-13 (electronic): 978-1-4842-3432-7
https://doi.org/10.1007/978-1-4842-3432-7

Library of Congress Control Number: 2018934680

Copyright © 2018 by Geoff Hulten

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Susan McDermott
Development Editor: Laura Berendson
Coordinating Editor: Rita Fernando

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484234310. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To Dad, for telling me what I needed to hear.

To Mom, for pretty much just telling me what I wanted to hear.

And to Nicole.

Table of Contents

About the Author

Geoff Hulten is a machine learning scientist and PhD in machine learning. He has managed applied machine learning teams for over a decade, building dozens of Internet-scale Intelligent Systems that have hundreds of millions of interactions with users every day. His research has appeared in top international conferences, received thousands of citations, and won a SIGKDD Test of Time award for influential contributions to the data mining research community that have stood the test of time.

About the Technical Reviewer

Jeb Haber has a BS in Computer Science from Willamette University. He spent nearly two decades at Microsoft working on a variety of projects across Windows, Internet Explorer, Office, and MSN. For the last decade-plus of his Microsoft career, Jeb led the program management team responsible for the safety and security services provided by Microsoft SmartScreen (anti-phishing, anti-malware, and so on.) Jeb's team developed and managed global-scale Intelligent Systems with hundreds of millions of users. His role included product vision/planning/strategy, project management, metrics definition and people/team development. Jeb helped organize a culture along with the systems and processes required to repeatedly build and run global scale, 24×7 intelligence and reputation systems. Jeb is currently serving as the president of two non-profit boards for organizations dedicated to individuals and families dealing with the rare genetic disorder phenylketonuria (PKU).

Acknowledgments

There are so many people who were part of the Intelligent Systems I worked on over the years. These people helped me learn, helped me understand. In particular, I'd like to thank:

Jeb Haber and John Scarrow for being two of the key minds in developing the concepts described in this book and for being great collaborators over the years. None of this would have happened without their leadership and dedication.

Also: Anthony P., Tomasz K., Rob S., Rob M., Dave D., Kyle K., Eric R., Ameya B., Kris I., Jeff M., Mike C., Shankar S., Robert R., Chris J., Susan H., Ivan O., Chad M. and many others…

Introduction

Building Intelligent Systems is a book about leveraging machine learning in practice.

It covers everything you need to produce a fully functioning Intelligent System, one that leverages machine learning and data from user interactions to improve over time and achieve success.

After reading this book you'll be able to design an Intelligent System end-to-end. You'll know:

- When to use an Intelligent System and how to make it achieve your goals.
- How to design effective interactions between users and Intelligent Systems.
- How to implement an Intelligent System across client, service, and back end.
- How to build the intelligence that powers an Intelligent System and grow it over time.
- How to orchestrate an Intelligent System over its life-cycle.

You'll also understand how to apply your existing skills, whether in software engineering, data science, machine learning, management or program management to the effort.

There are many great books that teach data and machine-learning skills. Those books are similar to books on programming languages; they teach valuable skills in great detail. This book is more like a book on software engineering; it teaches how to take those base skills and produce working systems.

This book is based on more than a decade of experience building Internet-scale Intelligent Systems that have hundreds of millions of user interactions per day in some of the largest and most important software systems in the world. I hope this book helps accelerate the proliferation of systems that turn data into impact and helps readers develop practical skills in this important area.

Who This Book Is For

This book is for anyone with a computer science degree who wants to understand what it takes to build effective Intelligent Systems.

Imagine a typical software engineer who is assigned to a machine learning project. They want to learn more about it so they pick up a book, and it is technical, full of statistics and math and modeling methods. These are important skills, but they are the wrong information to help the software engineer contribute to the effort. *Building Intelligent Systems* is the right book for them.

Imagine a machine learning practitioner who needs to understand how the end-to-end system will interact with the models they produce, what they can count on, and what they need to look out for in practice. *Building Intelligent Systems* is the right book for them.

Imagine a technical manager who wants to begin benefiting from machine learning. Maybe they hire a machine learning PhD and let them work for a while. The machine learning practitioner comes back with charts, precision/recall curves, and training data requests, but no framework for how they should be applied. *Building Intelligent Systems* is the right book for that manager.

Data and Machine Learning Practitioners

Data and machine learning are at the core of many Intelligent Systems, but there is an incredible amount of work to be done between the development of a working model (created with machine learning) and the eventual sustainable customer impact. Understanding this supporting work will help you be better at modeling in a number of ways.

First, it's important to **understand the constraints** these systems put on your modeling. For example, where will the model run? What data will it have access to? How fast does it need to be? What is the business impact of a false positive? A false negative? How should the model be tuned to maximize business results?

Second, it's important to be able to **influence the other participants**. Understanding the pressures on the engineers and business owners will help you come to good solutions and maximize your chance for success. For example, you may not be getting all the training data you'd like because of telemetry sampling. Should you double down on modeling around the problem, or would an engineering solution make more sense? Or maybe you are being pushed to optimize for a difficult extremely-high precision, when your models are already performing at a very good (but slightly lower) precision. Should you keep chasing that super-high precision or should you work to influence the user experience in ways that reduce the customer impact of mistakes?

Third, it's important to understand how the **supporting systems can benefit you**. The escalation paths, the manual over-rides, the telemetry, the guardrails that prevent against major mistakes—these are all tools you can leverage. You need to understand when to use them and how to integrate them with your modeling process. Should you discard a model that works acceptably for 99% of users but really, really badly for 1% of users? Or maybe you can count on other parts of the system to address the problem.

Software Engineers

Building software that delights customers is a lot of work. No way around it, behind every successful software product and service there is some serious engineering. Intelligent Systems have some unique properties which present interesting challenges. This book describes the associated concepts so you can design and build Intelligent Systems that are efficient, reliable, and that best-unlock the power of machine learning and data science.

First, this book will identify the **entities and abstractions** that need to exist within a successful Intelligent System. You will learn the concepts behind the intelligence runtime, context and features, models, telemetry, training data, intelligence management, orchestration, and more.

Second, the book will give you a **conceptual understanding of machine learning and data sciences**. These will prepare you to have good discussions about tradeoffs between engineering investments and modeling investments. Where can a little bit of your work really enable a solution? And where are you being asked to boil the ocean to save a little bit of modeling time?

Third, the book will explore **patterns for Intelligent Systems** that my colleagues and I have developed over a decade and through implementing many working systems. What are the pros and cons or running intelligence in a client or in a service? How do you bound and verify components that are probabilistic? What do you need to include in telemetry so the system can evolve?

Program Managers

Machine learning and Data Sciences are hot topics. They are fantastic tools, but they are tools; they are not solutions. This book will give you enough conceptual understanding so you know what these tools are good at and how to deploy them to solve your business problems.

The first thing you'll learn is to develop an **intuition for when machine learning and data science are appropriate**. There is nothing worse than trying to hammer a square peg into a round hole. You need to understand what types of problems can be solved by machine learning. But just as importantly, you need to understand what types of problems can't be—or at least not easily. There are so many participants in a successful endeavor, and they speak such different, highly-technical, languages, that this is particularly difficult. This book will help you understand enough so you can ask the right questions and understand what you need from the answers.

The second is to get an intuition on return on investment so you can determine **how much Intelligent System to use**. By understanding the real costs of building and maintaining a system that turns data into impact you can make better choices about when to do it. You can also go into it with open eyes, and have the investment level scoped for success. Sometimes you need all the elements described in this book, but sometimes the right choice for your business is something simpler. This book will help you make good decisions and communicate them with confidence and credibility.

Finally, the third thing a program manager will learn here is to understand how to **plan, staff, and manage an Intelligent System project**. You will get the benefit of our experience building many large-scale Intelligent Systems: the life cycle of an Intelligent System; the day-to-day process of running it; the team and skills you need to succeed.

PART I

Approaching an Intelligent Systems Project

Chapters 1-4 set the groundwork for a successful Intelligent Systems project. This part describes what Intelligent Systems are and what they are good for. It refreshes some important background. It explains how to ensure that an Intelligent System has a useful, achievable goal. And it gives an overview of what to expect when taking on an Intelligent Systems project.

CHAPTER 1

Introducing Intelligent Systems

Intelligent Systems are all around us. In our light bulbs. In our cars. In our watches. In our thermostats. In our computers. How do we make them do the things that make our lives better? That delight us?

When should a light bulb turn on? When should an e-commerce site show us a particular product? When should a search engine take us to a particular site? When should a speaker play some music?

Answering questions like these, and answering them extremely well, is fundamental to unlocking the value of Intelligent Systems. And it is really hard.

Some of the biggest, most valuable companies in the world have their core business built around answering simple questions, like these:

- What web page should I display based on a short query?
- What ad should I present in a particular context?
- What product should I show to this shopper?
- What movie would this user enjoy right now?
- What book would this person like to read?
- Which news stories will generate the most interest?
- Which programs should I block from running to keep a machine safe?

Answering each of these questions "well enough" has made companies worth billions—in a few cases hundreds of billions—of dollars. And it has done so by making lots of people smarter, more productive, happier, and safer. But this is just the tip of the iceberg.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_1

There are tens of thousands of similar questions we could try to answer: When should my front door unlock? What exercise should a fitness app suggest next? What type of song should an artist write next? How should a game evolve to maximize player engagement?

This book is about reliably and efficiently unlocking the potential of Intelligent Systems.

Elements of an Intelligent System

Intelligent Systems connect users to artificial intelligence (machine learning) to achieve meaningful objectives. An Intelligent System is one in which the intelligence evolves and improves over time, particularly when the intelligence improves by watching how users interact with the system.

Successful Intelligent Systems have all of the following:

- **A meaningful objective**. An Intelligent System must have a reason for being, one that is meaningful to users and accomplishes your goals, and one that is achievable by the Intelligent System you will be able to build and run. Selecting an objective is a critical part of achieving success, but it isn't easy to do. The first part of this book will help you understand what Intelligent Systems do, so you'll know when you should use one and what kinds of objectives you should set for it.

- **The intelligent experience**. An intelligent experience must take the output of the system's intelligence (such as the predictions its machine learning makes) and present it to users to achieve the desired outcomes.

  To do this it must have a user interface that adapts based on the predictions and that puts the intelligence in a position to shine when it is right—while minimizing the cost of mistakes it makes when it is wrong. The intelligent experience must also elicit both implicit and explicit feedback from users to help the system improve its intelligence over time. The second part of this book will explore intelligent experiences, the options and the pitfalls for connecting users with intelligence.

- **The implementation of the intelligence**. The Intelligent System implementation includes everything it takes to execute intelligence, to move the intelligence where it needs to be, to manage it, to light up the intelligent experiences based on it, to collect telemetry to verify the system is functioning, and to gather the user feedback that will improve the intelligence over time. The third part of this book describes all the components of an intelligence implementation. It will prepare you to design and implement an Intelligent System of your own.
- **Intelligence creation**. Intelligent Systems are about setting intelligence up for success. This intelligence can come from many different places, ranging from simple heuristics to complex machine learning. Intelligence must be organized so that the right types of intelligence address the right parts of the problem, and so it can be effectively created by a team of people over an extended time. The fourth part of this book discusses the act of creating and growing intelligence for Internet-scale Intelligent Systems. It will prepare you to leverage all available approaches and achieve success.
- **The orchestration**. An Intelligent System lives over time, and all its elements must be kept in balance to achieve its objectives. This orchestration includes controlling how the system changes, keeping the experience in sync with the quality of the intelligence, deciding what telemetry to gather to track down and eliminate problems, and how much money to spend building and deploying new intelligence. It also involves dealing with mistakes, controlling risk, and defusing abuse. The fifth part of this book explains everything it takes to orchestrate an Intelligent System and achieve its goals through all phases of its life-cycle.

An Intelligent System is one way to apply machine learning in practice. An Intelligent System takes *intelligence* (produced via machine learning and other approaches) and leverages and supports it to achieve your objectives and to improve over time.

An Example Intelligent System

Intelligent Systems might be used to implement search engines, e-commerce sites, self-driving cars, and computer vision systems that track the human body and know who a person is and when they are smiling. But these are big and complicated systems.

Let's look at a much simpler example to see how a solution might evolve from a traditional system into an Intelligent System.

The Internet Toaster

Let's consider an Internet-connected smart-toaster. A good idea? Maybe, maybe not. But let's consider it. Our toaster has two controls: a slider that controls the intensity of the toasting and a lever that starts the toast.

It seems simple enough. The toaster's intelligence simply needs to map settings of the intensity slider to toast times. At a low setting, the toaster runs for, say, 30 seconds. At high settings the toaster runs for two minutes. That kind of thing.

So sit down, think for a while, come up with some toast-time settings and send the toaster to customers. What could go wrong?

Well, if you choose a maximum intensity that toasts for too long it could burn the things it toasts. Most customers who use that setting will be unhappy, throw away their cinder-toast, and start again.

You can imagine other failure cases, all the little irritations of toast-making that result in customers standing over their toasters, hands on the levers, ready to cut the toast short. Or customers repeatedly toasting the same piece of bread, a bit at a time, to get it the way they like it.

That's not good. If we are going to build a toaster, we want to build a really good one.

So maybe we do some testing, tweaking the toaster until both the high and low settings make toast that we think is desirable. Not too crispy, not too cool.

Great.

Did we get it right? Will this toaster do what our customers want?

It's hard to know. No matter how much toast we've eaten in our lives, it's really impossible to prove we've gotten the settings right for all the types of toast all our customers might want to make.

And so we realize we need to incorporate the opinions and experiences of others into our toaster-building process. But how?

Maybe we start with a focus group. Bring in dozens of members of the toast-making public, put them in a toasting lab, and take notes as they toast.

Then we tweak the toast-time settings again to reflect the way these volunteers toast. Now do we have the perfect toaster? Will this focus-group–tuned toaster make all the right toast that hundreds of thousands of people all around the world will want?

What if someone puts something frozen into the toaster? Or something from the fridge? Or what if someone has a tradition of making toaster s'mores? Or what if someone invents a new toaster-product, unlike anything humanity has ever toasted before? Is our toaster right for all of these situations? Probably not.

Using Data to Toast

So maybe making a perfect toasting machine is a bit harder than just asking a few people what they like.

There are just too many use cases to optimize by hand if we want to get the perfect toast in every conceivable situation. We could run focus groups every day for the rest of our lives and we still wouldn't see all the types of toast a toaster might make.

We need to do better. It's time for some serious data science.

The toaster is Internet connected, so we can program it to send telemetry back to our service. Every time someone toasts something we can know what setting they used and how long the toasting went before it stopped.

We ship version 1 of the toaster (perhaps to a controlled set of users), and the toast telemetry starts flooding in to our servers.

Now we know exactly which intensity settings people are using in their real lives (not in some contrived lab setting). We know how many times people push the lever down to start a toast job and how many times they pop the lever up to stop one early.

Can we use this data to make a better toaster?

Of course!

We could set the maximum intensity to something that at least a few users are actually using. Then we could set up metrics to make sure we don't have the toaster biased to over-toasting. For example, we could monitor the percentage of toasts that are early-stopped by their user (presumably because they were about to burn something) and tweak and tweak till we get those under control.

We could set the minimum intensity to something reasonable too. Something that users seem to use. We could track the double-toast rate (where someone toasts something and immediately re-toasts it) and tweak to make sure the toaster isn't biased to under-toasting.

Heck, we can even set the default intensity, the one in the middle of the range, to the most commonly used toast time.

Since our toasters are Internet-connected, we can update them with the new settings by having them pull the data from our servers. Heck, we could tweak the toaster's settings every single day, twice on Sunday—this is the age of miracles and wonders!

There are some seams with this approach, some things we had to assume. For example, we had to assume that toasting multiple times in quick succession is a sign of failure, that the customer is re-toasting the same bread instead of making several pieces of toast rapid-fire.

We had to assume that an early-stop is a sign the bread was starting to burn and not a sign that the customer was late for work and rushing out the door.

Also when we deploy the new settings to toasters, how do we ensure that users are going to like them? We are pretty sure (based on data science) that the new settings better-match what our overall user population is doing, so that's good.

But what about the user who was getting their own brand of perfectly toasted bagel yesterday and today they get… something different?

Despite these problems, we've got a pretty decent toaster. We've got telemetry to know the toaster is roughly doing its job. We've got a way to service it and improve it over time. Now let's really blow the doors off this thing.

Sensors and Heuristic Intelligence

If we want to make the best toaster, we're going to need more than a single slider and a toast lever. Let's add some sensors:

- A weight sensor to know how much toast is in the toaster and to determine when a customer places something in the toaster and when they take something out of it.
- A temperature sensor to know if the item placed in the toaster is chilled, frozen, or room temperature.
- A location sensor to know what region of the world the toaster is in so it can adapt to different tastes in different locales.

- A proximity sensor to know if someone is near the toaster and a camera to identify who it is.
- A clock to know if a toast is a breakfast toast or a dinner one.
- A little memory to know what's been toasted recently and to monitor the pattern of setting changes and toastings.
- A smoke sensor to know when the toaster has made a bad mistake and is about to burn something.

Now when a customer walks up to the toaster and puts something in it, the toaster can look at who the user is, try to guess what they are trying to toast, and automatically suggest a setting.

Heck, if the toaster is good enough there is no longer any need for the intensity setting or the toasting lever at all. We could update the toasting experience to be totally automatic. We could ship toasters with no buttons or knobs or anything. The customer drops something in, walks away, and comes back to delightfully toasted—anything!

If we could just figure out a way to turn all these sensor readings into the correct toast times for… anything.

To do that we need intelligence—the program or rules or machine-learned model that makes these types of decisions.

Let's start simply, by hand-crafting some intelligence.

Let's write a set of rules that consider the sensor readings and output intensity suggestions. For example: If something cold and heavy is put into the toaster, then toast for 5 minutes at high intensity. But for every degree above freezing, reduce the toast time by 2 seconds. But in Britain add 15 seconds to the toast time (because they like it that way). But if the weight and size match a known toaster product (some big brand toaster meal), then toast it the "right amount of time" based on the product's directions.

And that kind of stuff.

Every time a user complains, because the toaster toasted something wrong, you can add a new rule.

Whenever the telemetry shows a spike of double-toastings you can tweak the rules to improve.

Every time you add a new rule or update an old one you can update all the toasters you shipped all around the world by having them download the new settings from your server.

With this approach, you'll probably need to write and maintain a lot of rules to deal with all the possibilities. It's going to be a lot of work. You could employ a dozen people and give them months of rule-writing-time, and you might not be happy with the outcome.

You might never be.

Toasting with Machine Learning

In situations like this, where there is an optimization problem that is too hard or too expensive to do by hand, people turn to machine learning.

At a high level, machine learning can observe examples of users using their Internet toasters and automatically produce a set of rules to control the toaster just the way users would want it. Kind of like the hand-crafted heuristic ones we just talked about, only machine-crafted.

And machines can make lot more rules than humans can. Machines can balance inputs from dozens of sensors; they can optimize for thousands of different types of users simultaneously. They can incorporate new data and reoptimize everything every day, every hour—sometimes even faster. They can personalize rules to each user.

In order to work, machine learning needs data that shows the situation the user faced (the sensor readings) the action they took (how they set the toaster) and the outcome they got (if they liked the toast or had to tweak something and try again). Machine learning needs examples of when things are going right as well as examples of when things are going wrong. Lots and lots of examples of them.

To find examples of when things are going right, we can look through the telemetry for times when the user put something in the toaster, pressed the start lever, waited to completion, took the item out, and walked away. These are cases where the user probably got what they wanted. We have a record of all the sensor settings, the toasting time. We can use this as training for the machine learning system. Perfect.

And for examples of when things went wrong we can look through the telemetry again. This time we find all the places where users had to fiddle with the toaster, every time they stopped the toaster early or retoasted the same item multiple times. We have records of the sensor readings and intensity settings that gave users bad outcomes, too.

So combining the good outcomes and the bad outcomes and some machine learning, we can automatically train the perfect toaster-controlling algorithm.

Put in data, get out a program that looks at sensor readings and determines the best intensity and time settings to use.

We push this learned program to our customers' toasters.

We can do it every day, every minute.

We can learn from hundreds of thousands of customer interactions—from hundreds of millions of them.

Magic!

Making an Intelligent System

And this is what Intelligent Systems do. Building an effective Intelligent System requires balancing five major components: the objective, the experience, the implementation, the intelligence, and the orchestration.

We need a problem that is suitable to solve with an Intelligent System, and worth the effort. We need to control the toaster based on what the intelligence thinks. We need to do it in a way the user finds helpful, and we also need to give the user the right controls to interact with the toaster and to give feedback we can use to learn. We need to build all the services and tools and code that gathers telemetry, produces intelligence, moves it where it needs to be, and hooks it up with users.

We need to create the intelligence every day, over and over, in a way that is predictable. And we need to keep everything running over time as new toastable products come into the market and tastes change.

We need to decide how much telemetry we should collect to trade off operational costs with the potential value. We need to decide how much to change the intelligence on any particular day so that the toaster improves, but without confusing or upsetting users.

We need to monitor our key metrics and react if they start to degrade. Maybe tune the experience? Maybe call an emergency intelligence intervention? And we need to deal with mistakes. And mistakes happen.

Intelligence (particularly machine-learned intelligence) can make bad mistakes—spectacular, counter-intuitive, highly customer-damaging mistakes.

For example, our toaster might learn that people in a specific zip code love cinder-toast. No matter what you put into the thing, if you live in that zip code—cinder. So we need ways to identify and manage these mistakes. We need ways to place guardrails on machine learning.

And unfortunately… people are people. Any time we use human feedback to optimize something—like using machine learning to build a toaster—we have to think about all the ways a human might benefit from things going wrong.

Imagine a major bread-maker paying a sweatshop of people to toast their competitor's products using crazy settings. They could carry out millions of toasts per week, flooding our telemetry system with misleading data. Our machine learning might pick up on this and "learn" what this toast-hacker is trying to teach it—our toaster might start regularly undercooking the competitor's product, leading to food safety issues.

Not good.

A successful Intelligent System will have to consider all of these issues and more.

Summary

This chapter introduced Intelligent Systems along with five key conceptual challenges that every Intelligent System must address—the objective, the experience, the implementation, the intelligence, and the orchestration.

It should now be clear that there are many ways to approach these challenges and that they are highly interrelated. To have a successful Intelligent System you must balance them. If one of the conceptual challenges is difficult in your context, the others will need to work harder to make up for it.

For example, if you are trying to retrofit an Intelligent System into an existing system where the experience has already been defined (and can't be changed), the Intelligent System might need to accept a less aggressive objective, to invest more in intelligence, or to have a fuller mistake-mitigation strategy.

But here's another way to look at it: there are a lot of ways to succeed.

Deploy an Intelligent System and discover that the intelligence problem is harder than you thought? No need to panic. There are lots of ways to compensate and make a system that delights customers and helps your business while the intelligence takes time to grow.

The rest of this book will give you the tools to approach Intelligent Systems projects with confidence.

For Thought…

After reading this chapter you should be able to:

- Identify Intelligent Systems in the world around you.
- See the potential that Intelligent Systems can unlock.
- Understand the difference between intelligence (which makes predictions about the world) and an Intelligent System (which combines objective, experience, implementation, intelligence, and orchestration to achieve results).
- Articulate all the conceptually hard things you will need to address to build a successful Intelligent System.
- Understand how these difficult things interact, including some of the tradeoffs and ways they can support one another.

You should be able to answer questions like these:

- What services do you use that (you suspect) are built by turning customer data into intelligence?
- What is the most magical experience you've had with one of these services?
- What is the worst experience you've had?
- Can you identify how the user experience supports the intelligence?
- Can you find any information on how its intelligence is produced? Maybe in a news story or publication?
- Can you determine any of the ways it detects and mitigates intelligence mistakes?

CHAPTER 2

Knowing When to Use Intelligent Systems

So you have a problem you want to solve. An existing system you need to optimize. A new idea to create a business. Something you think your customers will love. A machine-learning–based solution that isn't producing the value you'd hoped. Is an Intelligent System right for you? Sometimes yes. Sometimes no.

This chapter discusses when Intelligent Systems might be the right approach, and provides guidance on when other approaches might be better. It begins by describing the types of problems that can benefit from Intelligent Systems. It then discusses some of the properties required to make Intelligent Systems work.

Types of Problems That Need Intelligent Systems

It is always best to do things the easy way. If you can solve your problem without an Intelligent System, maybe you should. One key factor in knowing whether you'll need an Intelligent System is how often you think you'll need to update the system before you have it right. If the number is small, then an Intelligent System is probably not right.

For example, imagine implementing intelligence for part of a banking system. Your "intelligence" needs to update account balances as people withdraw money. The solution is pretty simple:

```
NewBalance = OldBalance - WithdrawalAmount
```

Something like that (and a bunch of error checking and logging and transaction management), not rocket science. Maybe you'll have a bug. Maybe you'll miss an error case (for example, if the balance goes negative). Then you might need to update the "intelligence" to get it right. Maybe a couple times? Or maybe you're super sloppy and you need to update it a dozen times before you're done.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_2

Intelligent Systems are not for problems like this. They are for problems where you think you're going to have to update the system much, much more. Thousands of times, tens of thousands of times, every hour for as long as the system exists. That kind of thing.

There are four situations that clearly require that level of iteration:

- Big problems, that require a lot of work to solve.
- Open-ended problems, which continue to grow over time.
- Time-changing problems, where the right answer changes over time.
- Intrinsically hard problems, which push the boundaries of what we think is possible.

The rest of this section will explore these each in turn.

Big Problems

Some problems are big. They have so many variables and conditions that need to be addressed that they can't really be completed in a single shot.

For example, there are more web pages than a single person could read in their lifetime—more than a hundred people could read. There are so many books, television programs, songs, video games, live event streams, tweets, news stories, and e-commerce products that it would take thousands of person-years just to experience them all.

These problems and others like them require massive scale. If you wanted to build a system to reason about one of these, and wanted to completely finish it before deploying a first version… Well, you'd probably go broke trying.

When you have a big problem that you don't think you can finish in one go, an Intelligent System might be a great way to get started, and an efficient way to make progress on achieving your vision, by giving users something they find valuable and something they are willing to help you improve.

Open-Ended Problems

Some problems are more than big. Some problems are open-ended. That is, they don't have a single fixed solution at all. They go on and on, requiring more work, without end. Web pages, books, television programs, songs, video games, live event streams—more and more of them are being created every day.

Trying to build a system to reason about and organize things that haven't even been created yet is hard.

In these cases, a static solution—one where you build it, deploy it, and walk away—is unlikely to work. Instead, these situations require services that live over long periods of time and grow throughout their lifetimes.

If your problem has a bounded solution, an Intelligent System might not be right. But if your problem is big and on-going, an Intelligent System might be the right solution.

Time-Changing Problems

Things change. Sometimes the right answer today is wrong tomorrow. For example:

- Imagine a system for identifying human faces—and then facial tattoos become super popular.
- Imagine a system for predicting stock prices—and then an airplane crashes into a building.
- Imagine a system for moving spam email to a junk folder—and then a new genius-savant decides to get in the spam business and changes the game.
- Or Imagine a UX that users struggle to use—and then they begin to learn how to work with it.

One thing's for certain—things are going to change.

Change means that the intelligence you implemented yesterday—which was totally right for what was happening, which was making a lot of users happy, maybe even making your business a lot of money—might be totally wrong for what is going to happen tomorrow.

Addressing problems that change over time requires the ability to detect that something has changed and to adapt quickly enough to be meaningful.

If your domain changes slowly or in predictable ways, an Intelligent System might not be needed. On the other hand, if change in your domain is unpredictable, drastic, or frequent, an Intelligent System might be the right solution for you.

Intrinsically Hard Problems

Some problems are just hard. So hard that humans can't quite figure out how to solve them. At least not all at once, not perfectly. Here are some examples of hard problems:

- Understanding human speech.
- Identifying objects in pictures.
- Predicting the weather more than a few minutes in the future (apparently).
- Competing with humans in complex, open-ended games.
- Understanding human expressions of emotion in text and video.

In these situations, machine learning has had great success, but this success has come on the back of years (or decades) of effort, gathering training data, understanding the problems, and developing intelligence. These types of systems are still improving and will continue to improve for the foreseeable future.

There are many ways to make progress on such hard problems. One way is to close the loop between users and intelligence creation in a meaningful application using an Intelligent System as described in this book.

Situations When Intelligent Systems Work

In addition to having a problem that is difficult enough to need an Intelligent System—one that is big, open-ended, time-changing, intrinsically hard, or some combination—an Intelligent System needs a few more things to succeed:

- A problem where a partial solution is viable and interesting.
- A way to get data from usage of the system to improve intelligence.
- An ability to influence a meaningful objective.
- A problem that justifies the effort of building an Intelligent System.

The rest of this section explores these requirements in turn.

When a Partial System Is Viable and Interesting

Intelligent Systems are essentially always incomplete, incorrect in important ways, and likely to make all sorts of mistakes. It isn't important that an Intelligent System is perfect. What matters is that an Intelligent System must be good enough to be interesting to users (and valuable to you).

An Intelligent System is viable when the value of the things it does right is (much) higher than the cost of the things it does wrong.

Consider an Intelligent System that makes "cheap" mistakes. Maybe the system is supposed to show a shopping list in an order that optimizes the user's time in the grocery store. Head to the right, pick up eggs, then milk, then loop to the back of the store for the steak, then into the next aisle for pickles… and on and on. If this system gets a few things backwards it might waste time by sending users in the wrong direction, say 15–20 seconds per mistake. Irritating, but not the end of the world. If the mistakes aren't too frequent—say just one every other shopping trip—it's easy to imagine someone using the system while it is learning the layouts of all the grocery stores in the world and adapting to changes.

Consider another Intelligent System, one that makes "expensive" mistakes. Maybe it is controlling surgical robots, and mistakes put human life at risk. This system would have to meet a much higher quality bar before it is viable. That is, you'd need to produce a much better partial system before deploying it to users. But even this system doesn't need to be perfect to be viable. Remember—surgeons make mistakes, too. An Intelligent System doesn't have to be perfect (and most never will be), it just has to be better than the other guy.

The point is that to be viable, an Intelligent System must provide a good deal for users (and for you) by producing more value with positive outcomes than it produces irritation (and cost) with negative outcomes.

When You Can Use Data from the System to Improve

Intelligent Systems work by closing the loop between usage and intelligence. When users use the system, the intelligence gets better; and when the intelligence gets better, the system produces more value for users.

To make intelligence better with usage, you need to record the interactions between users and the systems in a careful way (which we will discuss in detail later). At a high level, this involves capturing what the user saw, what they did, and whether the outcome was positive or negative. And you need to observe many, many user interactions (the harder the problem, the more you'll need). This type of data can be used to improve intelligence in many ways, including with machine learning.

When the System Can Interface with the Objective

The Intelligent System must be able to change things that influence the objective. This could be through automating a part of the system, or presenting information or options to a user that help them achieve their goals. Intelligent Systems are most effective when the following conditions are met:

- The actions the Intelligent System can take *directly* affect the objective. When there is a positive outcome, it should be largely because the Intelligent System did well; and when there is a negative outcome, it should be largely because the Intelligent System did poorly. The more external factors that affect the outcome, the harder it is to make an effective Intelligent System.
- The actions the Intelligent System can take *quickly* affect the objective. Because the more time between the action and the outcome, the more chance there is for external factors to interfere.
- The actions the Intelligent System can take are in balance with the objectives. That is, the actions the Intelligent System can take are meaningful enough to the outcome that they are usually measurable.

In practice, figuring out where to put Intelligent Systems and how much of your problem to try to solve with them will be key challenges. Too big an objective and you won't be able to connect the Intelligent System to it; too small and it won't be worth all the work.

In fact, many large, complex systems have space for more than one Intelligent System inside them. For example, an operating system might have an Intelligent System to predict what to put into a RAM cache, one to predict what files to show the user when they search, one to recommend apps to users based on usage, and another to manage the tradeoffs between power and performance when the computer is not plugged in.

You can imagine using a single Intelligent System to manage an entire OS. But it probably wouldn't work, because:

- There are too many things to control.
- The controls are too abstract from the goal.
- The types of feedback the system gets would not relate directly enough to the decisions the Intelligent System needs to make.

When it is Cost Effective

Intelligent Systems have different costs than other approaches to building computer systems and services. There are three main components of cost in most similar systems: The Intelligence, the Implementation, and the Orchestration. We will discuss these concepts in Parts 3, 4, and 5 of this book. But roughly *Intelligence* is the ways the system makes decisions about what is right to do and when, the "logic" of the system; *Implementation* is all the services, back-end systems, and client code that implements the system and interfaces intelligence outcomes with users; and *Orchestration* is the managing of the system throughout its life-cycle, such as making sure it is achieving its objectives, and dealing with mistakes.

The intelligence is usually cheaper in Intelligent Systems than other approaches, for a couple of reasons:

1. In general, Intelligent Systems produce the complex logic of a system automatically (instead of using humans). When addressing a big, hard, or open-ended task, Intelligent Systems can lead to substantial savings on intelligence production.
2. With the proper intelligent experience, Intelligent Systems produce the training data (which machine learning requires) automatically as users interact with it. Gathering training data can be a major cost driver for intelligence creation, often requiring substantial human effort, and so getting data from users can drastically shift the economics of a problem.

The implementation of an Intelligent System is similar to or a bit more expensive than the implementation other service-based systems (such as a web service or a chat service). Most of the components are similar between an Intelligent System and a nonintelligent system, but Intelligent Systems have additional requirements related to orchestrating them over their lifecycles, including creating intelligence, changing intelligent experiences, managing mistakes, organizing intelligence, and so on. We will discuss all of these in detail later in the book.

The orchestration costs of an Intelligent System are similar but a bit more expensive compared to running a non-intelligent service (like a web service or a chat service). Intelligent Systems require ongoing work, including tuning the way the intelligence is exposed to users, growing the intelligence to be better and deal with new situations, and identifying and mitigating mistakes.

By the time you've finished this book you'll be able to prepare a detailed analysis of the efficiency of using an Intelligent System for your application.

When You Aren't Sure You Need an Intelligent System

Sometimes you have a place where users, data, and intelligence interact, but you aren't sure you need all the elements of an Intelligent System. This might be for any of these reasons:

- You aren't sure your problem is actually hard, so you don't want to build a whole bunch of fancy intelligence you might not need.
- You realize the problem is hard, but aren't sure the extra work of building an Intelligent System will justify the effort for you.
- You want to take an incremental approach and solve problems as you encounter them, rather than stopping to build an Intelligent System.

That's fine. Intelligent Systems (as described in this book) aren't right for everyone—there are plenty of ways to build systems that solve hard problems. But if you do try to solve a big, open-ended, time-changing, or hard problem, you'll eventually run into many of the challenges described in this book. Even if you choose not to build an Intelligent System as described here, this book will arm you to identify common issues quickly, understand what is going on, and will prepare you to respond with proven approaches.

Summary

Intelligent Systems are most useful when solving big, open-ended, time-changing, or intrinsically hard problems. These are problems where a single solution won't work, and where the system needs to improve over time—months or years.

Intelligent Systems work well when:

- A partial solution is interesting to users and will support a viable business, so that you can start simply and grow the system's capability over time.
- You can get data to improve the system from users as they use the system. Data is usually one of the main costs in building intelligence, so crafting a system that produces it automatically is very valuable.
- When the things the system can control affect the objectives: directly, quickly, and meaningfully. That is, the outcomes the system is getting can be tied to things the Intelligent System is doing.
- When other approaches are too expensive to scale.

When looking for places to use Intelligent Systems, break down your problem into pieces. Each piece should have a clear objective that is critical to overall success, feedback that is quick and direct, and controls (or user experiences your system can control) that directly impact the objective. Each such piece is a candidate for an Intelligent System.

And even if you choose not to use an Intelligent System at all, the patterns in this book about practical machine learning, interfacing intelligence with user experience, and more will benefit just about any machine learning endeavor.

For Thought

After reading this chapter, you should know:

- Whether your system could benefit from an Intelligent System.
- How to find potential Intelligent Systems in the world around you.

You should be able to answer questions like these:

- What is your favorite hobby that would benefit from an Intelligent System?
- What is an example of a common activity you do every day where an Intelligent System might not be right? Why not?

Consider your favorite piece of software:

- Identify three places the software could use an Intelligent System.
- Which one is the most likely to succeed? Which the least? Why?

CHAPTER 3

A Brief Refresher on Working with Data

Data is central to every Intelligent System. This chapter gives a conceptual overview of working with data, introducing key concepts from data science, statistics, and machine learning. The goal is to establish a baseline of understanding to facilitate discussions and decision making between all participants of an Intelligent System project.

This chapter will cover:

- Structured data.
- Asking questions of data.
- Data models.
- Conceptual machine learning.
- Some common pitfalls of working with data.

Structured Data

Data is changing the world, that's for sure.

But what is it? A bunch of pictures on a disk? All the term papers you wrote in high school? The batting averages of every baseball player who ever played?

Yeah, all of that is data. There is so much data—oceans of numbers, filling hard disk drives all over the planet with ones and zeros. In raw form, data is often referred to as unstructured, and it can be quite difficult to work with.

When we turn data into intelligence, we usually work with data that has some structure to it. That is, data that is broken up into units that describe entities or events.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_3

For example, imagine working with data about people. Each unit of the data describes a single person by: their weight, their height, their gender, their eye color, that sort of stuff.

One convenient way to think about it is as a table in a spreadsheet (Figure 3-1). There is one row for each person and one column for each property of the person. Maybe the first column of each row contains a number, which is the corresponding person's weight. And maybe the third column of each row contains a number, which is the person's height. On and on.

Weight	Gender	Height (Inches)	Eye Color	...
170	Male	70	Hazel	...
140	Female	60	Brown	...
60	Male	50	Blue	...
...	...	...	...	...

Figure 3-1. *An example of structured data*

The columns will usually contain numbers (like height and weight), or be chosen from a small number of categories (like brown or blue for eye color), or be short text strings like names. There are more advanced options, of course, but this simple scheme is quite powerful and can be used to represent all sorts of things (and unlock our ability to do statistics and machine learning on them).

For example:

- You can represent a web page by: the number of words it contains, the number of images it has in it, the number of links it contains, and the number of times it contains each of 1,000 common keywords.

- You can represent a visit to a search engine by: the query the user typed, the amount of time the user remained on the results page, the number of search results the user clicked, and the URL of the final result they clicked.
- You can represent a toasting event on our Internet toaster by: the temperature of the item placed in the toaster, the intensity setting the user selected, the number of times the user stopped and started the toaster before removing the item, and the total amount of time between when the item was placed in the toaster and when it was taken out.

These are simple examples. In practice, even simple concepts tend to have dozens or hundreds of elements in their representation (dozens or hundreds of columns per row). Choosing the right data sets to collect and the right representations to give them is critical to getting good results from data. And you'll probably get it wrong a few times before you get it right—be prepared to evolve.

Asking Simple Questions of Data

So what can you do with data? You can *ask it questions*. For example, in a data set of people you might want to know:

- What is the average height?
- Who is the tallest person?
- How many people are shorter than 5'5"?

These are pretty easy to answer; simply look through the data and calculate the values (sum up the numbers, or count the rows that meet the criteria).

In an Intelligent System you'll be asking plenty of similar questions. Things like:

- How many times per day do users take a particular action?
- What percentage of users click an ad after they engage with the intelligent experience?
- What's the average revenue per customer per month?

Answering questions like these is important to understanding if a system is meeting its goals (and making sure it isn't going haywire).

Another thing you can do with data is to *make projections*. Imagine a data set with a hundred people in it. Want to know the average height? No problem; just calculate it. But what if you want to know the height of the next person who will be added to the data set? Or whether the next person added to the set will be shorter or taller than 5'?

Can you do these types of things? Sure! Well, sort of, using basic statistics.

With a few simple assumptions, statistics can estimate the most likely height for the next person added to the data set. But statistics can do more. It can express exactly how accurate the estimate is; for example:

> *The most likely height of the next person is 5'10", and with 95% confidence the next person will be between 5'8" and 6' tall.*

This is called a confidence interval. The width of the confidence interval depends on how much data you have, and how "well behaved" the data is. More data results in a narrower window (for example, that the next person is 95% likely to be between 5'9" and 5'11"). Less data results in a wider one (like 5'5" to 6'3"). Why does this matter?

Let's imagine optimizing an Internet business by reducing customer support capacity to save money. Say the current system has capacity for 200 customer support calls per week. So looking into historical telemetry, you can calculate that the average week had 75 support calls and that the maximum number of calls in any week was 98. Intuitively, 200 is much higher than 75—the system must be wasting a lot of money. Cutting capacity in half (down to 100) seems safe, particularly because you don't have any week on record with a higher call volume than that.

But check the confidence interval. What if it came back that the most likely call volume is 75 per week, and with 95% confidence the next week will have between 40 and 110 calls? Well, then 100 doesn't seem like such an obviously good answer. In a data-intensive project, you should always ask for answers. But you should also ask how sure the answers are. And you should make sure the decisions you make take both the answer and the degree of certainty into account.

Working with Data Models

Models capture the answers from data, converting them into *more convenient* or *more useful* formats.

Technically, the projection we discussed previously was a model. Recall that the data set on call center volumes was converted into an average weekly volume of 75 with a 95% confidence interval of 40 - 110. The raw data might have been huge. It might have contained ten terabytes of telemetry going back a decade. But the model contained just four numbers: 75, 95%, 40, and 110.

And this simple model was more useful for making decisions about capacity planning, because it contained exactly the relevant information, and captured it in an intuitive way for the task at hand.

Models can also *fill in gaps in data* and estimate answers to questions that the data doesn't contain.

Consider the data set of people and their heights and weights. Imagine the data set contains a person who is 5'8" and a person who is 5'10", but no one who is 5'9".

What if you need to predict how much a 5'9" person will weigh?

Well, you could build a simple model. Looking into the data, you might notice that the people in the data set tend to weigh about 2.5 pounds per inch of height. Using that model, a 5'9" person would weigh 172.5 pounds. Great. Or is it?

One common way to evaluate the quality of models is called *generalization error*. This is done by reserving a part of the data set, hiding it from the person doing the modeling, and then testing the model on the holdout data to see how well the model does—how well it generalizes to data it has not seen.

For example, maybe there really was a 5'9" person in the data set, but someone hid them from the modeling process. Maybe the hidden 5'9"er weighed 150 pounds. But the model had predicted 172.5 pounds. That's 22.5 pounds off.

Good? Bad? Depends on what you need the model for.

If you sum up these types of errors over dozens or hundreds of people who were held out from the modeling process, you can get some sense of how good the model is. There are lots of technical ways to sum up errors and communicate the quality of a model, two important ones are:

- **Regression Errors** are the difference between a numeric value and the predicted value, as in the weight example. In the previous example, the model's regression error was 22.5.

- **Classification Errors** are for models that predict categories, for example one that tries to predict if a person is a male or female based on height and weight (and, yeah, I agree that model wouldn't have many friends). One way to measure classification errors involves accuracy: the model is 85% accurate at telling males from females.

Conceptual Machine Learning

Simple models can be useful. The weight-at-height model is very simple, and it's easy to find flaws with it. For example, it doesn't take gender into account, or body fat percent, or waist circumference, or where the person is from. Statistics are useful for making projections. Create a simple model, use it to answer questions. But models can be complex too—very, very, very complex.

Machine learning uses computers to improve the process of producing (complex) models from data. And these models can make projections from the data, sometimes surprisingly accurate—and sometimes surprisingly inaccurate.

A machine-learning algorithm explores the data to determine the best way to combine the information contained in the representation (the columns in the data set) into a model that generalizes accurately to data you haven't already seen.

For example, a machine-learning algorithm might predict any of the following:

- Gender by combining height, weight, age, and name.
- Weight using gender, height, age, and name.
- Height from the gender and weight.
- And so on…

And the way the model works can be very complicated, multiplying and scaling the various inputs in ways that make no sense to humans but produce accurate results.

There are probably thousands of machine-learning algorithms that can produce models of data. Some are fast; others run for days or weeks before producing a model. Some produce very simple models; some produce models that take megabytes or gigabytes of disk space. Some produce models that can make predictions very quickly on new data, a millisecond or less; some produce models that are computationally intensive to execute. And some do well on certain types of data, but fail at other types of problems.

There doesn't seem to be one universally best machine-learning algorithm to use for all situations—most work well in some situations, but poorly in others. Every few years a new technique emerges that does surprisingly well at solving important problems, and gets a lot of attention.

And some machine-learning people really like the algorithm they are most familiar with and will fight to the death to defend it despite any evidence to the contrary. You have been warned.

The machine-learning process generally breaks down into the following phases, all of which are difficult and require considerable effort from experts:

- **Getting the data to model.** This involves dealing with noise, and figuring out exactly what will be predicted and how to get the training examples for the modeling algorithm.
- **Feature engineering.** This involves processing the data into a data set with the right representation (columns in the spreadsheet) to expose to machine learning algorithms.
- **The modeling.** This involves running one or more machine-learning algorithms on the data set, looking at the mistakes they make, tweaking and tuning things, and repeating until the model is accurate enough.
- **The deployment.** This involves choosing which model to run, how to connect it into the system to create positive impact and minimal damage from mistakes.
- **The maintenance.** This involves monitoring that the model is behaving as expected, and fixing it if it starts to go out of control. For example, by rerunning the training algorithm on new data.

This book is about how to do these steps for large, complex systems.

Common Pitfalls of Working with Data

Data can be complicated, and there are a lot of things that can go wrong. Here are a few common pitfalls to keep in mind when working with data intensive systems, like Intelligent Systems.

Confidence intervals can be broken. Confidence intervals are very useful. Knowing that something is 95% likely to be between two values is great. But a 95% chance of being within an interval means there is a 5% chance of being outside the interval, too. And if it's out, it can be way out. What does that mean? 5% is one in twenty. So if you are estimating something weekly, for example to adjust capacity of a call center, one out of every twenty weeks will be out of interval—that's 2.6 times per year that you'll have too much capacity or not enough. 5% sounds small, like something that might never happen, but keep in mind that even low probability outcomes will happen eventually, if you push your luck long enough.

There is noise in your data. Noise is another word for errors. Like maybe there is a person in the dataset who has a height of 1", or 15'7". Is this real? Probably not. Where does the noise come from? All sorts of places. For example, software has bugs; data has bugs too. Telemetry systems log incorrect values. Data gets corrupted while being processed. The code that implements statistical queries has mistakes. Users turn off their computers at odd times, resulting in odd client states and crazy telemetry. Sometimes when a client has an error, it won't generate a data element. Sometimes it will generate a partial one. Sometimes this noise is caused by computers behaving in ways they shouldn't. Sometimes it is caused by miscommunication between people. Every large data set will have noise, which will inject some error into the things created from the data.

Your data has bias. Bias happens when data is collected in ways that are systematically different from the way the data is used. For example, maybe the people dataset we used to estimate height was created by interviewing random people on the streets of New York. Would a model of this data be accurate for estimating the heights of people in Japan? In Guatemala? In Timbuktu?

Bias can make data less useful and bias can come from all sorts of innocent places, like simple oversights about where the data is collected, or the context of when the data was collected. For example, users who "sorta-liked" something are less likely to respond to a survey than users who loved it or who hated it. Make sure data is being used to do the thing it was meant to do, or make sure you spend time to understand the implications of the bias your data has.

Your data is out of date. Most (simple) statistical and machine learning techniques have a big underlying assumption—that is, that things don't change. But things do change. Imagine building a model of support call volume, then using it to try to estimate support call volume for the week after a major new feature is released. Or the week after a major storm. One of the main reasons to implement all the parts of an Intelligent System is to make the system robust to change.

You want the data to say things it doesn't. Sometimes data is inconclusive, but people like to have answers. It's human nature. People might downplay the degree of uncertainty by saying things like "the answer is 42" instead of saying "we can't answer that question" or "the answer is between 12 and 72." It makes people feel smarter to be precise. It can almost seem more polite to give people answers they can work with (instead of giving them a long, partial answer). It is fun to find stories in the data, like "our best product is the toothpaste, because we redid the display-case last month." These stories are so seductive that people will find them even where they don't exist.

Tip When working with data, always ask a few questions: Is this right? How sure are we? Is there another interpretation? How can we know which is correct?

Your models of data will make mistakes. Intelligent Systems, especially ones built by modeling data, are wrong. A lot. Sometimes these mistakes make sense. But sometimes they are totally counter-intuitive gibberish. And sometimes it is very hard to fix one mistake without introducing new ones. But this is fine. Don't be afraid of mistakes. Just keep in mind: any system based on models is going to need a plan for dealing with mistakes.

Summary

Data is changing the world. Data is most useful when it is structured in a way that lines up with what it needs to be used for. Structured data is like a spreadsheet, with one row per thing (person, event, web page, and so on), and one column per property of that thing (height, weight, gender).

Data can be used to answer questions and make projections. Statistics can answer simple questions and give you guidance on how accurate the answer is. Machine learning can create very complicated models to make very sophisticated projections. Machine learning has many, many useful tools that can help make accurate models of data. More are being developed all the time.

And data can be misused badly—you have to be careful.

For Thought

After reading this chapter, you should:

- Understand the types of things data is used for.
- Have some intuition about how to apply the things data can do.
- Have some intuition about how data can go wrong and lead to bad outcomes.

You should be able to answer questions like this:

- What is the biggest mistake you know of that was probably made because someone misused data?

CHAPTER 4

Defining the Intelligent System's Goals

An Intelligent System connects intelligence with experience to achieve a desired outcome. Success comes when all of these elements are aligned: the outcome is achievable; the intelligence is targeted at the right problem; and the experience encourages the correct user behavior.

A good success criterion connects these elements. It expresses the desired outcome in plain language. It indicates what sub-problems the intelligence and experience need to solve, and it ties those solutions to the desired larger scale (organizational) outcome. Implicit in good success criteria is a framework that allows all participants to see how their work contributes to the overall goal. This helps prevent them from heading in the wrong direction, even when data and their experiences are conspiring to mislead.

This chapter discusses setting goals for Intelligent Systems, including:

- What makes good goals.
- Why finding good goals is hard.
- The various types of goals a system can have.
- Some ways to measure goals.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_4

Criteria for a Good Goal

A successful goal will do all of the following:

1. Clearly ***communicate the desired outcome*** to all participants. Everyone should be able to understand what success looks like and why it is important, no matter what their background or technical experience.

2. ***Be achievable***. Everyone on the team should believe they are set up to succeed. The goal can be difficult, but team members should be able to explain roughly how they are going to approach success and why there is a good chance it will work.

3. ***Be measurable***. Intelligent Systems are about optimizing, and so Intelligent Systems are about measuring. Because if you can't measure something you really aren't going to be able to optimize it.

It isn't easy to know if a goal is correct. In fact, bridging the gap between high-level objectives and detailed properties of the implementation is often the key challenge to creating a successful Intelligent System. Some goals will seem perfect to some participants but make no sense to others. Some will clearly align with positive impact but be impossible to measure or achieve. There will always be trade-offs, and it is common to spend a great deal of time refining the definition of success.

But that's OK. It's absolutely worth the effort because failing to define success before starting a project is the easiest, most certain way to waste time and money.

An Example of Why Choosing Goals Is Hard

Consider an anti-phishing feature backed by an Intelligent System.

One form of phishing involves web sites that look like legitimate banking sites but are actually fake sites, controlled by abusers. Users are lured to these phishing sites and tricked into giving their banking passwords to criminals. Not good.

So what should an Intelligent System do?

Talk to a machine-learning person and it won't take long to get them excited. They'll quickly see how to build a model that examines web pages and predicts whether they are phishing sites or not. These models will consider things like the text and images on the web pages to make their predictions. If the model thinks a page is a phish, block it. If a page is blocked, a user won't browse to it, won't type their banking password into it. No more problem. Easy. Everyone knows what to do.

So the number of blocks seems like a great thing to measure—block more sites and the system is doing a better job.

Or is it?

What if the system is so effective that phishers quit? Every single phisher in the world gives up and finds something better to do with their time?

Perfect!

But then there wouldn't be any more phishing sites and the number of blocks would drop to zero. The system has achieved total success, but the metric indicates total failure.

Not great.

Or what if the system blocks one million phishing sites per day, every day, but the phishers just don't care? Every time the system blocks a site, the phishers simply make another site. The Intelligent System is blocking millions of things, everyone on the team is happy, and everyone feels like they are helping people—but the same number of users are losing their credentials to abusers after the system was built as were losing their credentials before it was built.

Not great.

One pitfall with defining success in an Intelligent System is that there are so many things that can be measured and optimized. It's very easy to find something that is familiar to work with, choose it as an objective, and get distracted from true success.

Recall the three properties of a good success criterion:

1. Communicate the desired outcome
2. Be achievable
3. Be measurable

Using the number of blocked phishing pages as a success metric hits #2 and #3 out of the park, but fails on #1.

The desired outcome of this system isn't to block phishing sites—it is to stop abusers from getting users' banking passwords.

Types of Goals

There are many types of things a system can try to optimize, ranging from very concrete to very abstract.

A system's true objective tends to be very abstract (like making money next quarter), but the things it can directly affect tend to be very concrete (like deciding whether a toaster should run for 45 or 55 seconds). Finding a clear connection between the abstract and concrete is a key source of tension in setting effective goals. And it is really, really hard.

One reason it is hard is that different participants will care about different types of goals. For example:

- Some participants will care about making money and attracting and engaging customers.
- Some participants will care about helping users achieve what they are trying to do.
- Some participants will care that the intelligence of the system is accurate.

These are all important goals, and they are related, but the connection between them is indirect. For example, you won't make much money if the system is always doing the wrong thing; but making the intelligence 1% better will not translate into 1% more profit.

This section discusses different ways to consider the success of an Intelligent System, including:

- Organizational objectives
- Leading indicators
- User outcomes
- Model properties

Most Intelligent Systems use several of these on a regular basis but focus primarily on user outcomes and model properties for day-to-day optimization.

Organizational Objectives

Organizational objectives are the real reason for the Intelligent System. In a business these might be things like revenue, profit, or number of units sold. In a nonprofit organization these might be trees saved, lives improved, or other benefits to society.

Organizational objectives are clearly important to optimize. But they are problematic as direct objectives for Intelligent Systems for at least three reasons:

1. They are very distant from what the technology can affect. For example, a person working on an Internet toaster can change the amount of time a cold piece of bread is toasted—how does that relate to number of units sold?
2. They are affected by many things out of the system's control. For example, market conditions, marketing strategies, competitive forces, changes to user behavior over time, and so on.
3. They are very slow indicators. It may take weeks or months to know if any particular action has impacted an organizational objective. This makes them difficult to optimize directly.

Every Intelligent System should contribute to an organizational objective, but the day-to-day orchestration of an Intelligent System will usually focus on more direct measures—like the ones we'll discuss in the next few sections (particularly user outcomes and model properties).

Leading Indicators

Leading indicators are measures that correlate with future success. For example:

- You are more likely to make a profit when your customers like your product than when they hate it.
- You are more likely to grow your customer base when your customers are recommending your product to their friends than when your customers are telling their friends to stay away.
- You are more likely to retain customers when they use your product every day than when they use your product once every couple of months.

Leading indicators are a way to bridge between organizational objectives and the more concrete properties of an Intelligent System (like user outcomes and model properties). If an Intelligent System gets better, customers will probably like it more. That may lead to more sales or it might not, because other factors—like competitors, marketing activities, trends, and so on—can affect sales. Leading indicators factor some of these external forces out and can help you get quicker feedback as you change your Intelligent System.

There are two main types of leading indicators: customer sentiment and customer engagement.

Customer sentiment is a measure of how your customers feel about your product. Do they like using it? Does it make them happy? Would they recommend it to a friend (or would they rather recommend it to an enemy)?

If everyone who uses your product loves it, it is a sign that you are on the right track. Keep going, keep expanding your user base, and eventually you will have business success (make revenue, sell a lot of units, and so on).

On the other hand, if everyone who uses your product hates it you might be in for some trouble. You might have some customers, they might use your product, but they aren't happy with it. They are looking for a way out. If you get a strong competitor your customers are ready to jump ship.

Sentiment is a fuzzy measure, because users' feelings can be fickle. It can also be very hard to measure sentiment accurately—users don't always want to tell you exactly what you ask them to tell you. Still, swings in sentiment can be useful indicators of future business outcomes, and Intelligent Systems can certainly affect the sentiment of users who encounter them.

Customer engagement is a measure of how much your customers use your product. This could mean the frequency of usage. It could also mean the depth of usage, as in using all the various features your product has to offer.

Customers with high engagement are demonstrating that they find value in your product. They've made a habit of your product, and they come back again and again. They will be valuable to you and your business over time.

Customers with low engagement use the product infrequently. These customers may be getting value from your offering, but they have other things on their minds. They might drift away and never think about you or your product again.

Leading indicators have some disadvantages as goals for Intelligent Systems, similar to those that organizational outcomes suffer from:

- They are indirect.
- They are affected by factors out of control of the Intelligent System.
- They aren't good at detecting small changes.

- They provide slow feedback, so they are difficult to optimize directly.
- And they are often harder to measure than organizational objectives (how many surveys do you like to answer?).

Still, leading indicators can be useful, particularly as early indicators of problems—no matter what you think your Intelligent System should be doing, if customers have much worse sentiment after an update than they had before the update, you are probably doing something wrong.

User Outcomes

Another approach for setting goals for Intelligent Systems is to look at the outcomes your users are getting. For example:

- If your system is about helping users find information, are they finding useful information efficiently?
- If your system is about helping users make better decisions, are they making better decisions?
- If your system is about helping users find content they will enjoy, are they finding content that they end up liking?
- If your system is about optimizing the settings on a computer, are the computers it is optimizing faster?
- And if your system is about helping users avoid scams, are they avoiding scams?

Intelligent Systems can set goals around questions and decisions like these and try to optimize the outcomes users get.

This is particularly useful because outcomes rely on a combination of the intelligence and the experience of the Intelligent System. In order for a user to get a good outcome, the intelligence must be correct, and the experience must help the user benefit.

For example, in the anti-phishing example, imagine intelligence that is 100% accurate at identifying scams. If the experience blocks scam pages based on this intelligence, users will get good outcomes. But what if the experience is more subtle? Maybe it puts a little warning on the user's browser when they visit a scam site—a little red X in the address bar. Some users won't notice the warning. Some won't interpret it correctly. Some will ignore it. In this case some users will get bad outcomes (and give their passwords to scammers) even though the intelligence had correctly identified the scam.

User outcomes can make very good targets for Intelligent Systems because they measure how well the intelligence and the experience work together to influence user behavior.

Model Properties

Within every Intelligent System there are concrete, direct things to optimize, for example:

- The error rate of the model that identifies scams.
- The probability a user will have to re-toast their bread.
- The fraction of times a user will accept the first recommendation of what content to use.
- The click-through rate of the ads the system decides to show.

These types of properties don't always line up exactly with user outcomes, leading indicators, or organizational objectives, but they do make very good goals for the people who are working to improve Intelligent Systems.

For example, a model that is right 85% of the time (on test data in the lab) is clearly better than one that is right 75% of the time. Clear and concrete. Easy to get fast feedback. Easy to make progress.

But model properties have some disadvantages as goals for Intelligent Systems:

1. *They are not connected to actual user reality*. For example, if the Internet toaster always gets within 10 seconds of the optimal toast time will users like it? Would 5 seconds of error be better? Sure, obviously, of course. But how much better? Will that error reduction make tastier toast? What should the goal be? If we could get to a model with 4 seconds of error is that enough? Should we stop or press on for 3 seconds of error? How much investment is each additional second worth?
2. *They don't leverage the full system*. A model might make a mistake, but the mistake will be perceived by users in the context of a full system. Maybe the user experience makes the mistake seem so

minor that no one cares. Or maybe there is a really good way for the user to give feedback, which quickly corrects the mistake—way more cheaply than investing in optimizing the last few points of the model's performance.

3. *They are too familiar to machine-learning people*. It is easy to build Intelligent Systems to optimize model properties—it is precisely what machine-learning people spend their lives doing, so it will naturally come up in any conversation about objectives. Be careful with them. They are so powerful and familiar that they may stifle and hijack the system's actual objective.

Optimizing model properties is what intelligence is about, but it is seldom the goal. A good goal will show how improving model properties contributes to having the desired impact on users and the business. A good goal will give guidance on how much model property optimization is worth.

Layering Goals

Success in an Intelligent System project is hard to define with a single metric, and the metrics that define it are often hard to measure. One good practice is to define success on different levels of abstraction and have some story about how success at one layer contributes to the others. This doesn't have to be a precise technical endeavor, like a mathematical equation, but it should be an honest attempt at telling a story that all participants can get behind.

For example, participants in an Intelligent System might:

- On an hourly or daily basis optimize model properties.
- On a weekly basis review the user outcomes and make sure changes in model properties are affecting user outcomes as expected.
- On a monthly basis review the leading indicators and make sure nothing has gone off the rails.
- On a quarterly basis look at the organizational objectives and make sure the Intelligent System is moving in the right direction to affect them.

Revisit the goals of the Intelligent System often during the course of the project.

Because things change.

Ways to Measure Goals

One reason defining success is so hard is that measuring success is harder still.

How the heck are we supposed to know how many passwords abusers got with their phishing pages?

When we discuss intelligent experiences in Part II of this book we will discuss ways to design intelligent experiences to help measure goals and get data to make the intelligence better. This section introduces some basic approaches. Using techniques like these should allow more flexibility in defining success.

Waiting for More Information

Sometimes it is impossible to tell if an action is right or wrong at the time it happens, but a few hours or days or weeks later it becomes much easier. As time passes you'll usually have more information to interpret the interaction. Here are some examples of how waiting might help:

- The system recommends content to the user, and the user consumes it completely—by waiting to see if the user consumes the content, you can get some evidence of whether the recommendation was good or bad.
- The system allows a user to type their password into a web page—by waiting to see if the user logs in from eastern Europe and tries to get all their friends to install malware, you can get some evidence if the password was stolen or not.

Waiting can be a very cheap and effective way to make a success criterion easier to measure, particularly when the user's behavior implicitly indicates success or failure.

There are a couple of downsides.

First, waiting adds latency. This means that waiting might not help with optimizing, or making fine-grained measurements.

Second, waiting adds uncertainty. There are lots of reasons a user might change their behavior. Waiting gives more time for other factors to affect the measurement.

A/B Testing

Showing different versions of the feature/intelligence to different users can be a very powerful way to quantify the effect of the feature.

Imagine giving half the users an intelligent experience and the other half a stubbed-out (simple, default) experience. Maybe half the users of the Internet toaster get a toast time of one minute no matter what settings they use or what they put into the toaster. The other half get a toast time that is determined by all the fanciest intelligence you can find.

If users who got the stubbed experience are just as happy/engaged/effective at toasting as the ones who got the full experience—you've got a problem.

A/B testing can be difficult to manage, because it involves maintaining multiple versions of the product simultaneously.

It can also have trouble distinguishing small effects. Imagine testing two versions of the intelligent toaster, one that toasts 1 minute no matter what, and one that toasts 61 seconds no matter what. Is one of them better than the other? Maybe, but it will probably take a long time (and a lot of observations of user interactions) to figure out which.

A/B testing is a great way to make sure that large changes to your system are positive, but it is troublesome with day-to-day optimization.

Hand Labeling

Sometimes a computer can't tell if an action was aligned with success, but a human can. You can hire some humans to periodically examine a small number of events/interactions and tell you if they were successful or not. In many cases, this hand labeling is easy to do, and doesn't require any particular skill or training.

In order to hand-label interactions, the Intelligent System needs to have enough telemetry to capture and replay interactions. This telemetry must contain enough detail so a human can reliably tell what happened and whether the outcome was good or not (while preserving user privacy). This isn't always possible, particular when it involves having to guess what the user was trying to do, or how they were feeling while they were doing it.

But it never hurts to look at what your system is doing and how it is affecting users. The scale can be small. It can be cheap. But it can also be very useful.

Asking Users

Perhaps the most direct way to figure out if something is succeeding or not is to ask the user. For example, by building feedback mechanisms right into the product:

- The user is shown several pieces of content, selects one. The system pops up a dialog box asking if the user was happy with the choices.
- A self-driving car takes a user to their destination, and as the user is getting out it asks if the user felt safe and comfortable during the trip.
- The toaster periodically asks, "is that toasted just the way you like it?" (And yeah, that could be pretty darn creepy, especially if it does it when you are home alone after dark and the lights aren't on.)

A couple of things to keep in mind:

Users don't always have the answer. For example, asking someone "did you just give your password to a criminal?" might not be very effective—and it might scare a lot of people for no good reason.

Users might not always feel like giving an answer to all of your questions. This will introduce bias. For example, users who are very engaged in their task might not pause to consider a survey, even though they are getting a good outcome.

Users will get sick of being asked questions. This type of feedback should be used sparingly.

But sometimes, asking just .1% of users a simple question once per month can unlock a lot of potential in helping you know if your Intelligent System is succeeding.

Decoupling Goals

Some things are hard to measure directly but can be broken down into simpler pieces, and the pieces can be measured (perhaps using some of the techniques from this section) and then stitched together into an estimate of the whole.

For example, consider the phishing example. The number of credentials lost to phishers can be decoupled into the following:

- The number of your users who visit phish sites (which is estimated by waiting for user reports to identify phishing sites and combining with traffic telemetry to estimate the number of visits).

- The percent of users who type their passwords into phish sites when the system doesn't provide a warning (which is estimated by doing user studies).
- The percent of users who type their passwords into phish sites when the system does provide a warning (which is estimated using telemetry on how many users dismiss the warning and proceed to type into the password box on known phishing sites).

Multiply these together and you have an estimate of a pretty good potential goal for the Intelligent System.

Decoupling is particularly useful when it identifies critical, measurable sub-problems and shows how they chain together with other clear sub-problems to reach overall success.

Keeping Goals Healthy

Sometimes a goal should change. But changing goals is difficult, particularly when people have organized around them. There might be a lot of inertia, many opinions, and no good answers. Having a process for reviewing goals and adapting them is a very good idea. Goals might change if any of the following happen:

- A new data source comes on-line and shows some assumption was wrong.
- A part of the system is functioning very well, and goals around further improvement to it should be changed (before they take on a life of their own, resulting in investing in the wrong things).
- Someone comes up with a better idea about how to connect the Intelligent System's work to actual customer impact.
- The world changes and previous goal no longer reflects success.

And even if the goals don't have to change, it's good to get everyone together once and a while and remind yourselves what you all are trying to accomplish together.

Summary

Having goals is crucial to success in an Intelligent System. But goals are hard to get right. Effective goals should:

1. Communicate the desired outcome
2. Be achievable
3. Be measurable

Goals can be very abstract (like organizational objectives). They can be less abstract (like leading indicators). They can be sort of concrete (like user outcomes). Or they can be super concrete (like model properties).

An effective set of goals will usually tie these various types of goals together into a story that clearly heads toward success.

Most Intelligent Systems will contribute to organizational objectives and leading indicators, but the core work of day-to-day improvement will be focused on user outcomes and model properties.

Goals can be measured through telemetry, through waiting for outcomes to become clear, by using human judgment, and by asking users about their experiences.

And did I mention—goals are hard to get right. They will probably take iteration.

But without effective goals, an Intelligent System is almost certainly doomed to waste time and money—to fail.

For Thought

After reading this chapter, you should:

- Understand the ways you can define success for an Intelligent System, and how to measure whether success is being achieved.
- Be able to define success on several levels of abstraction and tell the story of how the different types of success contribute to each other.

You should also be able to answer questions like these:
Consider your favorite hobby that might benefit from an Intelligent System.

- What organizational objective would the Intelligent System contribute to for its developers?
- What leading outcomes would make the most sense for it?
- What are the specific user outcomes that the Intelligent System would be tracked on?
- Which way would you measure these? Why?

PART II

Intelligent Experiences

Chapters 5-10 explain how to connect intelligence with users to achieve an Intelligent System's objectives. This part discusses the pitfalls and challenges of creating user experiences based on models and machine learning, along with the properties of a successful intelligent experience. It explains how to adapt experiences to get data to grow the system. And it gives a framework for verifying that an intelligent experience is behaving as intended.

CHAPTER 5

The Components of Intelligent Experiences

At the core of every Intelligent System is a connection between the intelligence and the user. This connection is called the *intelligent experience*. An effective intelligent experience will:

- **Present the intelligence to the user** by choosing how they perceive it, balancing the quality of the intelligence with the forcefulness of the experience to engage users.
- **Achieve the system's objectives** by creating an environment where users behave in the ways that achieve the system's objectives—without getting upset.
- **Minimize any intelligence flaws** by reducing the impact of small errors and helping users discover and correct any serious problems that do occur.
- **Create data to grow the system** by shaping the interactions so they produce unbiased data that is clear, frequent, and accurate enough to improve the system's intelligence.

Achieving these objectives and keeping them balanced as the Intelligent System evolves is as difficult as any other part of creating an Intelligent System, and it is critical for achieving success.

This chapter will introduce these components of intelligent experiences. Following chapters will explore key concepts in much more detail.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_5

Presenting Intelligence to Users

Intelligence will make predictions about the world, the user, and what is going to happen next. The intelligent experience must use these predictions to change what the user sees and interacts with. Imagine:

- The intelligence identifies 10 pieces of content a user might like to explore. How should the experience present these to the user?
- The intelligence thinks the user is making an unsafe change to their computer's settings. What should the experience do?
- The intelligence determines the user is in a room where it a little bit too dark for humans to see comfortably. How should the automated home's experience respond?

The experiences in these examples could be passive, giving gentle hints to the user, or they could be forceful, flashing big red lights in the user's face. They might even automate things. Choosing the right balance is one of the key challenges when building intelligent experiences.

An intelligent experience might:

- **Automate:** By taking an action on the user's behalf. For example, when the intelligence is very sure the user is asleep, the experience might automatically turn down the volume on the television.
- **Prompt:** By asking the user if an action should be taken. For example, when the intelligence thinks the user might have forgotten their wife's birthday, the experience might ask: "Would you like me to order flowers for your wife's birthday?"
- **Organize:** By presenting a set of items in an order that might be useful. For example, when the intelligence thinks the user is on a diet, the experience might organize the electronic-menu at their favorite restaurant by putting the healthiest things the user might actually eat on top (and hiding everything that is a little too delicious).
- **Annotate:** By adding information to a display. For example, when the intelligence thinks the user is running late for a meeting, the experience might display a little flashing running-man icon on the corner of the screen.

Or an intelligent experience might use some combination of these methods, for example annotating when the issue is uncertain but automating when something bad is very likely to happen soon.

The challenge is that the experience should get the most value out of the intelligence *when the intelligence is right*, and it should cause only an acceptable amount of trouble *when the intelligence makes a mistake*.

Any Intelligent System (regardless of the intelligence) can use any of these approaches to experience. And recall, the intelligence in an Intelligent System will change over time, becoming better as more users use it (or worse as the problem changes). The right experience might change over time as well.

An Example of Presenting Intelligence

As an example, Let's consider a smart-home product that automates the lights in rooms of users' homes. A user installs a few sensors to detect the light level, a few sensors to detect motion, and a few computer-controlled light bulbs. Now the user is going to want to be amazed. They are going to want to impress their friends with their new toy. They are going to want to never have to think about a light switch again.

So how could it work? First, let's consider the intelligence. The role of the intelligence is to interpret the sensor data and make predictions. For example: given the recent motion sensor data and the recent light sensor readings, what is the probability a user would want to adjust the lights in the near future? The intelligence might know things like:

- If the room is dark, and someone just entered it, the lights are 95% likely to turn on soon.
- If the lights are on, and someone is leaving the room, the lights are 80% likely to turn off soon.
- If the lights are on, and no one has been around for four hours, the next person into the room is likely to yell at the kids about how much power used to cost in their day and about the meaning of respect and about groundings—and then turn off the lights.

So what should the experience for the light-automation system do?

One choice would be to completely automate the lights. If the intelligence thinks the user would like to change the lights soon, then go ahead and change them. This is a pretty forceful experience, because it automates the raw intelligence output with no accommodation for mistakes. Will it be right? Will it make the user happy?

Well, what if the user is watching a movie? The light-automation system's intelligence doesn't know anything about the TV. If the user is watching a movie, and the motion sensor detects them—like maybe they stretch or reach for their drink—the lights might flip on. Not a great experience. Or what if the user is sleeping? Will the lights flash on every-time they roll over? Or what if the user is having a romantic dinner?

In these cases, an experience that simply automates the lights is not likely to do the right thing. That's partly because the intelligence doesn't have enough information to make perfect decisions and partly because the intelligence wouldn't make perfect decisions even if it had all the information in the world (because intelligence makes mistakes).

So are we doomed? Should we scrap the product, go back to the drawing board?

Well, we could. Or we could consider some other ways of presenting the intelligence to users.

For example, instead of *automating* the lights, the experience might *prompt* the user when the intelligence thinks the lights should change. For example, if the system thinks it is too dark, it could ask (in a pleasantly computerized female voice), "Would you like me to turn on the lights?"

This might be better than automation, because the mistakes the systems makes are less irritating (a question in a nice voice instead of a suddenly-dark room). But will it be good enough? What if the intelligence makes too many mistakes? The voice is chiming in every few minutes, pleasantly asking: "Would you like me to mess up your lights now?"… "How about now?"… "Maybe now?"

The system would seem pretty stupid, asking the user if they would like to screw up their lighting multiple times per hour.

Another, even less intrusive, approach would expose the lighting information by providing *annotations*. For example, maybe a subtle "energy usage" warning on the user's watch if there are a couple of lights on that the intelligence thinks should be off. Or maybe a subtle "health warning" if the intelligence thinks the user is in a room that is too dark. If the user notices the warning, and decides it is correct, they can change the lights.

These different experiences might all be built on the same intelligence—they may have exactly the same prediction in any particular situation—but they would seem very different from the user's perspective.

So which one is right?

It depends on the system's goals and the quality of the intelligence. If the intelligence is excellent, automation might work. If the intelligence is new (and not too accurate yet), the annotations might make more sense. One of the key goals of the intelligence experience is to take the user's side and make something that works for them.

Achieve the System's Objectives

An effective intelligent experience will present intelligence in a way that achieves the Intelligent System's objective. The experience will be designed so users see the intelligence in ways that help them have good outcomes (and ways that help the business behind the Intelligent System achieve its objectives). Recall that the goals of an Intelligent System can include:

- **Organizational outcomes**: sales and profit.
- **Leading indicators**: engagement and sentiment.
- **User outcomes**: achieving user objectives.
- **Model Properties**: having accurate intelligence.

The job of the intelligent experience is to take the output of intelligence and use it to achieve user outcomes, to improve leading indicators, and to achieve organizational outcomes. This means that when the intelligence is right, the experience should push users toward actions that achieve an objective. And when the intelligence is wrong, the experience should protect users from having too bad an experience.

Consider the home lighting automation system. If the goal is to save power, what should the system do when it thinks the lights should be off?

- **Play a soft click at the light-switch**: If the user happens to hear it and wants to walk to the switch, they might turn the light off. Maybe.
- **Prompt the user**: If we ask whether the user would like the light off, they might say yes, or they might be in the middle of something (and miss the question).
- **Prompt the user and then automate after a delay**: Give the user a chance to stop the action, but then take the action anyway.
- **Automate the lights**: Just turn them off.
- **Shut down power to the house's main breaker**: Because, come on, the user is wasting power; they are a menace to themselves and to others…

These experience options are increasingly likely to get the lights to turn (and stay) off. Choosing one that is too passive will make the system less effective at achieving its objective. Choosing one that is too extreme will make the system unpleasant to use.

In fact, the experience is just as important as the intelligence in accomplishing the Intelligent System's objectives: an ineffective experience can mess up the system just as thoroughly as terribly inaccurate intelligence predictions.

An intelligent experience is free to pick and choose, to ignore unhelpful intelligence—whatever it takes to achieve the desired outcomes.

An Example of Achieving Objectives

Consider the automated light system again. Imagine two possible objectives for the product:

1. To minimize the amount of power consumed.
2. To minimize the chance of a senior citizen tripping over something and getting hurt.

Everything else is the same—the intelligence, the sensors, the way the system is implemented—only the objectives are different. Would the same experience be successful for both of these objectives? Probably not.

The system designed to conserve power might never turn lights on, even if the intelligence thinks having a light on would be appropriate. Instead, the experience might rely on users to use the switch when they want a light on, and simply turn lights off when it thinks the user doesn't need the light any more.

The system designed to avoid senior-citizen-tripping might behave very differently. It might turn on lights whenever there is any chance there is a user in the room, and leave them on until it is very certain the room is empty.

These experiences are quite different, so different they might be sold as totally different products—packaged differently, marketed differently, priced differently—and they might each be perfectly successful at achieving their overall objectives. And they might both use (and contribute to) exactly the same underlying intelligence.

Minimize Intelligence Flaws

Intelligence makes mistakes. Lots of them. All different types of mistakes. An effective intelligent experience will function despite these mistakes, minimizing their impact and making it easy for users to recover from them.

When designing an intelligent experience it is important to understand all of the following:

1. What types of mistakes the intelligence will make.
2. How often it will make mistakes.
3. What mistakes cost the user (and the organization):
    a. If the user notices the mistake and tries to correct it.
    b. If the user never notices that the mistake happened.

Then the intelligent experience can decide what to do about the mistakes. The experience can:

1. Stay away from bad mistakes by choosing not to do things that are too risky.
2. Control the number of interactions the user will have with the intelligence to control the number of mistakes the user will encounter.
3. Take less forceful actions in situations where mistakes are hard to undo (for example, prompting the user to be sure they want to launch the nuclear missile, instead of launching it automatically).
4. Provide the user with feedback about actions the system took and guidance on how to recover if something is wrong.

These techniques make an Intelligent System safer. They also reduce the potential of the Intelligent System by watering down interactions, and by demanding the user pay more attention to what it is doing. Creating a balance between achieving the objective and controlling mistakes is a key part of designing effective intelligent experiences.

Create Data to Grow the System

Intelligence needs data to grow. It needs to see examples of things happening in the world, and then see the outcomes that occurred. Was the interaction good for the user? For the business? Was it bad? Intelligence can use these examples of the world and of the outcomes to improve over time.

The experience plays a large role in making the data that comes from the Intelligent System valuable. Done right, an experience can produce large data sets, perfectly tailored for machine learning. Done wrong, an experience can make the data from the system useless.

An effective intelligent experience will interact with users in clear ways where the system can know:

1. The context of the interaction.
2. The action the user took.
3. The outcome.

Sometimes the experience can make this interaction completely implicit; that is, the experience can be designed so elegantly that the user produces the right type of data simply by using the system.

At other times the experience may require users to participate explicitly in creating good data (maybe by leaving ratings). We will discuss techniques and tradeoffs for getting good data from intelligent experiences in great detail in a later chapter.

An Example of Collecting Data

Consider the automated-light example.

The intelligence will examine the sensors and make a decision about what the lights should do. The experience will act on this intelligence, interacting with the user in some way. Then we need to know—was that interaction correct?

What if the lights were on, but the system thought they should be off? How would we know if the intelligence is right or not?

We might be right if:

- The experience automatically turns off the lights and the user doesn't turn them back on.
- The experience prompts the user to turn off the lights, and the user says yes.
- The experience provides a power warning, and the user turns off the lights.

We might be wrong if:

- The experience automatically turns off the lights and the user immediately turns them back on.
- The experience prompts the user to turn off the lights, but the user says no.
- The experience provides a power warning, but the user doesn't turn off the lights.

But this feedback isn't totally clear. Users might not bother to correct mistakes. For example:

- They might have preferred the lights remain on, but have given up fighting with the darn auto-lights and are trying to learn to see in the dark.
- They might see the prompt to turn the lights off, but be in the middle of a level on their favorite game and be unable to respond.
- They might have put their smart watch in a drawer and so they aren't seeing any power warnings any more.

Data is best when outcomes are clear and users have an incentive to react to every interaction the experience initiates. This isn't always easy, and it may come with some tradeoffs. But creating effective training data when users interact with intelligent experiences is a key way to unlock the value of an Intelligent System.

Summary

This chapter introduced intelligent experiences. The role of an intelligent experience is to present intelligence to users in a way that achieves the system's objectives, minimizes intelligence flaws, and creates data to improve the intelligence.

Presenting intelligence involves selecting where it will appear, how often it will appear, and how forcefully it will appear. There are many options, and selecting the right combination for your situation is critical to having success with Intelligent Systems.

With the right experiences, a single intelligence can be used to achieve very different goals.

Experience can control the cost of mistakes by reducing their forcefulness, by making them easier to notice, by making them less costly, and by making them easier to correct.

An effective experience will support improving the intelligence. This can be done by creating interactions with users that collect frequent and unbiased samples of both positive and negative outcomes.

The intelligent experience must be on the user's side and make them happy, engaged, and productive no matter how poor and quirky the intelligence might be.

For Thought…

After reading this chapter, you should:

- Understand how user experience and intelligence come together to produce desirable impact.
- Know the main goals of the intelligent experience and be able to explain some of the challenges associated with them.

You should be able to answer questions like these:

- Consider your interactions with Intelligent Systems. What is the most forceful use of intelligence you have experienced? What is the most passive?
- Name three ways you've provided data that helped improve an intelligent experience. Which was the most effective and why?

CHAPTER 6

Why Creating Intelligent Experiences Is Hard

This chapter explores some ways that experiences based on intelligence are different from more traditional experiences. And here is the bottom line: intelligence makes mistakes.

Intelligence is going to make mistakes; to create effective intelligent experiences, you are going to have to get happy with that fact. You are going to have to embrace mistakes. You are going to have to make good decisions about how to work with and around the mistakes.

The mistakes intelligence makes:

- Aren't necessarily intuitive.
- Aren't the same from day to day.
- Aren't easy to find ahead of time.
- Aren't possible to fix with "just a little more work on intelligence."

This chapter is about the mistakes that intelligence makes. Its goal is to introduce you to the challenges you'll need to address to create effective intelligent experiences. Understanding the ways intelligence makes mistakes and the potential trade-offs will help you know what to expect, and, more importantly, how to design intelligent experiences that give users the best possible outcomes—and achieve the system's objectives—despite the mistakes that intelligence makes.

Intelligence Make Mistakes

Intelligent Systems make mistakes. There is no way around it. The mistakes will be inconvenient, and some will be actually quite bad. If left unmitigated, the mistakes can make an Intelligent System seem stupid; they could even render an Intelligent System useless or dangerous.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_6

Here are some example situations that might result from mistakes in intelligence:

- You ask your phone to order pizza and it tells you the population of Mumbai, India.
- Your self-driving car starts following a road that doesn't exist and you end up in a lake.
- Your smart-doorbell tells you someone is approaching your front door, but upon reviewing the video, you realize no one is there.
- You put a piece of bread in your smart toaster, come back five minutes later, and find that the bread is cold—the toaster hadn't done anything at all.

These types of mistakes, and many others, are just part of the cost of using intelligence, particularly when using intelligence created by machine learning.

And these mistakes are not the fault of the people producing the intelligence. I mean, I guess the mistakes could be their fault—it's always possible for people to be bad at their jobs—but even people who are excellent—world class—at applied machine learning will produce intelligences that make mistakes.

So one of the big roles of experience in Intelligent Systems is to present the intelligence so it is effective when it is right, and so the mistakes it makes are minimized and are easy to recover from.

Consider that an intelligence that is right 95% of the time makes a mistake one out of every twenty interactions. And 95% is a very high accuracy. If your intelligence-producers get to 95% percent they are going to feel like they've done a great job. They are going to want to get rewarded for their hard work, and then they are going to want to move on and work on other projects. They aren't going to want to hear you telling them the intelligence isn't accurate enough.

But in a system that has a million user interactions per day, 95% accuracy results in 50,000 mistakes per day. That's a lot of mistakes. If the mistakes cost users time or money, that could add up to a real problem.

Another useful way to think about mistakes is how many interactions a user has between seeing a mistake. For example, a user with 20 interactions per day with a 97% accurate intelligence would expect to see 4.2 mistakes per week.

Is this a disaster?

It depends on how bad the mistakes are and what options the user has to recover.

But consider the alternative to building systems that make mistakes—that is, to demand a perfect intelligence before building a product around it. Perfection is very expensive, in many cases impossible. And perfection isn't needed to get real value from Intelligent Systems—sitting around waiting for perfection is a great way to miss out on a lot of potential. Consider these examples:

- Speech recognition isn't perfect, it probably never will be.
- Search engines don't always return the right answer.
- Game AI is commonly ridiculed for running in circles and shooting at walls.
- Self-driving cars have accidents.

But all of these (somewhat) inaccurate intelligences are part of products that many of us use every day—that make our lives better. Mitigating mistakes is a fundamental activity in producing intelligent experiences.

Intelligence Makes Crazy Mistakes

At the risk of belaboring the point, intelligences make crazy mistakes. Wild, inexplicable, counter-intuitive mistakes.

Consider—if a human expert is right 99% of the time, you might expect the mistakes they make to be close to correct. You assume this expert pretty much understands what is going on, and if they make a mistake it is probably going to be consistent with a rational view of how the world works.

Machine learning and artificial intelligence aren't like that.

An artificial intelligence might be right 99.9% of the time, and then one out of a thousand times say something that is right out of bat-o-bizarro land. For example:

- A system that recommends music might recommend you 999 rock songs, and then recommend you a teen-pop track.
- A smart toaster might toast 99 pieces of bread perfectly, and decide the 100th piece of bread doesn't need any toasting—no matter what, no heat.
- A system that reads human emotion from images might correctly identify 999 people as happy, then it might see my face and say I am sad no matter how big of an idiot-grin I slap on.

These crazy mistakes are part of working with Intelligent Systems.

And even crazy mistakes can be mitigated, recovered from, and minimized. But to design experiences that work with intelligence you should throw out pre-conceived notions of how a rational thing would act. Instead, you will need to develop intuition with the intelligence you are working with. Use it. See what tends to confuse it and how. Think of ways to support it.

One instinct is to talk to the people making the intelligence and say, “Come on, this is crazy. Fix this one mistake. It is killing us!”

That’s fine. They can probably change the intelligence to fix the mistake that’s bugging you.

But fixing one mistake usually introduces a new mistake somewhere else. That new mistake is likely just as crazy. And you won’t know when or how the new mistake will show up. Fixing obvious mistakes isn’t always the right thing to do; in fact, playing whack-a-mole with prominent mistakes can be detrimental. Sometimes it’s better to let the intelligence optimize as best it can, then support the intelligence by covering over mistakes with elegant experiences.

Intelligence Makes Different Types of Mistakes

And even though you may be sure I’m belaboring the point by now, Intelligent Systems make different types of mistakes; for example:

- The smart-doorbell might say someone is approaching the door when no one actually is; or it might fail to notify its user when someone is approaching the door.
- The smart toaster might undercook bread; or it might overcook it.
- A speech recognition system might incorrectly interpret what the user said; or it might refuse to try to guess what the user said.

Three main types of mistakes here are:

1. Mistaking one situation for another situation.
2. Estimating the wrong value (for example, the correct toast time).
3. Being too confused (or conservative) to say anything at all.

Perhaps the simplest form of this is confusing one situation for another, as in the smart doorbell example. There are four possible outcomes for the intelligence of a system like this (Figure 6-1):

- **True Positive**—When there *is someone standing at the door* and the intelligence thinks there *is someone standing at the door*, it is called a true positive (this is a correct answer, not a mistake).
- **True Negative**—When there *is not someone standing at the door* and the intelligence thinks there *is no one there*, it is called a true negative (this is another type of correct answer, not a mistake).
- **False Positive**—When there *is not someone standing at the door* but the intelligence thinks there *is someone standing at the door*, it is called a false positive (this is a mistake).
- **False Negative**—When there *is someone standing at the door* but the intelligence thinks there *is no one there*, it is called a false negative (this is another type of mistake) .

		The Intelligence says some is	
		There	**Not There**
Someone Actually is	**There**	**True Positive**	**False Negative**
	Not There	**False Positive**	**True Negative**

Figure 6-1. *Different types of mistakes*

It's often possible for the system's intelligence to be tuned to control the type of mistakes it makes, making more of one type of mistake and fewer of the other. (Usually this involves tuning a threshold on the probability output by the models that drive the intelligence.)

For example, when designing the smart-doorbell experience, is it better to have the system alert the user when there is no one at the door, or to fail to alert the user when someone is at the door?

What if you could get rid of three occurrences of the first type of mistake at the cost of adding one of the second type? Would that be a better trade-off for the user? Would it be more helpful in achieving the systems objectives?

It will depend on the experience. How much does each type of mistake irritate the user? Which type of mistake is easier to hide? Which type of mistake gets in the way of achieving the system's objectives more? And how many times per day/month/year will a user have to experience each one of these mistakes? Will they become fatigued and stop responding to alerts from the system at all? Or will they come to distrust the system and turn it off?

Other examples of trade-offs include:

- Trading off underestimates for overestimates. For example, when the smart toaster predicts how long to toast for, would it be better to over-toast by 5% or to under-toast by 5%?
- Trading off the number of options the intelligence gives. For example, in a recommender system, the system might be pretty sure the user wants to read one of 15 different books. Would it be better to let the user choose from the top 5? Or to display all 15?
- Trading off between the intelligence saying something and saying nothing. For example, in a computer vision system, the intelligence might be 90% sure it knows who is in an image. Would it be better to have the intelligence label a face in an image with a name that is going to be wrong 1 out of 10 times? Or would it better to have the system do nothing? Or ask for the user to help?

Intelligence Changes

Traditional user experiences are deterministic. They do the same thing in response to the same command. For example, when you right click on a file, a menu comes up, every time (unless there is a bug). Deterministic is good, for many reasons.

Intelligent Systems improve themselves over time. An action that did one thing yesterday might do a different thing today. And that might be a very good thing. Consider:

- Yesterday you searched for your favorite band and got some OK articles. Today you search for your favorite band again and you get newer, better information.
- Yesterday you cooked some popcorn in your microwave and a few of the pieces burned. Today you cook popcorn and the microwave doesn't cook it quite so long, so the popcorn comes out perfectly.
- Yesterday your navigation app directed you drive through a construction zone and your commute took an hour longer than usual. Today the app directs you on an alternate route and you arrive at work happier and more productive.

These are examples of positive change, and these types of change are the goal for Intelligent Systems—from the user's perspective the system works better today than it did yesterday. If something is a bit off right now, that's no problem—the user can trust that the system will improve and that problems will decrease over time.

That's the goal. But change can be disruptive too. Even change that is positive in the long run can leave users feeling confused and out of control in the short term. Imagine the following:

- Yesterday you were taking notes with a smart-pen, and the ink was a nice shade of blue. But overnight the pen connected to its service and the service—optimizing for something you may or may not ever know—decided to tweak the shade of blue. Today the notes you take are in a different color.
- Yesterday you went to the local convenience store and bought a big, cool cup of your favorite soda, drank and enjoyed it. Overnight the soda machine connected to its service and tweaked the recipe. Today you bought another cup of the soda and you don't like is quite as much.

Are these changes good? Maybe. Maybe they improve some very important properties of the businesses involved. Maybe you'll realize you actually do like the changed behavior—eventually—even though it bothered you at first.

Or maybe the changes are simply disastrous.

When dealing with changing intelligence, some things to consider are:

1. Can the **rate of change be controlled**? This might be done by putting some constraints on the intelligence producers. For example, maybe the intelligence changes its predictions on no more than 5% of interactions per day. Taken too far, this could result in static intelligence (and eliminate much of the potential of intelligence improving over time), but some simple constraints can help a lot.

2. Can the experience **help the user navigate through change?** Perhaps it does this by letting them know a change has happened. Or maybe by keeping the old behavior but offering the user the changed behavior as an option. For example, the smart-pen could let the user know it has found a new shade of ink that makes notes 5% more useful, and offer to make the change if the user wants or keep the ink the old way if the user prefers that.

3. Can the experience **limit the impact of the changing portions** of the system, while still allowing them to have enough prominence to achieve impact? For example, by using a more passive experience with the parts of the intelligence that are changing the most?

The Human Factor

Intelligent experiences succeed by meshing with their users in positive ways, making users happier, more efficient, and helping them act in more productive ways (or ways that better align with positive business outcomes).

But dealing with Intelligent Systems can be stressful for some users, by challenging expectations.

One way to think about it is this.

Humans deal with tools, like saws, books, cars, objects. These things behave in predictable ways. We've evolved over a long time to understand them, to count on them, to know what to expect out of them. Sometimes they break, but that's rare. Mostly they are what they are; we learn to use them, and then stop thinking so much about them.

Tools become, in some ways, parts of ourselves, allowing us powers we wouldn't have without them.

They can make us feel very good, safe, and comfortable.

Intelligent Systems aren't like this, exactly.

Intelligent Systems make mistakes. They change their "minds." They take very subtle factors into consideration in deciding to act. Sometimes they won't do the same thing twice in a row, even though a user can't tell that anything is different. Sometimes they even have their own motivations that aren't quite aligned with their user's motivations.

Interacting with Intelligent Systems can seem more like a human relationship than like using a tool. Here are some ways this can affect users:

- **Confusion**: When the Intelligent System acts in strange ways or makes mistakes, users will be confused. They might want (or need) to invest some thought and energy to understanding what is going on.
- **Distrust**: When the Intelligent System influences user actions will the user like it or not? For example, a system might magically make the user's life better, or it might nag them to do things, particularly things the user feels are putting others' interests above theirs (such as by showing them ads).
- **Lack of Confidence**: Does the user trust the system enough to let it do its thing or does the user come to believe the system is ineffective, always trying to be helpful, but always doing it wrong?
- **Fatigue**: When the system demands user attention, is it using it well, or is asking too much of the user? Users are good at ignoring things they don't like.
- **Creep Factor**: Will the interactions make the user feel uncomfortable? Maybe the system knows them too well. Maybe it makes them do things they don't want to do, or post information they feel is private to public forums. If a smart TV sees a couple getting familiar on the couch, it could lower the lights and play some Barry White music—but should it?

Be conscious of how users feel about the intelligent experience. Manage their feeling by making good choices about what to present and how forceful to be.

Summary

Intelligence makes mistakes. No matter what, it will make mistakes. Complaining won't help. The mistakes aren't there because someone else is bad at their job. And chasing mistakes, playing whack-a-mole, is often (usually) detrimental.

The mistakes an intelligence makes can be quite counter intuitive. Even intelligence that is usually right can make really strange mistakes (ones that no human would ever make).

There are different types of mistakes. One common way to express them is as: False positives (saying something happened when it didn't) and false negatives (saying something didn't happen when it did). Intelligence can often make trade-offs between these types of mistakes (for example, by adding a few more false positives to get rid of a few false negatives).

Change can be disruptive to users, even when the change is positive in the long run. Intelligent Systems change, regularly. Helping users deal with the change is part of creating an effective Intelligent System.

Intelligent Systems can have negative effects on users, leading them to dislike and distrust the system.

Creating intelligent experiences is hard.

For Thought…

After reading this chapter you should:

- Understand that intelligence makes mistakes—it is one of the fundamental challenges that intelligent experiences need to address.
- Realize why these challenges are not intelligence bugs (which should be fixed) but are intrinsic to working with intelligence (and must be embraced).
- Have a sense of how users will experience the negative parts of an Intelligent System and some empathy to help create experiences to help users.

You should be able to answer questions like:

- What is the worst mistake you've seen an Intelligent System make?
- What could an intelligent experience have done to make the mistake less bad?
- For what product change did you have the wildest reaction, disliking it at first, but coming to like it more over time? Why?

CHAPTER 7

Balancing Intelligent Experiences

Designing a successful intelligent experience is a balancing act between:

1. Achieving the desired outcome
2. Protecting users from mistakes
3. Getting data to grow the intelligence

When the intelligence is right, the system should shine, creating value, automating something, giving the user the choices they need, and encouraging the safest, most enjoyable, most profitable actions. The experience should strongly encourage the user to take advantage of whatever it is the intelligence was right about (as long as the user isn't irritated and feels they are getting a good deal from the interaction).

When the intelligence is wrong, the system should minimize damage. This might involve allowing the user to undo any actions taken. It might involve letting the user look for more options. It might involve telling the user why it made the mistake. It might involve some way for the user to get help. It might involve ways for the user to avoid the particular mistake in the future.

The problem is that the experience won't know if the intelligence is right or wrong. And so every piece of experience needs to be considered through two lenses: what should the user do if they got there because the intelligence was right; and what should the user do if they got there because the intelligence was wrong.

This creates tension, because the things that make an Intelligent System magical (like automating actions without any fuss) are at odds with the things that let a user cope with mistakes (like demanding their attention to examine every decision before the intelligence acts).

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_7

There are five main factors that affect the balance of an intelligent experience:

- The ***forcefulness*** of the experience; that is, how strongly it encourages the user to do what the intelligence thinks they should.
- The ***frequency*** of the experience; that is, how often the intelligent experience tries to interact with the user.
- The ***value*** of the interaction when the intelligence is right; that is, how much the user thinks it benefits them, and how much it helps the Intelligent System achieve its goals.
- The ***cost*** of the interaction when the intelligence is wrong; that is how much damage the mistake does and how hard it is for the user to notice and undo the damage.
- The ***quality of the intelligence***; that is, how often the intelligence is right and how often it is wrong.

To create an effective intelligent experience you must understand these factors and how they are related. Then you must take the user's side and build an experience that effectively connects them to the intelligence. This chapter will explore these factors in more detail.

Forcefulness

An interaction is forceful if the user has a hard time ignoring (or stopping) it. An interaction is passive if it is less likely to attract the user's attention or to affect them. For example, a forceful experience might:

- Automate an action.
- Interrupt the user and make them respond to a prompt before they can continue.
- Flash a big, garish pop-up in front of the user every few seconds until they respond.

Forceful interactions are effective when:

- The system is confident in the quality of the intelligence (that it is much more likely to be right than it is to be wrong).
- The system really wants to engage the user's attention.
- The value of success is significantly higher than the cost of being wrong.
- The value of knowing what the user thinks about the intelligence's decision is high (to help create new intelligence).

A *passive experience* does not demand the user's attention. It is easy for the user to choose to engage with a passive experience or not. Passive experiences include:

- A subtle prompt that does not force the user to respond immediately.
- A small icon in the corner of the screen that the user may or may not notice.
- A list of recommended content on the bottom of the screen that the user can choose to click on or to ignore.

Passive interactions are effective when:

- The system is not confident in the quality of the intelligence.
- The system isn't sure the value of the intelligent interaction is higher than what the user is currently doing.
- The cost of a mistake is high.

One way to think about the forcefulness of an interaction is as an advertisement on a web page. If the ad pops over the web page and won't let you continue till you click it—that's a forceful experience. You'll probably click the stupid ad (because you have to). And if the ad isn't interesting, you'll be mad at the product that is being advertised, at the web page that showed you the ad, and at whoever programed a web browser stupid enough to let such an irritating ad exist.

On the other hand, if the ad is blended tastefully into the page you may not notice it. You may or may not click the ad. If the ad is for something awesome you might miss out. But you are less likely to be irritated. You might go back to the web page again and again. Over time you may spend more time on the web page and end up clicking far more ads if they are passive and tasteful than if they are forceful and garish.

Frequency

An intelligent experience can choose to interact with the user, or it can choose not to. For example, imagine a smart map application that is giving you driving directions. This smart map might be getting millions of data points about traffic conditions, accidents, weather conditions, which lights are red and which are green, and so on. Whenever you come to an intersection, the directions app could say "turn left to save 17 seconds," or "go faster to beat the next light and save 3 minutes," or "pop a U and go back to the last intersection and turn right LIKE I TOLD YOU LAST TIME and you can still save 2 minutes."

Or the app could choose to be more subtle, limiting itself to one suggestion per trip, and only making suggestions that save at least 10 minutes.

More frequent interactions tend to fatigue users, particularly if the frequent interactions are forceful ones. On the other hand, less frequent interactions have fewer chances to help users, and may be confusing when they do show up (if users aren't accustomed to them).

Some ways to control frequency are to:

1. *Interact whenever the intelligence thinks it has a different answer*. For example, fifteen seconds ago the intelligence thought the right direction was straight, now it has more information and thinks the right direction is right. And ten seconds later it changes its mind back to thinking you should go straight. Using intelligence output directly like this can result in very frequent interactions. This can be effective if the intelligence is usually right and the interaction doesn't take too much user attention. Or it could drive users crazy.

2. *Interact whenever the intelligence thinks it has a significantly different answer*. That is, only interact when the new intelligence will create a "large value" for the user. This trades off some potential value for a reduction in user interruption. And the meaning of "large value" can be tuned over time to control the number of interaction.

3. *Explicitly limit the rate of interaction*. For example, you might allow one interaction per hour, or ten interactions per day. This can be effective when you aren't sure how users will respond. It allows limiting the potential cost and fatigue while gaining data to test assumptions and to improve the Intelligent System.

4. *Interact whenever the intelligence thinks the user will respond*. This involves having some understanding of the user. Do they like the interactions you are offering? Do they tend to accept or ignore them? Are they doing something else or are they receptive to an interruption? Are they starting to get sick of all the interactions? Done well, this type of interaction mode can work well for users, meshing the Intelligent System with their style. But it is more complex. And there is always the risk of misunderstanding the user. For example, maybe one day the user has a guest in the car, so they ignore all the suggestions and focus on their guest. Then the intelligent experience (incorrectly) learns that the user doesn't like interruptions at all. It stops providing improved directions. The user misses out on the value of the system because the system was trying to be too-cute.

5. *Interact whenever the user asks for it*. That is, do not interact until the user explicitly requests an interaction. This can be very effective at reducing user fatigue in the experience. Allowing users to initiate interaction is good as a backstop, allowing the system to interact a little bit less, but allowing the users to get information or actions when they want them. On the other hand, relying too heavily on this mode of interaction can greatly reduce the potential of the Intelligent System—what if the user never asks for an interaction? How would the user even know (or remember) how to ask for an interaction?

And keep in mind that humans get sick of things. Human brains are very good at ignoring things that nag at them. If your experience interacts with a user too much, they will start to ignore it. Be careful with frequent interactions—less can be more.

Value of Success

Users will be more willing to put up with mistakes and with problems if they feel they are getting value. When an Intelligent System is helping with something obviously important—like a life-critical problem or saving users a large amount of money and time—it will be easier to balance the intelligent experience in a way that users like, because users will be willing to engage more of their time on the decisions the Intelligent

System is making. When the Intelligent System is providing smaller value—like helping users toast their bread or saving them a few pennies of power per day—it will be intrinsically harder to create a balanced intelligent experience that users feel is worth the effort to engage with. Users tend to find interactions valuable if they:

- Notice that something happened.
- Feel that the interaction solved a problem they care about, or believe it provided them some meaningful (if indirect) improvement.
- Can connect the outcome with the interaction; they realize what the interaction was trying to do, and what it did do.
- Trust that the system is on their side and not just trying to make someone else a buck.
- Think the system is cool, smart, or amazing.

It is possible for an interaction to leave users in a better situation and have zero perceived value. It is also possible for an interaction to leave the users in a worse situation, but leave the user thinking the Intelligent System is great. The intelligent experience plays a large role in helping users feel they are getting value from an Intelligent System.

When the intelligence is right, an effective intelligent experience will be prominent, will make users feel good about what happened, and will take credit where credit is due. But a savvy intelligent experience will be careful—intelligence won't always be right, and there is nothing worse than an intelligent experience taking credit for helping you when it actually made your life worse.

Interactions must also be valuable to the Intelligent System. In general, an interaction will be valuable to the Intelligent System if it achieves the system's objectives; for example, when it:

- Causes the user to use the Intelligent System more (*increases engagement*).
- Causes the user to have better feelings about the Intelligent System (*improves sentiment*).
- Causes the user to spend more money (*creates revenue*).
- Creates data that helps the Intelligent System improve (*grows intelligence*).

In general, experiences that explicitly try to make money and to get data from users will be in conflict with making users feel they are getting a good value. An effective intelligent experience will be flexible to make trade-offs between these over the life-cycle of the Intelligent System to help everyone benefit.

Cost of Mistakes

Users don't like problems; Intelligent Systems will have problems. A balanced intelligent experience will be sensitive to how much mistakes cost users and will do as much as possible to minimize those costs.

Mistakes have intrinsic costs based on the type of mistake. For example, a mistake that threatens human life or that costs a large amount of money and time is intrinsically very costly. And a mistake that causes a minor inconvenience, like causing a grill to be a few degrees colder than the user requests, is not so costly.

But most mistakes can be corrected. And mistakes that are easy to correct are less costly than ones that are hard (or impossible) to correct. An intelligent experience will help users *notice when there is a mistake* and it will provide good options for *recovering from mistakes*.

Sometimes the mistakes are pretty minor, in which case users might not care enough to know the mistake even happened. Or there may be no way to recover, in which case the Intelligent System might want to pretend nothing is wrong—no sense crying over spilt milk.

Knowing There Is a Mistake

The first step to solving a problem is to knowing there is a problem. An effective intelligent experience will help users know there is a mistake in a way that:

1. Doesn't take too much of the user's attention, especially when it turns out there isn't a mistake.
2. Finds the mistake while there is still time to recover from the damage.
3. Makes the user feel better about the mistake and the overall interaction.

Sometimes the mistakes are obvious, as when an intelligent experience turns off the lights, leaving the user in the dark. The user knows that something is wrong the instant they are sitting in the dark. But sometimes the mistakes are not obvious, as when an intelligent experience changes the configuration of a computer-system's settings without any user interaction. The user might go for years with the suboptimal settings and never know that something is wrong. Some options to help users identify mistakes include these:

- ***Informing the user*** when the intelligent experience makes a change. For example, the intelligent experience might automate an action, but provide a subtle prompt that the action was taken. When these notifications are forceful, they will demand the user's attention and give the user a chance to consider and find mistakes. But this will also fatigue users and should be used sparingly.
- ***Maintaining a log*** of the actions the intelligent experience took. For example, the "junk" folder in a spam filtering system is a log of messages the spam filter suppressed. This lets users track down problems but doesn't require them to babysit every interaction. Note that the log does not need to be complete; it might only contain interactions where the intelligence was not confident.

Recovering from a Mistake

Intelligent experiences can also help users *recover from mistakes*.

Some mistakes are easy to recover from—for example, when a light turns off, the user can go turn it back on. And some mistakes are much harder to recover from, as when the intelligent experience sends an email on the user's behalf—once that email is beeping on the recipient's screen there is no turning back.

The two elements of recovering from mistakes are: how much of the cost of the mistake can be recovered; and what the user has to do to recover from the mistake. To help make mistakes recoverable, an intelligent experience might:

1. Limit the scope of a decision, for example, by taking a partial action to see if the user complains.
2. Delay an action, for example, by giving the user time to reflect on their decision before taking destructive actions (like deleting all their files).

3. Be designed not to take destructive actions at all, for example by limiting the experience to actions that can be undone.

If the cost of mistakes is high enough, the experience might want to go to great lengths to help make mistakes recoverable.

When the user wants to recover from a mistake, the intelligent experience can:

1. Put an option to undo the action directly in the experience (with a single command).
2. Force the user to undo the action manually (by tracking down whatever the action did and changing the various parts of the action back one by one).
3. Provide an option to escalate to some support agent (when the action can only be undone by an administrator of the system).

The best mistakes are ones that can be noticed easily, that don't require too much user attention to discover, and that can be completely recovered from in a single interaction.

The worst are ones that are hard to find, that can only be partially recovered from, and where the user needs to spend a lot of time (and the Intelligent System needs to pay for support staff) to recover.

Intelligence Quality

The experience can't control the quality of the intelligence. But to make an effective intelligent experience you had better understand the quality of the intelligence in detail. When the intelligence is good, you'll want to be more frequent and forceful. When the intelligence is shaky you'll want to be cautious about when and how you interact with the user.

Mistakes come in many type. An intelligence might:

- Make many essentially *random mistakes*. For example, it might have the right answer 90% of the time and the wrong answer 10% of the time. In this case, the experience can be balanced for all users simultaneously.

- Make *focused mistakes*. For example, it might be more likely to have the right answer for some users than for others. For example, making more mistakes with users who have glasses than with users who do not. Focused mistakes are quite common, and are often hard to identify. When focused mistakes become a problem it may be necessary to balance the intelligent experience for the lowest common denominator—the user-category that has the worst outcomes (or to abandon some users and focus on the ones where the intelligence works well).

Intelligence quality is a life-cycle, and experience will need to adapt as the intelligence does. For example, early in development, before the system has real customers, the intelligence will generally be poor. In some cases, the intelligence will be so poor that the best experience results from hiding the intelligence from users completely (while gathering data).

As the system is deployed, and customers come, the system will gather data to help the intelligence improve. And as the intelligence improves, the experience will have more options to find balance and to achieve the Intelligent System's objectives. Keep in mind:

1. Better intelligence can support more forceful, frequent experiences.

2. Change is not always easy (even if the change comes from improved intelligence). Be careful to bring your users along with you.

3. Sometimes intelligence will change in bad ways, maybe if the problem gets harder or new types of users start using the Intelligent System (or if the intelligence creators make some mistakes). Sometimes the experience will need to take a step backwards to support the broader system. That's fine—life happens.

Summary

An effective intelligent experience will balance:

- The ***forcefulness*** of the interactions it has with users.
- The ***frequency*** with which it interacts with users.
- The ***value*** that successful interactions have for the user and for the Intelligent System.
- The ***cost*** that mistakes have for the user and for the Intelligent System.
- The ***quality of intelligence***.

Both the users and the Intelligent System must be getting a good deal. For users this means they must perceive the Intelligent System as being worth the time to engage with. For the Intelligent System this means it must be achieving objectives and getting data to improve.

When intelligence is good (very accurate), the intelligent experience can be frequent and forceful, creating a lot of value for everyone, and not making many mistakes.

When the intelligence is not good (or mistakes are very expensive), the intelligent experience must be more cautious about what actions it proposes and how it proposes them, and there must be ways for users to find mistakes and recover from them.

The frequency of interaction can be controlled by interacting:

- Whenever the intelligence changes its mind.
- Whenever the intelligence finds a large improvement.
- A limited number of times.
- Only when the system thinks the user will respond.
- Only when the user asks for an interaction.

Interacting more often creates more opportunity for producing value. But interacting can also create fatigue and lead to users ignoring the intelligent experience.

Users find value when they understand what is going on and think they are getting a good deal. The intelligent experience is critical in helping users see the value in what the intelligence is providing them.

Intelligence will be wrong, and to make an effective intelligent experience you must understand how it is wrong and work to support it. The quality of intelligence will change over the life-cycle of an Intelligent System and as it does, the intelligent experience can rebalance to create more value.

For Thought…

After reading this chapter you should:

- Know the factors that must be balanced to create an effective intelligent experience: one that is pleasant to use, achieves objectives and mitigates mistakes, and evolves as intelligence does.
- Know how to balance intelligent experiences by controlling the key factors of: forcefulness, frequency, value, cost, and understanding intelligence quality.

You should be able to answer questions like these:

- What is the most forceful intelligent experience you have ever interacted with? What is the most passive intelligent experience you can remember encountering?
- Give an example of an intelligent experience you think would be more valuable if it were less frequent. Why?
- List three ways intelligent experiences you've interacted with have helped you find mistakes. Which was most effective and why?

CHAPTER 8

Modes of Intelligent Interaction

There are many, many ways to create user experiences, and just about all of them can be made intelligent. This chapter explores some broad approaches to interaction between intelligence and users and discusses how these approaches can be used to create well-balanced intelligent experiences. These approaches include:

- ***Automating*** actions on behalf of the user.
- ***Prompting*** users to see if they want to take an action.
- ***Organizing*** the information a user sees to help them make better decisions.
- ***Annotating*** other parts of the user experience with intelligent content.
- ***Hybrids*** of these that interact differently depending on the intelligence.

The following sections will explore these approaches, providing examples and discussing their pros and cons.

Automate

An automated experience is one in which the system does something for the user without allowing the user to approve or to stop the action. For example:

- You get into your car on a Monday morning, take a nap, and wake up in your office parking lot.
- You lounge in front of your TV, take a bite of popcorn, and the perfect movie starts playing.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_8

- You log into your computer and your computer changes its settings to make it run faster.
- You rip all the light switches out of your house, hook the bulbs up to an Intelligent System, and get perfectly correct lighting for all situations without ever touching a switch or wasting one watt of power again.

These are great, magical, delightful… if they work. But in order for experiences like these to produce good outcomes, the intelligence behind them needs to be exceptionally good. If the intelligence is not good, automated intelligent experiences can be disastrous.

Automated experiences are:

- **Very forceful**, in that they force actions on the user.
- **Not always obvious**, in that they may require extra user experience to let the user know what is happening and why. This may reduce the value of the interactions and make mistakes harder to notice.
- **Difficult to get training data from**, in that users will generally only give feedback when mistakes happen. Sometimes the system can tie outcomes back to the automated action, but sometimes it can't. Automated systems usually need careful thought about how to interpret user actions and outcomes as training data. They often require additional user experiences that gather information about the quality of the outcomes.

Automation is best used when:

- The intelligence is very good.
- There is a long-term commitment to maintaining intelligence.
- The cost of mistakes is not too high compared to the value of a correct automation.

Prompt

Intelligence can initiate an interaction between the system and the user. For example:

- If the intelligence suspects a user is about to do something wrong, it might ask them if they are sure they want to proceed.
- If the intelligence thinks the user is entering a grocery store, it might ask if they would like to call their significant other and ask if there is anything they need.
- If the intelligence notices the user has a meeting soon and estimates they are going to be late, it might ask if the user would like to send a notice to the other participants.

These interactions demand the user's attention. They also allow the user to consider the action, make a decision if it is right or not, and approve or reject the action. In this sense, experiences based on prompting allow the user to act as a back-stop for the intelligence, catching mistakes before they happen. Interactions based on prompting are:

- **Variably forceful**. Depending on how the interaction is presented, it can be extremely forceful (for example with a dialog box that the user must respond to before progressing), or it can be very passive (for example, by playing a subtle sound or showing a small icon for a few moments).
- **Usually obvious**, in that the user is aware of the action that was taken and why it was taken. This helps the user perceive the value of the intelligent interaction. It also helps users notice and recover from mistakes. But frequent prompts can contribute to fatigue: if users are prompted too often, they will begin to ignore the prompts and become irritated, reducing the value of the Intelligent System over time.
- **Usually good to get training data from**, in that the user will be responding to specific requests. The Intelligent System will have visibility into exactly what is going on. It will see the context that led to the interaction. It will have the user consider the context and give input on the action. This allows the system to learn, so that it will know whether the prompt was good or bad and be able to improve the intelligence.

These types of interactions are often used when:

- The intelligence is unreliable or the system is missing context to make a definitive decision.
- The intelligence is good enough that the prompts won't seem stupid, and the prompts can be infrequent enough that they don't lead to fatigue.
- The cost of a mistake is high relative to the value of the action, so the system needs the user to take part in approving action or changing behavior.
- The action to take is outside the control of the system, for example when the user needs to get up and walk to their next meeting.

Organize

The intelligence can be used to decide what information to present to the user, and in what order. For example:

- If the intelligence thinks the user is querying for information about their favorite band, it might select the most relevant web pages to display.
- If the intelligence thinks the user is shopping for a camera, it might select camera-related ads to tempt the user.
- If the intelligence thinks the user is having a party, it might offer a bunch of upbeat 80s songs on the user's smart-jukebox.

These types of experiences are commonly used when there are many, many potential options, so that even "good" intelligence would be unlikely to narrow the answer down to one exactly correct choice.

For example, at any time there might be 50 movies a user would enjoy watching, depending on their mood, how much time they have, and who they are with. Maybe the user is 10% likely to watch the top movie, 5% likely to watch the second, and so on. If the system showed the user just the top choice, it would be wrong 90% of the time.

Instead, these types of systems use intelligence to pre-filter the possible choices down to a manageable set and then present this set in some browsable or eye-catching way to achieve their objectives.

Interactions that organize choices for the user are:

- **Not intrinsically forceful**, in that they don't take important actions for the user and they don't demand the user's attention. If the user experience presenting the organized information is big and jarring, the interaction might be perceived as forceful, but that is fully in the hands of the experience designer.
- **Somewhat obvious**, in that the user will see the choices, and might or might not feel that there was significant intelligence behind the ordering of the information. It is also sometimes challenging for users to find mistakes—if an option isn't presented, the user may not know whether the option was excluded because the system doesn't support the option (have the product, movie, song, and so on) or because the intelligence was wrong.
- **OK to get training data from**, in that it is easy to see what the user interacts with (when the system did something right), but it is harder to know when the system got something wrong (when it suppressed the option the user might have selected). This can lead to bias, in that the options users tend to see more often will tend to be selected more. These systems often require some coordination between intelligence and experience to avoid bias, for example by testing various options with users—the intelligence doesn't know if the user will like something or not, so the experience tries it out.

Interactions that organize information/options are best when:

- There are a lot of potential options and the intelligence can't reasonably detect a "best" option.
- The intelligence is still able to find some good options, these good options are a small set, and the user is probably going to want one of them.
- The problem is big and open-ended and so that users can't reasonably be expected to browse through all options and find things on their own.

Annotate

The intelligence can add subtle information to other parts of the experience to help the user make better decisions. For example:

- If the intelligence thinks the user has their sprinklers set to water for too long (maybe there is rain in the forecast), it can turn on a blinking yellow light on the sprinkler box.
- If the intelligence thinks there is a small chance the user's car will break down soon (maybe the engine is running a little hot), it might display a "get service" indicator on the dashboard.
- If the intelligence finds a sequence of characters that doesn't seem to be a word, it can underline it with a subtle red line to indicate the spelling might be wrong.

These types of experiences add a little bit of information, usually in a subtle way. When the information is correct, the user is smarter, can make better decisions, can initiate an interaction to achieve a positive outcome. When the information is wrong it is easy to ignore, and the cost of the mistake is generally small.

Interactions based on annotation are:

- **Generally passive**, in that they don't demand anything of the user and may not even be noticed. This can reduce the fatigue that can come with prompts. It can also lead to users never noticing or interacting with the experience at all.
- **Variably obvious**, in that the user may or may not know where the annotation came from and why it is intelligent. Depending on how prominent the user experience is around the annotation, the user might never notice it. Users may or may not be able to understand and correct mistakes.
- **Difficult to get training data from**, as it is often difficult for the system to know: 1) if the user noticed the annotation; 2) if they changed their behavior because of it; and 3) if the change in behavior was positive or negative. Interactions based on annotation can require additional user experience to understand what actions the user took, and to improve the intelligence.

Annotations work best when:

- The intelligence is not very good and you want to expose it to users in a very limited way.
- The Intelligent System is not able to act on the information, so users will have to use the information on their own.
- The information can be presented in a way that isn't too prominent but is easy for the user to find when they want it.

Hybrid Experiences

Intelligent experiences can be built by combining these types of experiences, using one type of experience in places where the intelligence is confident, and another type in places where the intelligence is not. For example, when the intelligence is really, really sure, the experience might automate an action. But when the intelligence is only sort of sure, the experience might prompt the user. And when the intelligence is unsure, the experience might annotate the user experience with some information.

An example of a hybrid intelligent experience is spam filtering. The ideal spam filter would delete all spam email messages before a user had to see them and would never, never, never delete a legitimate email. But this is difficult.

Sometimes the intelligence can be very sure that a message is spam, because some spammers don't try very hard. They put the name of certain enhancement pills right in their email messages without bothering to scramble the letters at all. They send their messages from part of the Internet known be owned by abusers. In these cases it is easy for a spam filter to identify these obvious spam messages and be very certain that they aren't legitimate. Most intelligent spam filters delete such obvious spam messages without any user involvement at all. Spam filtering systems have deleted billions of messages this way.

But some spammers are smart. They disguise their messages in ways designed to fool spam filtering systems—like replacing i's with l's or replacing text with images—and they are very good at tricking spam filters. When the intelligence encounters a carefully obscured spam email, it isn't always able to distinguish it from legitimate emails. If the spam filter tried to delete these types of messages it would make lots of mistakes and delete a lot of legitimate messages too. If the spam filter put these types of messages into the inbox, skilled spammers would be able to circumvent most filters. And so most spam filtering systems have a junk folder. Difficult messages are moved to the junk folder, where the user is able to inspect them and rescue any mistakes.

These are two forms of automation, one extremely forceful (deleting a message), and one less forceful (moving a message to a junk folder). The intelligent experience chooses between them based on the confidence of the intelligence.

But many spam filters provide even more experiences than this. Consider the case when the spam filter thinks the message is good, but it is a little bit suspicious. Maybe the message seems to be a perfect personal communication from an old friend, but it comes from a part of the Internet that a lot of spammers use. In this case the experience might put the message into the user's inbox but add a subtle message to the top of the message that says "Reminder—do not share personal information or passwords over e-mail."

In this example, the Intelligent System uses forceful automation where it is appropriate (deleting messages without user involvement). It uses less-forceful automation when it must (moving messages to a junk folder). And it uses some annotation occasionally when it thinks that will help (warnings on a few percent of messages).

Hybrid experiences are common in large Intelligent Systems. They are very effective when:

- The problem can be decoupled into clear parts, some of which are easy and some of which are hard.
- The intelligence problem is difficult and it needs many types of support from the experience.
- You want to reduce the amount of attention you demand from users and avoid asking them questions where the intelligence is certain, or where the intelligence is particularly unreliable. This can make the questions you do ask the user much more meaningful and helpful (both to the user and as training data).

Summary

There are many ways to present intelligence to users, but important ones include these:

- Automating actions
- Prompting the user to action

- Organizing information
- Adding information
- Hybrid experiences

Options like automating and prompting require very good intelligence (or low costs when there is a mistake). Other options are less visible to the user and can be good at masking mistakes when the intelligence is less reliable or the cost of a mistake is high.

Most large Intelligent Systems have hybrid experiences that automate easy things, prompt for high-quality interactions, and annotate for things they are less sure about.

For Thought…

After reading this chapter you should:

- Know the most common ways to connect intelligence to users.
- Have some intuition about which to use when.

You should be able to answer questions like these:

- Find examples of all the types of intelligent experiences in systems you interact with regularly (automation, prompting, organizing, annotating, and a hybrid experience).

Consider the intelligent experience you interact with most.

- What type of experience is it?
- Re-imagine it with another interaction mode (for example, by replacing prompts with automation or annotation with organization).

CHAPTER 9

Getting Data from Experience

Intelligence creators can work with all sorts of data, even crummy data. But when they have good data, their job is much easier—and the potential of the intelligence they can create is much greater. An ideal intelligent experience will control the interactions between users and the Intelligent System so that the record of those interactions makes it easy to create high-quality intelligence.

Before exploring ways you can craft intelligent experiences to get data from your users, let's consider the alternative: you could gather data yourself. Traditionally, machine learning systems have a data-collection phase. That is, if you want to build an intelligence for counting the number of cows in a picture, you'd go out and take a lot of pictures of cows. You'd travel from farm to farm, taking pictures of cows in different situations. You'd pose cows so you get pictures of their good sides, and their bad sides; their faces and their… other parts. You'd take pictures of cows behind bushes. You'd take pictures of cows laying down, running, grazing, sleeping, yawning, on sunny days, on cloudy days, at night… A traditional computer vision system might require tens of thousands of different pictures of cows. And it would get better with more pictures—hundreds of thousands or millions might help. And once you had all that data you'd need to *label* it. That is, you'd pay lots of people to look at all those cow pictures and draw circles around all the cows—fun!

This can get very expensive. Data collection and labeling can be the primary cost in traditional machine-learning systems. Because of this, there is a lot of value in designing experiences that leverage users, and their interactions with the Intelligent System, to produce data with the correct properties to create intelligence. Getting this right makes it possible to use intelligence and machine learning in many situations where it would have been prohibitively expensive using manual data collection.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_9

This chapter begins with an example of what it means to collect data from experience and how various approaches can affect the quality of the resulting data. We will then explore the properties that make data good for intelligence creation and some of the options for capturing user outcomes (or opinions) on the data.

An Example: TeamMaker

Let's consider an example: an Intelligent System designed to create teams for basketball leagues—we'll call it TeamMaker. Every player is entered into the system, along with some basic statistics: height, experience, positions they play, and maybe something about how well they shoot the ball. Then the system divides the players up into teams. If the teams are balanced, the league will be fun to play in. If the teams are imbalanced the league might be boring. One team will always win and everyone else will feel like losers. Feuds will begin, rioting in the streets, hysteria…

So we better get this right.

We need to build an experience that collects useful feedback about the teams the system creates, so the intelligence can improve (and suggest better teams) over time.

Simple Interactions

One approach is to let users correct the teams and use the corrections to improve the intelligence.

For example, TeamMaker might produce "suggested teams" as a starting point. These proposals could be presented to a human—the league commissioner—to correct any problems. Maybe the intelligence got it wrong and put all the best players on one team. Then the commissioner could intervene and move some players around. Each time the commissioner moves a player, TeamMaker gets training data to make the intelligence better. After a few seasons, the intelligence might be so good that the commissioner doesn't need to make any more changes.

But, to be honest, this basketball-team–creating system sounds pretty lame. Who wants to use a tool to make teams, and then have to make them by hand anyway? This approach could fail if commissioners don't provide any corrections to the teams, because:

- It sounds boring.
- They don't feel confident they can beat the machine.

- They actually don't understand how to play basketball, so they make corrections that make things worse.
- They make the corrections offline and don't bother to enter them into TeamMaker at all.

Slow or irregular interactions will limit the speed the intelligence can grow and limit its ultimate potential. A good experience will *set up interactions that provide obvious value to the user and that the user can do a good job at getting right*.

Making It Fun

Another approach would be to make the connection between usage and intelligence something that's more fun. Something users really want to do and are motivated to do right.

For example, TeamMaker could make the teams, but it could also support betting. Most people who bet will want to win (because most people care about money), so they will try to bet on the best team—and will avoid betting on obvious losers. So TeamMaker can use the distribution of bets to figure out if it did a good job of constructing teams. If the bets are heavily skewed toward (or away) from one of the teams, TeamMaker can learn it used the wrong factors to balance the teams. It can do better next time.

This betting-based-interaction is an improvement to the team-tweaking–based interaction system and should result in more data per season and more opportunities to improve the intelligence, and it should produce a better Intelligent System faster.

But the data from betting-based interaction will not be perfect. For example, some people might only bet for their own team, even if it is obvious they can't win. Or maybe the betters don't know anything about basketball and are betting to pump up some good-natured office rivalry. (Or betting is illegal or against your morals, so as a developer you don't want to use betting-based interaction with TeamMaker at all…) These types of problems can lead to bias, and bias can lead intelligence astray. When an intelligence learns from biased betting data the resulting teams will not be optimized for creating a competitive league, they will be optimized for… something else.

When users have a different objective than the Intelligent System the data from their interactions can be misleading. A good experience will *align usage with the intelligence to get unbiased observations of success.*

Connecting to Outcomes

Another approach would be to connect the intelligence directly to outcomes. No human judgment needed, just the facts. When a single team wins every single game, the intelligence knows it did something wrong. When all the games are close, the intelligence can learn it did something right.

For this approach to work, someone would have to be motivated to enter all the scores into TeamMaker. So maybe TeamMaker creates some fun features around this, like integrating with the betting, or leader-boards, or scheduling. People might crowd around their computers every time a game is completed, just to see how the standings have changed, who won money, and who lost pride.

As games are played and users enter statistics about which teams won, which games were close, and which were blowouts, TeamMaker has everything it needs to improve the intelligence. If there is a team that never loses, it can learn what is special about that team and avoid doing that in the future. And next season TeamMaker would do a better job, creating tournaments that are more likely to be balanced, and less likely to have an unstoppable (or a hopeless) team.

When the experience can *directly track the desired outcomes,* it can produce the most useful intelligence.

Properties of Good Data

In order to solve hard, large, open-ended, time-changing problems, you're going to need data—lots of data. But not just any data will do. To build intelligence, you're going to need data with specific properties. The best data will:

- Contain the context of the interaction, any actions that were taken, and the outcomes.
- Have good coverage of what your users want to do with the Intelligent System.
- Reflect real interactions with the system (and not guesses or surveys about how people might interact).
- Have few (or no) biases.

- Avoid feedback, where the intelligence influences the data, which influences intelligence, which influences data…
- Be large enough to be relevant to the problem. (Creating intelligence for difficult problems can require incredible amounts of data.)

Achieving all of these properties is not easy. In fact, you are unlikely to get the data you need to effectively create intelligence unless you explicitly address each of these properties in your design.

Context, Actions, and Outcomes

The basic requirement for creating intelligence from data is to know:

1. The context of what was going on when the intelligence was invoked.
2. Any actions that were taken as a result.
3. The outcomes of the interaction, and specifically if the outcomes were positive or negative.

For example:

- A self-driving car needs to know what all the sensors on the car see (the context), the things a human driver might do in that situation (the actions), and whether the car ends up crashing or getting honked at (the outcome).
- A book-recommending system needs to know what books the user has already read and how much they enjoyed them (the context), what books might be recommended to that user, whether the user purchased any of the books or not (the actions), and which of the books the user ended up liking (the outcomes).
- An anti-malware system needs to know what file the user downloaded and where they got it from (the context), whether they installed it or not (the action), and whether their machine ended up infected or not (the outcome).

An ideal intelligent experience will create situations that have enough context to make good decisions, both for the user and for the intelligence. For example: if the automatic car doesn't have any sensors other than a speedometer, then it doesn't help to know what the user did with the steering wheel—no intelligence could learn how to steer based on the speed alone.

Good Coverage

The data should contain observations of all the situations where the intelligence will need to operate. For example:

- If the system needs to automate lights, the data should contain the context, actions, and outcomes of controlling lights:
  - During the day.
  - At night.
  - In the winter.
  - In the summer.
  - During daylight savings time.
  - With lights that are heavily used.
  - With lights that are rarely used.
  - And so on.
- If the lights need to work in 50 different countries around the world, the data should contain observations from those 50 countries in all of these situations.
- If the system needs to work in a mineshaft, the data should contain observations of lights being used in a mineshaft.
- If the system needs to work on a space station, the data should contain observations of lights being used in a space station.

An intelligence operating in a situation it was not trained (or evaluated) on is likely to make mistakes, crazy ones.

Intelligent Systems will be expected to work in new contexts over their lifetime. There will always be new books, movies, songs, web pages, documents, programs, posts, users, and so on. An effective intelligent experience will be able to put users into these new contexts with confidence that the mistakes made while collecting data will have low cost and the value of the collected data will be high.

Real Usage

The best data will come from users doing things they actually care about. For example:

- A user driving in a car simulator might make decisions they wouldn't make if they were driving a real car (when their life is on the line).
- A user telling you what books they like might talk about literary classics (because they are trying to impress you) but they might never actually read those books.
- A user might give very different answers when asked if a file is safe to install on a lab computer or on their mother's computer.

Connecting real users to interactions they care about ensures the resulting data is honest, and that building intelligence from it will be most likely to give other users what they actually want.

Unbiased

Bias occurs when the experience influences the types of interactions users have, or influences the types of feedback users give.

One common source of bias is that different types of outcomes get reported at different rates.

Consider a spam filtering program. If a spam email gets to the user's inbox, it is right in their face. They are likely to notice it and may press a "this is junk" button and generate useful data on a bad outcome. On the other hand, if the filter deletes a personal email, the user may never notice.

In this case, choices made in designing the experience have introduced bias and have made the resulting data much less useful for building intelligence.

Another potential source of bias is that users with strong sentiment toward the interaction (either good or bad) are more likely to give feedback than users with neutral opinions.

Another source of bias is when the experience encourages users to take certain choices over others. For example, when the experience presents choices in a list, the user is more likely to choose the first item on the list (and is very unlikely to choose an item that isn't on the list at all).

Does Not Contain Feedback Loops

Experience and intelligence will affect one another, sometimes in negative ways.

For example, if some part of the user experience becomes more prominent, users will interact with it more. If the intelligence learns from these interactions, it might think users like the action more (when they don't like it more, they simply notice it more because of the change to the user experience).

Conversely, if the intelligence makes a mistake and starts suppressing an action that users like, users will stop seeing the option. They will stop selecting the action (because they can't). The action will disappear from the data. The intelligence will think users don't like the option any more. But it will be wrong…

Here are some ways to deal with feedback:

- Include what the user saw in the context of the recorded data. If one option is presented in a larger font than another, record it in context. If some options were suppressed, record it in context.
- Put a bit of randomization in what users see, for example switching the order of certain options. This helps gather data in broader situations, and it helps identify that feedback may be occurring.
- Record the effort the user took to select the option in the context. For example, if the user had to issue a command manually (because it was suppressed in the intelligent experience) the intelligence should know it made a costly mistake. If the user had to browse through many pages of content to find the option they want, the intelligence should know.

Scale

In order to improve, an Intelligent System must be used. This means that the intelligent experiences must be prominent, they must be interesting to use, they must be easy to find, and they must be the things that users do often.

Consider two options: a new Intelligent System and an established Intelligent System.

The new Intelligent System will probably not have many users. This means the intelligent experience must be central to what the users will do, so they will interact with it regularly. In these cases, the intelligent experience will need to encourage interaction, make it fun, and maybe even change the whole product around to put the intelligent interaction front and center.

The established Intelligent System will have much more usage. This means that the intelligent experiences can be more subtle. They can be put in places where fewer users will see them. This doesn't mean they are less valuable—they may be solving very important problems, but problems that users encounter on a weekly or monthly basis instead of on a daily basis.

For some context, a system that:

- Generates tens of interaction per day is basically useless for verifying or improving intelligence.
- Generates hundreds of interactions per day can probably validate intelligence and produce some simple intelligence.
- Generates thousands or tens of thousands of interactions per day can certainly validate the Intelligent System and produce intelligence for many hard, open-ended problems.
- Generates hundreds of thousands or millions of interactions per day will probably have all the data it needs for most tasks.

An effective intelligent experience will attract users and get them engaging, building a base for gathering data and producing interesting intelligence.

Ways to Understand Outcomes

When an interaction between a user and intelligence occurs, it isn't always easy to understand whether the outcome is positive or negative. For example, in a music recommender system—did the user push the "next" button because they hate the song (the intelligence got it wrong) or because they see the next song coming up and they really love that song (the intelligence got it right)?

The best intelligent experiences are set up to make the outcome clear implicitly—that is, without relying on the user to do anything other than use the product. But this isn't always possible. Even when it is possible, it can be useful to have a backstop to make sure the implicit data has all the right properties to create intelligence. Methods for understanding outcomes include:

- Implicit observations
- User ratings
- Problem reports
- Escalations
- User classifications

Many large Intelligent Systems use all of these approaches to understand as much as possible about the outcomes their users are having.

Implicit Outcomes

An ideal experience will produce good, useful data without requiring any special user action. Users will produce data simply by using the system, and that data will have all the properties required to grow intelligence.

For example, when a user sets their thermostat to 72 degrees it's pretty clear what they would have wanted the intelligence to do. Or when the user buys a product or some content, it's clear they were interested in it.

It's often hard to know exactly how to interpret a user's action. Users may not spend the time to make perfect decisions, so the way data is presented to them could bias their results. For example, they might have selected one of the five recommended movies because it was the absolute best movie for them at that moment—or because they didn't feel like looking at more options.

Because of this, achieving a fully implicit data collection system is quite difficult, requiring careful coordination and curtailing of the experience so the usage can be effectively interpreted.

Ratings

User ratings and reviews can be a very good source of data. For example, designing experiences that allow users to:

- Rate the content they consume with 1-5 stars.
- Give a thumbs-up or thumbs-down on a particular interaction.
- Leave some short text description of their experience.

Users are used to leaving ratings and many users enjoy doing it, feeling like they are helping others or personalizing their experience.

But there are some challenges with ratings:

1. Users don't always rate everything. They may not feel like taking the effort.
2. There can be some time between the interaction and the rating. The user might not remember exactly what happened, or they might not attribute their outcome to the interaction.
3. The rating might not capture what the intelligence is optimizing. For example, consider these two questions: How good is this book? Was that the right book to recommend to you yesterday?
4. Ratings vary across different user populations—five stars in one country is not the same as five stars in others.
5. Infrequent interactions will not get many ratings, and be hard to learn about.

Reports

An experience can allow users to report that something went wrong. For example, many email systems have a "report as spam" button for spam messages that get to the inbox.

Data collected through reports is explicitly biased, in that the only outcomes it captures are bad ones (when the intelligence made a mistake), and users won't report every problem. For example, users might click "report as spam" on 10% of the spam they see, and just ignore or delete the rest of the spam.

Because of this, data from reports is difficult to use as an exclusive source for creating intelligence. However, reports can be very effective in verifying intelligence and in tuning the experience to control the negative effects of poor intelligence.

Escalations

Escalations are an expensive form of report that occur when users get upset enough to contact customer support (or visit a support forum, and so on). Escalations are similar to reports but are out-of-band of the main product experience. And they usually represent very unhappy users and important mistakes.

Escalations tend to be messy and unfocused. The user who is having trouble won't always know what part of the system is causing trouble. They won't express the problem in the same terms as other users who reported it.

Because of this, it can be difficult and expensive to sort through escalations and use them to improve intelligence—they are better for identifying big problems than for refining the system.

In general, it's better to allow users to report problems in context (with an in-line reporting experience, at the time of the problem) so the system can capture the relevant context and actions that led to the bad outcome.

User Classifications

You can simply ask users about their outcomes using specific (carefully controlled) questions, in context as users interact with your system. Sort of like a survey. Imagine:

- Using a light automation system and once every three months it says: In order to improve our product we'd like to know—right now would you prefer the lights to be on or off?
- Using a photo editing program and after touching up a hundred photos it says: Please help improve this program. Is there a face in the image you are working on?

Classification can produce very focused data for improving intelligence. It can also be made unobtrusive, for example by limiting questions:

- To one question per thousand interactions.
- To users who have volunteered to help.
- To users who haven't signed up for premium service.
- And so on.

Summary

The intelligent experience plays a critical role in getting data for growing intelligence. If an intelligent experience isn't explicitly designed to produce good data for creating intelligence, it is almost certainly going to produce data that is poor (or useless) for creating intelligence.

Experiences produce better data when users interact with intelligence often, when they perceive value in the interactions, and when they have the right information to make good decisions.

Data is most useful when it:

- Contains the context of the interaction, the action taken, and the outcomes.
- Contains good coverage of all the situations where the Intelligent System will be used.
- Represents real usage, that is, interactions that users care about.
- Contains unbiased data.
- Contains enough information to identify and break feedback loops.
- Produces a meaningful amount of data with respect to the complexity of the problem.

The most effective data comes from implicit interactions, where the user naturally expresses their intent and their opinion of the outcome by using the product.

When implicit interactions aren't sufficient other methods of understanding outcomes include:

- Allowing users to explicitly rate the content or interactions they have.
- Allowing users to report problems directly from the user experience.
- Giving users access to support paths to report large, costly problems.
- Asking users to answer specifically curtailed questions (infrequently).

Many Intelligent Systems use all of these approaches. The majority of their data might come from implicit interactions, but the other techniques will be available to monitor and make sure the implicit data is high quality (has not started to develop bias or feedback).

Producing an experience that produces excellent data without burdening users is hard—and it is very important to building great Intelligent Systems.

For Thought…

After reading this chapter you should:

- Know how to craft experiences that collect the data needed to evaluate and grow intelligence.
- Understand the options for collecting this data from users, which range from ones requiring no explicit user action to ones requiring extensive user involvement.

You should be able to answer questions like these:

- Think of an intelligent service you use regularly that seems to collect all its data implicitly. How could it be changed to leverage user classifications?
- Imagine an Intelligent System to assist your favorite hobby. Think of an implicit way to collect "good data" when it does something wrong. Now think of another way to collect data in the same situation that expresses a different user interpretation of the mistake (for example maybe that the intelligence was correct, but the user didn't want to do the suggested action for some other reason).

CHAPTER 10

Verifying Intelligent Experiences

Intelligent experiences fail for two main reasons:

1. Because they are implemented incorrectly: the experiences have a bug, are confusing, or don't help users achieve their goals.
2. Because the intelligence is incorrect: it suggests the wrong automation, is certain of something that isn't true, or proposes the wrong ordering of information.

For example, an intelligent experience might:

- Present a list of images with faces, but fail to include the user's favorite photo from their wedding. Is the image excluded because the experience has a bug? Or is it excluded because the intelligence didn't recognize that the image had a face in it?
- Turn the lights off when the room is already sort of dark. Did the experience turn the light off because it knew the sun was coming up soon (even though it knew the room was too dark)? Or did the light turn off because the intelligence thought the room was empty?
- The user hates horror movies, but the system is recommending lots of horror movies. Is it doing this because the user's husband set a preference in some hidden menu to hide all the romantic comedy movies forever and ever no matter what? Or is it doing this because the intelligence examined the movies and thought they were adventure, not horror?

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_10

Verifying that an intelligent experience is functioning properly, separate from the quality of intelligence, is critical to producing effective intelligent experiences.

Getting Intended Experiences

So the intelligence might be right, the intelligence might be wrong; fine. But how did the rest of the system perform?

- Did the user get the intended experience when the intelligence was correct?
- Did the experience actually help to achieve the user's (and the system's) goals?
- Does the experience remain effective as the system and the intelligence evolves?

This section talks about ways of isolating the experience from the rest of the changes that might occur in an Intelligent System so you can have some basis for verification.

Working with Context

The context represents the state of the Intelligent System when an intelligent experience is triggered. This can include all the variables the intelligence might take into account in producing its decisions. It should also include the variables that impact how users will interpret the system's outputs and how they will react. For example:

- The context in a computer vision system might include the lighting in the environment, the camera properties, the users and their features (ethnicity, gender, clothing), and so on.
- The context in a music playing system might include the user's ratings of songs, the list of songs the user owns, the recent listening history, and so on.
- The context in a home automation system might include the number and types of sensors, the rooms they are placed in, the distance between them, and so on.

When you want to verify an intelligent experience works in a particular context, you need to create the context and try it. For example, you need to set up the computer vision system in exactly the right way to recreate the context, get the lighting just right, bring in the same users, put them in front of the camera, and have them smile exactly right, and so on.

This can be problematic. Particularly for large, open-ended problems that have many, many possible contexts—an infinite number of them.

When verifying intelligent experiences, it can help to have some tools to help produce (or capture) important contexts, explore them, and replay them in the intelligent experience so you can test them again and again as the Intelligent System evolves. Some options for working with contexts include:

- **Manually producing contexts** the same way users would, by using the system, creating an account, building up the history of interactions, browsing to the right place, and so on. This can be extremely challenging and error prone, as the contexts can be subtle; for example, the change in inflection when speaking to a computer can affect how well a speech recognition system will perform. They can be quite extensive—for example, usage history over a long period of time.
- **Crafting via an editor** or some other tool that lets contexts be created, edited, captured, and played back to the system. For example, a tool that specifies a house layout, the location of sensors, and the various readings that are on the sensors at a particular time. The level of sophistication and automation can vary, but these types of tools can be very helpful in developing baseline verifications.
- **Recorded via usage** either in a lab setting or from actual users. Examples include allowing the system to dump the context at any time, and creating tools to load the dumps and evaluate them against updated versions of the intelligent experience and intelligence. These types of approaches can help get actual usage and a great deal of variety into the verification process.

When verifying intelligence it is good to have all of these options. A typical work-flow might look like this:

- Reproducing a problem that a user reports.
- Capturing the exact context that caused the report.
- Viewing the context in some kind of tool to see what was really going on (not just the part the user was complaining about).
- Making a change to the experience.
- Executing the context again to see if the change would fix the problem.
- Communicating the problem to intelligence creators (but they may or may not be able to help).

Note that a context for verifying experiences is similar to, but different from the data needed for creating intelligence. To create intelligence you also need to capture information about the outcome. But the context alone can help verify (and improve) the experience.

Working with Intelligence

The workflow of tracking and replaying contexts is similar to tracking and fixing bugs in other complex software products. But changing intelligence can make it difficult. This is because the experience a user sees results from combining a particular context with a particular intelligence—and intelligence changes.

Because of this, it also helps to have a collection of known intelligences that can be combined with known contexts to see the results. Some options for getting intelligences to help with verification include:

- **Live intelligence** that is just what users will see. If things are going well, this will be the most accurate intelligence available, and it will be the best way to verify what users are actually seeing. But it may also change very rapidly (as intelligence evolves), requiring some work to distinguish intelligence changes from experience changes. One example of a problem this might cause is that you have an experience problem you are working to fix and then… it disappears.

You haven't changed anything yet, but maybe the intelligence changed out from under you and you can no longer reproduce the problem.

- **Fixed intelligence** that is the same every time it is executed. For example, some sort of checkpoint on the live system's intelligence (maybe the first day of alpha testing or the first day of every month). This intelligence is similar to the live intelligence, but provides a stable baseline to use for working with experience problems over an extended period of time.
- **Simple intelligence** that is somewhat interpretable. For example, instead of using a model produced by machine learning, use a small set of heuristic rules. This can be useful when debugging problems—by having just enough intelligence to cause interesting problems, but not so much that the intelligence errors overwhelm the experience problems.
- **Dumb intelligence** that always gives the same answer no matter what context is provided. For example, no matter what bands the user likes, the intelligence recommends Dire Straits. A dumb intelligence can help rule out obvious experience problems.

When verifying intelligence it helps to have all of these options available.

Bringing it Together

Common types of errors to look for include:

- The context being incorrectly interpreted by the intelligence, resulting in unexpected output.
- The intelligence being misinterpreted by the experience, resulting in systematically incorrect experiences.
- The rendering of the user experience being incorrect for some combinations of context and intelligence, resulting in hard-to-interpret, unappealing, or illegible results.

Achieving Goals

The experience plays a very important role in achieving a system's objectives.

By presenting intelligence, deciding how and how often users see it, deciding how it is colored, where it is displayed, how often it happens, and what words or icons are used to describe it, an experience can make a poor-quality intelligence into a useful tool; an experience can also render a high-quality intelligence useless.

Consider a system designed to protect users from scams on the Internet. If the user visits a page that the intelligence thinks is a scam web page, the experience gives them a warning of some sort.

If the intelligence were perfect, the warning could be very forceful (perhaps blocking access to the page and giving the user no recourse) but intelligence is never perfect, so the warning needs to balance the risk of the user getting scammed with the risk of the user being scared away from a site they actually wanted to go to.

So how do we verify the effectiveness of any particular experience?

This can be accomplished by isolating the job of the intelligence from the job of the experience.

The job of the intelligence is to flag scam sites. It might be right; it might be wrong.

The job of the experience is to:

1. Successfully keep users from being scammed when intelligence correctly flags a scam.
2. Minimize the cost to the user when the intelligence incorrectly flags a non-scam as a scam.

And so these are the properties that need to be measured to verify that the experience is doing its job:

- What percent of users who correctly received a warning got scammed anyway?
- What percent of users who incorrectly received a warning got turned away from a legitimate site?

More generally, for every possible outcome of the intelligence—all the ways it could have been right and all the ways it could have been wrong—consider what the role of the experience is in that setting—and measure it.

And note: these types of things are very difficult to measure in a development setting—accurately measuring them often requires real users making real decisions.

Continual Verification

Intelligence changes over time. User behavior changes over time, too. These changes can cause experiences to become less effective over the life of a system. Or the changes could result in opportunities to make the experience more forceful and get more benefit.

Some changes that can affect the experience are:

1. The quality of the intelligence could become better or worse.
2. The quality of the intelligence might remain the same, but the types of mistakes it makes might change.
3. Users might become accustomed to (or fatigued by) the experience and start to ignore it.
4. New users might adopt the system and have different behaviors than the original users.

In an Intelligent System there are many sources of change. And most changes provide opportunity (at least if you're an optimist, you'll consider them opportunities) to rebalance the parts of the system to achieve better results—as long as you know the change happened and how the change affected the system.

Adapting the experience isn't always the right answer—but it might be.

Summary

Verifying that an intelligent experience is working can be complex, because mistakes can be caused by the experience, by the intelligence, by having the user in some context you didn't anticipate, or because something has changed.

When there are a lot of mistakes they can become overwhelming, and it is useful to have some ways to simplify the problem and isolate what is going on. This can also help prevent lots of finger-pointing: "This is an intelligence mistake!" "No, it is an experience problem." "No, it is an intelligence problem." … And so on.

It is good to have a way to capture, inspect, and manipulate contexts. This can be done through:

- Manually producing them.
- Having tools to inspect and modify them.
- Having ways to record them from real users and situations.

It is also good to have ways to play contexts back against various versions of the intelligence and see what the experience would produce. Common intelligences used in verification include:

- The live intelligence that users are getting.
- A checkpoint of the live intelligence that can be used over an extended period without changing.
- A simple intelligence that is easy to interpret (maybe some hand-crafted rules).
- A dumb intelligence that always says the same thing.

It is also important to verify that the intelligent experience is helping users achieve their objectives. This is most commonly done by looking at live interactions when the intelligence was right and ensuring that users are ending up with good outcomes (and not being confused or misled by the experience).

And Intelligent Systems are always changing. The effectiveness of the intelligent experience needs to be constantly reverified. Change also creates opportunities to rebalance and provide more user value—if you are tracking it and have the tools to identify the effects of the change.

For Thought…

After reading this chapter, you should be able to:

- Isolate experience issues (from intelligence ones) and verify that the experience is working as intended on its own.
- Build an effective test plan for an intelligent experience.
- Understanding the importance of reverifying as the intelligence and user-base change.

You should be able to answer questions like these:

Consider an Intelligent System you use regularly.

- Sketch a plan to verify that its experience is functioning correctly.
- Design a simple intelligence to help verify this. Describe three contexts you would manually create to execute this simple intelligence as part of verification. Why did you pick those three?
- How could you monitor the system to detect that the experience is losing effectiveness?

PART III

Implementing Intelligence

Chapters 11-15 discuss the implementation of Intelligent Systems. This part covers all the important components that need to be built, along with their responsibilities, the options, and trade-offs. These are the systems that take intelligence (the models produced by machine learning) and turn it into a fully functional system—executing the intelligence, moving it where it needs to be, combining it, monitoring it, and supporting its creation. These systems enable the operation of Intelligent Systems throughout their life cycle, with confidence.

CHAPTER 11

The Components of an Intelligence Implementation

In order to have impact, intelligence must be connected to the user.

That is, when the user interacts with the system, the intelligence must be executed with the proper inputs and the user experience must respond. When new intelligence is created, it must be verified and deployed to where it is needed. And there must be ways to determine if everything is working, and mechanisms to gather what is needed to improve the intelligence over time.

The intelligence implementation takes care of all of this. It is the foundation upon which an intelligent service is created.

As an analogy, consider a web server.

A web server implementation needs to accept network connections, do authentication, interpret requests and find the right content, process it (maybe running scripts), serve it, create logs—and a lot more.

Once a web server implementation is available, content creators can use it to produce all sorts of useful web sites.

This is similar to an implementation of an Intelligent System—it allows intelligence orchestrators to produce useful user interactions over time. Intelligence, artificial intelligence, and machine learning are changing the world, but proper implementations of Intelligent Systems put these into position to fulfill their promise.

This chapter introduces the components that make up an implementation of an Intelligent System.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_11

An Example of Intelligence Implementation

Imagine an Intelligent System designed to help users know if a web page is funny or not.

The user browses to a page and a little smiley face pops up if the system thinks the user will laugh if they read the page. You know, so no one ever has to miss something funny ever again…

Simple enough. Let's discuss how this Intelligent System might be implemented.

It starts with a program that can examine the contents of a web page and estimate if it is funny or not. This is probably a model produced by machine learning, but it could be some simple heuristics. For example, a very simple heuristic might say that any page containing the phrases "walked into a bar" or "who's there" is funny, and every page that doesn't contain those phrases is not funny.

And for some it might actually be true—maybe you have that uncle, too…

Someone could write a plug-in for web browsers that includes this model. When a new web page is loaded, the plug-in takes the contents of the page, checks it against the model, and displays the appropriate user experience depending on what the model says (the smiley face if the model thinks the page is funny, nothing if the model thinks the page is not funny).

This plug-in is a very simple intelligence implementation.

But ship the plug-in, see users download it, and pretty soon someone is going to want to know if it's working.

So maybe the plug-in includes a way for users to provide feedback. If they get to the bottom of a page the system said would be funny, but they didn't laugh, there is a frowny face they can click to let the system know it made a mistake. If they read a page the system didn't flag as funny, but find themselves laughing anyway, there is a laugh button they can click to let the system know. We might even want to know if users are more likely to read pages that are marked as funny compared to pages that aren't, so maybe we measure how long users spend on pages we've flagged as funny, and how long they spend on ones we haven't.

The plug-in gathers all of this user feedback and behavior and sends it back to the service as telemetry.

Adding telemetry improves the overall intelligent implementation because it lets us answer important questions about how good the intelligence is and how users are perceiving the overall system—if we are achieving our overall objectives. It also sets us up to improve.

Because when the intelligence creators look at the telemetry data, they are going to find all sorts of places where their initial model didn't work well. Maybe most funny things on the Internet are in images, not text. Maybe people in one country aren't amused by the same things that people in another country find hilarious. That kind of thing.

The intelligence creators will spend some time looking at the telemetry; maybe they'll crawl some of the pages where the initial system made mistakes, and build some new models. Then they are going to have a model they like better than the original one and they are going to want to ship it to users.

The intelligence implementation is going to have to take care of this. One option would be to ship a whole new version of the plug-in—the Funny Finder v2.0—which contains the new model. But users of the first version would need to find this new plug-in and choose to install it. Most of them won't. And even the ones who do might take a long time to do it. This causes the intelligence to update slowly (if at all) and reduces the potential of the intelligence creators' work. Further, the intelligence might change fast: maybe every week, maybe every day, maybe multiple times per hour. Unless the implementation can get new intelligence to users regularly and reliably, the Intelligent System won't be very intelligent.

So the implementation might upgrade the plug-in to do automatic updates. Or better, the plug-in might be changed to update just the models (which are essentially data) while leaving all the rest of the code the same. Periodically, the plug-in checks a server, determines if there is some new intelligence, and downloads it.

Great. Now the implementation runs the intelligence, measures it, and updates it. The amount of code to service the intelligence at this point is probably more than the intelligence itself: a good portion of any Intelligent System is implementation. And this version of the system is complete, and closes the loop between the user and the intelligence, allowing the user to benefit from and improve the intelligence simply by using the system and providing feedback.

But some things are really not funny. Some things are offensive, horrible. We really, really don't want our system to make the type of mistake that flags highly offensive content as funny.

So maybe there is a way for users to report when our system is making really terrible mistakes. A new button in the user experience that sends telemetry about offensive pages back to our service.

Maybe we build a little workflow and employ a few humans to verify that user-reported-offensive sites actually are terrible (and not some sort of comedian-war where they report each other's content). This results in a list of "really, truly not funny" sites. The implementation needs to make sure clients get updated with changes to this list as soon as possible. This list could be updated when the model is updated. Or maybe that isn't fast enough, and the plug-in needs to be more active about checking this list and combining the intelligence it contains with the intelligence the model outputs.

So now the plug-in is updated so that every time it visits a new page it makes a service call while it runs its local model. Then it combines the results of these two forms of intelligence (the server-based intelligence and the client-based intelligence). If the server says a site is "really, truly not funny," it doesn't matter what the client-side model says—that site is not funny.

By this point the intelligence creators are going to have all sorts of ideas for how to build better models that can't run in the plug-in. Maybe the new ideas can't be in the client because they take too much RAM. Maybe they can't be on the client because they required external lookups (for example, to language translation services) that introduce too much latency in the plug-in. Maybe the plug-in needs to run in seriously CPU-restrained environments, like in a phone, and the intelligence creators just want a bit more headroom.

These types of intelligences may not be runnable on the client, but they may be perfectly usable in the service's back end.

For example, when a lot of users start visiting a new site, the back end could notice. It could crawl the site. It could run dozens of different algorithms—ones specialized for images, ones that look for particular types of humor, ones tuned to different languages or cultures—and ship the outputs to the client somehow.

So now the plug-in is combining intelligence from multiple sources—some in the cloud, and some that it executes itself. It is managing experience. It is measuring the results of users interacting with the intelligence and collecting data for improvement. And more.

Not every Intelligent System implementation needs all of these components. And not every Intelligent System needs them implemented to the same degree. This part of the book will provide a foundation to help you know when and how to invest in various components so your Intelligent System has the greatest chance of achieving its goals—and doing it efficiently.

Components of an Intelligence Implementation

An intelligence implementation can be very simple, or it can be very complex. But there are some key functions that each Intelligent System implementation must address, these are:

- Execute the intelligence at runtime and light up the experience.
- Ingest new intelligence, verify it, and get it where it needs to be.
- Monitor the intelligence (and user outcomes) and get any telemetry needed to improve the system.
- Provide support to the intelligence creators.
- Provide controls to orchestrate the system, to evolve it in a controlled fashion, and to deal with mistakes.

The Intelligence Runtime

Intelligence must be executed and connected to the user experience.

In order to execute intelligence, the system must gather up the context of the interaction, all the things the intelligence needs to consider to make a good decisions. This might include: what the user is doing; what the user has done recently; what the relevant sensors say; what the user is looking at on the screen; anything that might be relevant to the intelligence making its decision.

The context must be bundled up, converted, and presented to the model (or models) that represent the intelligence. The combination of context and model results in a prediction.

The runtime might be entirely in the client, or it might coordinate between a client and a service.

Then the prediction must be used to affect the system, light up the experience—create impact.

Intelligence Management

As new intelligence becomes available it must be ingested and delivered to where it is needed.

For example, if the intelligence is created in a lab at corporate headquarters, and the runtime needs to execute the intelligence on toasters all across America, the intelligence management system must ship the intelligence to all the toasters.

Or maybe the intelligence runs in a service.

Or maybe it runs partially in a back-end, and partially in the client.

There are many options for where intelligence can live, with pros and cons.

Along the way, the intelligence needs to be verified to make sure it isn't going to do anything (too) crazy.

And Intelligent Systems usually rely on more than one source of intelligence. These might include multiple models, heuristics, and error-overrides. These must be combined, and their combination needs to be verified and delivered.

Intelligence Telemetry Pipeline

Getting the right monitoring and telemetry is foundational to producing an intelligence that functions correctly and that can be improved over time.

Effective monitoring and telemetry includes knowing what contexts the intelligence is running in and what types of answers it is giving. It includes knowing what experiences the intelligence is producing and how users are responding. And an effective telemetry system will get appropriate data to grow new and better intelligence.

In large Intelligent Systems the telemetry and monitoring systems can produce a lot of data—a LOT.

And so the intelligence implementation must decide what to observe, what to sample, and how to digest and summarize the information to enable intelligence creation and orchestration among the various parts of the Intelligent System.

The Intelligence Creation Environment

In order for an Intelligent System to succeed, there needs to be a great deal of coordination between the runtime, the delivery, the monitoring, and the intelligence creation.

For example, in order to produce accurate intelligence, the intelligence creator must be able to recreate exactly what happens at runtime. This means that:

- The telemetry must capture the same context that the runtime uses (the information about what the user was doing, the content, the sensor output, and so on).
- The intelligence creator must be able to process this context exactly the same way it is processed at runtime.

- The intelligence creator must be able to connect contexts that show up in telemetry to the eventual outcome the user got from the interaction—good or bad.

This can be hard when intelligence creators want to innovate on what types of information/context their models examine (and they will; it's called feature engineering and is a key part of the intelligence creation process). It can be hard when the monitoring system doesn't collect exactly the right information, leading to mismatches between runtime and what the intelligence creator can see. It can be hard when the intelligence is executed on a type of device different from the intelligence creation device. Maybe the runtime has a different coprocessor, a different graphics card, or a different version of a math library from the ones in the intelligence creation environment.

Any one of these can lead to problems.

An intelligence implementation will create an environment that mitigates these problems, by providing as much consistency for intelligence creators as possible.

Intelligence Orchestration

The Intelligent System needs to be orchestrated.

- If it gets into a bad state, someone needs to get it out.
- If it starts making bad mistakes, someone needs to mitigate them while the root cause is investigated.
- As the problem, the intelligence, and the user base evolve, someone needs to be able to take control and tune the intelligence and experience to achieve the desired results.

For example, if the intelligence creation produces a bad model (and none of the checks catch it) the model may start giving bad experiences to customers. The intelligence orchestration team should be able to identify the regression quickly in telemetry, track the problems down to the specific model, and disable or revert the model to a previous version.

If things go badly enough, the intelligence orchestrator might need to shut the whole thing down, and the implementation and experience should respond gracefully.

Providing good visibility and tools for orchestrating an Intelligent System allows intelligence creation to act with more confidence, take bigger risks, and improve more rapidly.

Summary

An Intelligent System needs to be implemented so that intelligence is connected with customers. This includes:

- Executing the intelligence and lighting up the intelligent experience.
- Managing the intelligence, shipping it where it needs to be.
- Collecting telemetry on how the intelligence is operating and how users are responding (and to improve the intelligence).
- Supporting the people who are going to have to create intelligence by allowing them to interact with contexts exactly the same way users will.
- Helping orchestrate the Intelligent System through its life cycle, controlling its components, dealing with mistakes, and so on.

An Intelligent System is not like a traditional program, in that it is completed, implemented, shipped and walked away from. It is more like a service that must be run and improved over time. The implementation of the Intelligent System is the platform that allows this to happen.

For Thought…

After reading this chapter, you should:

- Understand the properties of an effective intelligent implementation—what it takes to go from a piece of intelligence (for example, a machine learned model) to a fully functional Intelligent System.
- Be able to name and describe the key components that make up an implementation of an Intelligent System.

You should be able to answer questions like these:

Consider an activity you do daily:

- What would a minimalist implementation of an Intelligent System to support the activity look like?
- Which component—the runtime, the intelligence management, the telemetry, the intelligence creation environment, the orchestration—do you think would require the most investment? Why?

CHAPTER 12

The Intelligence Runtime

The intelligence runtime puts intelligence into action. It is responsible for interfacing with the rest of the Intelligent System, gathering the information needed to execute the system's intelligence, loading and interpreting the intelligence, and connecting the intelligence's predictions back to the rest of the system.

Conceptually, an intelligence runtime might look something like this:

```
Intelligence theIntelligence =
               InitializeIntelligence(<intelligence data file>,
                              <server address>,
                              <etc...>);

Context theContext =
               GatherContext(<sensors>,
                              <content>,
                              <user history>,
                              <other system properties>,
                              <etc...>);

Prediction thePrediction =
       ExecuteIntelligence(theIntelligence, theContext);

UpdateTheExperience(thePrediction);
```

It's simple in principal, but, like most good things, it can require a great deal of complexity in practice.

This section covers the principal elements that an intelligence runtime will handle:

- Gathering context.
- Constructing features.
- Loading models.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_12

- Executing models on the context/features.
- Connecting the resulting predictions with the experience.

Context

The context in an intelligence runtime consists of all the data the intelligence might use to make its decision. For example:

- If the Intelligent System is trying to determine if a sprinkler system should run, the context might include.
    - The amount of time since the sprinkler last ran.
    - The forecast for the next several days (will it rain or not).
    - The amount of rain that has fallen in the past several days.
    - The type of landscaping the user has.
    - The time of day.
    - The temperature.
    - The humidity.
    - And so on.
- If the Intelligent System is looking for cats in an image, the context might include.
    - The raw image data (RGB intensity for the pixels).
    - Metadata about how the image was captured (the exposure and frame rate, the time, the zoom).
    - The location the image came from.
    - And so on.
- If the Intelligent System is trying to tell if a user should take a break from their computer, the context might include.
    - The time since the last break.
    - The activities the user has done since the break (number of keystrokes and mouse movements).

- An indication of how active the user has been.
- The amount of time until the user's next meeting (according to their calendar).
- The name of the application that the user is currently interacting with.
- And so on.

The context can be just about any computer-processable information, like a URL, a sensor reading from some embedded device, a digital image, a catalog of movies, a log of a user's past behavior, a paragraph of text, the output of a medical device, sales receipts from a store, the behavior of a program, and so on.

These are the things that might be useful in making the decision about what the Intelligent System should do. In fact, the context is usually a superset of the things the intelligence will actually use to make the decision.

A good context will:

- Contain enough information to allow the intelligence be effective.
- Contain enough information to allow the intelligence to grow and adapt.
- Contain enough information to allow the Intelligent System's orchestrators to track down mistakes.
- Be efficient enough to gather at runtime for the intelligence call.
- Be compact enough to include in telemetry.

Clearly there are trade-offs. More complete context gives the intelligence more potential, but it may also encounter engineering and operational constraints, requiring additional CPU and memory usage and network bandwidth, and introducing latency and increasing data size.

Feature Extraction

Gathering these context variables can require significant effort. But it isn't the end of the process. There is also another layer of code that converts these variables into the representation the intelligence needs. This extra layer is often called "feature extraction" or "featurization" and is a common part of machine-learning systems.

There are a number of reasons this matters for an intelligence implementation:

- The code that produces features from context needs to run in the runtime. That is, it needs to be efficient and reliable.
- This code is often technical, computationally expensive, mathy, and not intuitive to people who don't have extensive experience in working with data.
- The intelligence creators are going to want to constantly change what this code does; in fact, they may need to change it periodically to continue to grow the intelligence.
- This code is tightly coupled to the intelligence, and the feature extraction code and intelligence (models) absolutely must be kept in sync—the exact version of the feature extraction code used to build the intelligence must also be used in the runtime when the intelligence is executed.

Because of these factors, feature extraction must be carefully planned. Some best practices include:

- Have extensive test sets for feature extraction code. This should include:
  - Unit tests on the extractors.
  - Performance tests to make sure the code meets CPU and RAM requirements.
  - A test that periodically executes feature extraction code against a set of known contexts (maybe daily, maybe at each check-in) and compares the output to a known benchmark. Any change in output should be examined by engineers and intelligence creators to make sure it is what was intended.
  - A test that ensures the feature extraction code at runtime (when users are interacting with it) is doing exactly the same thing it is doing in the intelligence creation environment (when it is being used to create models).
- Have version information encoded in intelligence and in the feature extraction code and have the runtime verify that they are in sync.

- Have the ability to change feature-extraction code out-of-band of the full application; for example, by sending a new version of the feature extraction code with each model update.
- Develop a life-cycle for moving feature extraction changes from intelligence creators, to engineers, to deployment. The life-cycle should allow intelligence creators to move quickly in evaluating changes, and the quality of the code deployed to users to be high.

Models

The model is the representation of the intelligence (and we will discuss representation options in coming chapters). For now, imagine the model as a data file. In some Intelligent Systems, the models are changed every few days. In some systems they are changed every few minutes.

The rate of change will depend on these factors:

1. How quickly the system gets enough data to improve the models.
2. How long it takes to construct new models.
3. How much it costs to distribute the new models.
4. How necessary change is relative to the success criteria.

In general, models will change (much) more quickly than the larger program. An effective runtime will:

- Allow the model to be updated easily (that is, without having to restart the whole system).
- Make sure the models it uses are valid, and all the components (the feature extractors, the models, the context) are in sync.
- Make it easy to recover from mistakes in models (by rolling back to previous versions).

One key consideration with models is the amount of space they take up on disk and in RAM. Left unchecked, machine learning can produce models that are quite large. But it is always possible to make tradeoffs, for example between model size and accuracy, if needed.

Execution

Executing a model is the process of asking the model to make its prediction based on the context (and associated features).

In the simplest case, when there is a single model and it lives on the client, executing a model can be as easy as loading the features into an array and making a function call into an off-the-shelf model execution engine. There are libraries to execute pretty much every type of model that machine learning can build, and in most cases these libraries are perfectly acceptable—so most of the time you won't even need to write much code to load and execute a model.

Of course, in some cases these libraries aren't acceptable. Examples include when RAM or CPU is a constraint; or when the execution is happening on a device with special hardware (like a GPU or FPGA) that the libraries don't take advantage of.

Another form of execution involves a model running on a server. In these cases, the client needs to bundle some (compact) representation of the context (or the features), ship them to the server, and wait for an answer, dealing with all the coordination required to properly account for the latency of the call and keep it from resulting in bad user experiences.

When the system has more than one model (which is often the case), the execution needs to execute each of the models, gather their results, and combine them into a final answer. There are many ways to do this, including:

- Averaging their results.
- Taking the highest (most certain) result.
- Having some rules about the context that select which one to trust (for example, one model nails house prices in 90210 but is terrible everywhere else, another model works with condos but not single-family homes).
- Having a model that combines the outputs of the other models.

The chapters in Part IV, "Creating Intelligence," will discuss model combination in more detail along with the pros and cons of these (and other) approaches.

Results

Executing the intelligence results in an answer. Maybe the intelligence outputs a 0.91, which represents a 91% probability in whatever the intelligence is talking about. The implementation must use this to affect what the intelligent experience will do. For example, it might compare the value to a threshold and if the value is high (or low) enough, initiate a prompt, or automate something, or annotate some information—whatever the intelligent experience wants to do.

Instability in Intelligence

Using the raw output of intelligence can be risky, because intelligence is intrinsically unstable. For example:

Take a digital image of a cow. Use a cow-detector model to predict the probability the image contains a cow, and the output might be high, maybe 97.2%.

Now wait a day. The back-end system has new training data, builds a new cow-detector model, distributes it to the runtime. Apply the updated model to the same cow picture and the probability estimate will almost certainly be different. The difference might be small, like 97.21% (compared to 97.2% from the original) or it might be relatively large, like 99.2%. The new estimate might be more accurate, or it might be less accurate. But it is very, very unlikely the estimate will be the same between two different versions of intelligence.

Subtle changes in context can have similar effects. For example, point your camera at a cow and snap two digital images right in a row, one after the other, as fast as you can. To a human, the cow looks the same in both images, the background looks the same, and the lighting looks the same—the "cow detector" model should output the same probability estimation on the first image as on the second. According to a human.

But to a model, the images might look very different: digital sensors introduce noise, and the noise profile will be different between the images (resulting in little speckles that human eyes have a hard time seeing); the cow might have blinked in one and not the other; the wind might have blown, changing the location of shadows from leaves; the position of the camera may have changed ever-so-slightly; the camera's auto-focus or exposure correction might have done slightly different things to the images.

Because of all of this, the intelligence might make different estimations on the two images. Maybe 94.1% in one and 92.7% in the other.

Maybe that change isn't a big deal to the intelligent experience that results, but maybe it is.

Trying to pretend that there isn't instability in machine-learning systems can result in less effective (or downright bad) intelligent experiences.

Intelligence APIs

To address instability in intelligence outputs, it's often helpful to encapsulate intelligence behind an interface that exposes the minimum amount of information needed to achieve the system's goals:

- When dealing with a probability, consider throwing away a bunch of resolution: round 92.333534% to 90%.
- Consider turning probabilities into classifications: instead of "45% there is a cow," say, "There is not a cow."
- Consider quantizing predictions: instead of "There is a cow at coordinates 14,92 in the image," say, "There is a cow on the left side of the image."

The thresholds and policies needed to implement these types of transformations are tightly coupled to the intelligence and should be included in the model with each update.

And while these practices certainly help, making the model's output more stable, and providing some level of encapsulation of the model's inner workings, they aren't perfect. For example, imagine the cow detector is 81% sure there is a cow in one frame. The model's threshold is at 80%. So the system creates a "there is a cow here" classification and lights up a UX element.

Then on the next frame, the cow detector is 79.5% sure there is a cow present, and the user experience flickers off. This can look really bad.

A well designed API, combined with an experience designed to minimize flaws, are both required to make an effective intelligent experience.

Summary

The intelligence runtime is responsible for: gathering the context of the system; converting this context into a form that works with the intelligence; executing the various components that make up the intelligence; combining the intelligence results and creating a good interface between the intelligence and the experience; and using the output of the intelligence to affect the user experience.

Some key components that an intelligence runtime will deal with include these:

- **The context,** including all the information relevant to making a good decision in the Intelligent System.
- **The features** (and the feature extraction code), which convert the context into a form that is compatible with the specific models that contain the system's intelligence.
- **The models,** which represent the intelligence and are typically contained in data files that will change relatively frequently during the lifetime of an Intelligent System.
- **The execution engine,** which executes the models on the features and returns the predictions. There are many great libraries to support executing models, but these will often need to be wrapped to combine intelligence; they will sometimes need to be replaced for unique execution environments.
- **The results,** which are the predictions of the intelligence. It is good practice to keep the raw results private and create an intelligence API that exposes the minimum information to power the intelligent experience, while being robust to changes over time.

An effective intelligence runtime will make it easy to track down mistakes; will execute in the runtime the same way it executes in the intelligence creation environment; will support easy changes to both models and feature extraction; and will make it hard for parts of the system to get out of sync.

For Thought…

After reading this chapter, you should:

- Be able to design a runtime that executes intelligence and uses it to power user experience.
- Understand how to structure an intelligence runtime to allow innovation in intelligence and support the other components of the intelligence implementation.

You should be able to answer questions like these:

- What is the difference between the context of an intelligence call and the features used by the machine-learned model?

Consider the Intelligent System you used most recently.

- What type of information might be in the context of the intelligence calls it makes?
- At a high level, walk through the steps to go from that context to a user-impacting experience (invent any details you need to about how the system might work).
- What are some ways this system's intelligence might be encapsulated to mitigate the effects that small intelligence changes have on the user?

CHAPTER 13

Where Intelligence Lives

When building an Intelligent System you'll need to decide where the intelligence should live. That is, where you will bring the model, the runtime, and the context together to produce predictions—and then how you will get those predictions back to the intelligent experience.

The runtime could be located in the user's device, in which case you'll need to figure out how to update models across your user base.

The runtime could be in a service you run, in which case you'll have to figure out how to get the context (or features) from the user's device to the service cheaply enough, and with low enough latency, to make the end-to-end experience effective.

This chapter discusses how to decide where your intelligence should live. It starts by discussing the considerations for deciding where intelligence should live, including latency and cost, and why these matter for various types of intelligent experiences. It then discusses common patterns for positioning intelligence across clients and services, including the pros and cons of each option.

Considerations for Positioning Intelligence

Some key considerations when deciding where intelligence should live are these:

- Latency in updating the intelligence.
- Latency in executing the intelligence.
- The cost of operating the intelligence.
- What happens when users go offline.

In general, you will have to make trade-offs between these properties. This section discusses each of them in turn.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_13

Latency in Updating

One consideration when deciding where to position intelligence is the latency you will incur when you try to update the intelligence. New intelligence cannot benefit users until it gets to their runtime and replaces the old intelligence.

The latency in updating intelligence can be very short when the runtime is on the same computer as the intelligence creation environment; the latency can be very long when the runtime is on a client computer, and the client computer does not connect to the Internet very often (and so it can only get new intelligence once in a while).

Latency in transporting intelligence to a runtime is important when:

- The quality of intelligence is evolving quickly.
- The problem you are trying to solve is changing quickly.
- There is risk of costly mistakes.

We'll discuss these situations where latency in updating can cause problems in more detail, including examples of how they can impact intelligent experiences.

Quality Is Evolving Quickly

Latency in updating intelligence matters when the quality of the intelligence is evolving quickly.

When a problem is new, it is easy to make progress. There will be all sorts of techniques to explore. There will be new data arriving every day, new users using the product in ways you didn't anticipate. There will be excitement—energy. The quality of intelligence might improve rapidly.

For example, imagine it's day one—you've just shipped your smart shoe. This shoe is designed to adjust how tight it is based on what its wearer is doing. Sitting around doing nothing? The shoe automatically relaxes, loosening the laces so the wearer can be more comfortable. But if the user starts running? Jumping? The shoe automatically tightens the laces, so their wearer will get better support and be less likely to get an injury.

So you've had a huge launch. You have tens of thousands of smart shoes in the market. You are getting telemetry for how users are moving, when they are overriding the shoe to make their laces tighter, and when they are overriding the shoe to make their laces looser. You are going to want to take all this data and produce new intelligence.

If you produce the new intelligence, test it, and determine it isn't much better than the intelligence you shipped with—you're fine! No need to worry about latency in updating intelligence, because you don't have any benefit to gain.

But if the new intelligence is better—a lot better—you are going to be desperate to get it pushed out to all the shoes on the market. Because you have users wearing those shoes. They'll be talking to their friends. Reviewers will be writing articles. You are going to want to update that intelligence quickly!

When intelligence is improving quickly, latency in deployment will get in the way of taking advantage of the intelligence. This is particularly important in problems that are very big (with many, many contexts) or problems that are very hard to solve (where quality will be improving for a long time to come).

Problem Changing Quickly

Latency in updating intelligence matters when the problem is changing quickly, because it is open-ended or changing over time.

In some domains new contexts appear slowly, over the course of weeks or months. For example, new roads take time to build; tastes in music evolve slowly; new toaster products don't come onto the market every day. In these cases having days or weeks of latency in deploying new intelligence might not matter.

On the other hand, in some domains new contexts appear rapidly, hundreds or thousands of times per day. For example, new spam attacks happen every second; hundreds of new news articles are written per day; hurricanes make landfall; stock markets crash. In these cases, it might be important to be able to deploy new intelligence fast.

There are two important aspects to how problems change:

1. How quickly do new contexts appear?

2. How quickly do existing contexts disappear?

Erosion will happen slowly in domains where new contexts are added, but old contexts remain relevant for a reasonably long time. For example, when a new road is constructed your intelligence might not know what to do with it, but your intelligence will still work fine on all the existing roads it does know about.

Erosion will happen quickly in domains where new contexts displace old ones, and the intelligence you had yesterday is no longer useful today. For example, when a new song comes onto the top of the charts, an old song leaves. When a spammer starts their new attack, they stop their old one.

When a problem changes frequently in ways that erode the quality of existing intelligence, latency in updating intelligence can be a critical factor in the success of an Intelligent System.

Risk of Costly Mistakes

Latency in updating intelligence matters when the Intelligent System can make costly mistakes that need to be corrected quickly.

Intelligence makes all kinds of mistakes. Some of these mistakes aren't too bad, particularly when the intelligent experience helps users deal with them. But some mistakes can be real problems.

Consider the smart-shoe example, where laces automatically loosen or tighten based on what the wearer is doing. Imagine that some small percent of users fidget in a particular way that makes the shoe clamp down on their feet, painfully.

Or imagine that a small percentage of users play right field on a baseball team. Ninety-nine percent of the time they are standing there, looking at the clouds go by—and then one percent of time they are sprinting wildly to try to catch a fly ball. Maybe the smart-shoes can't keep up. Maybe right fielders are running out of their shoes, slipping, and getting hurt.

When your Intelligent System's mistakes have high cost, and these costs can't be mitigated with a good user experience, latency can be very painful. Imagine users calling, complaining, crying about the damage you've caused them, day after day, while you wait for new intelligence to propagate. It can make you feel bad. It can also put your business at risk.

Latency in Execution

Another consideration for deciding where intelligence should live is the latency in executing the intelligence at runtime.

To execute intelligence, the system must gather the context, convert the context to features, transport the features to where the intelligence is located (or sometimes the whole context gets transported and feature extraction happens later), wait for the intelligence to execute, wait for the result of the intelligence execution to get back to the experience, and then update the experience. Each of these steps can introduce latency in executing the intelligence.

The latency in execution can be short when the intelligent runtime lives on the same computer as the intelligent experience; it can be long when the intelligent runtime and intelligent experience live on different computers.

Latency in intelligence execution can be a problem when:

- Users will have to wait for the latency.
- The right answer changes quickly and drastically.

Latency and its effects on users can be difficult to predict. Sometimes users don't care about a little bit of latency. Sometimes a little latency drives them crazy and totally ruins an experience. Try to design experiences where latency trade-offs can be changed easily (during orchestration) so various options can be tested with real users.

Latency in Intelligent Experience

Latency in execution matters when users notice it, particularly when they have to wait for the latency before they can continue. This can occur when:

- **The intelligence call is an important part of rendering the experience**. Imagine an application that requires multiple intelligence calls before it can properly render. Maybe the application needs to figure out if the content is potentially harmful or offensive for the user before rendering it. If there is nothing for the user to do while waiting for the intelligence call, then latency in intelligence execution could be a problem.
- **The intelligence call is interactive**. Imagine an application where the user is interacting directly with the intelligence. Maybe they throw a switch, and they expect the light to turn on instantly. If the switch needs to check with the intelligence before changing the light, and the intelligence call takes hundreds or thousands of milliseconds— users might stub their toes.

On the other hand, intelligent experiences that are intrinsically asynchronous, such as deciding whether to display a prompt to a user, are less sensitive to latency in execution.

The Right Answer Changes Drastically

Latency in executing intelligence matters when the right answer changes rapidly and drastically.

Imagine creating an intelligence to fly a drone. The drone is heading to an objective; the correct answer is to fly straight at the objective. No problem. And then someone steps in front of the drone. Or imagine the drone flying on a beautiful, blue, sunny day and then a huge gust of wind comes out of nowhere. In these situations, the right control for the drone changes, and changes quickly.

When the right course of action for an Intelligent System changes rapidly and drastically, latency in executing intelligence can lead to serious failures.

Cost of Operation

Another consideration when deciding where intelligence should live is the cost of operating the Intelligent System.

Distributing and executing intelligence takes CPU, RAM, and network bandwidth. These cost money. Some key factors that can drive the cost of an Intelligent System include:

- The cost of distributing intelligence.
- The cost of executing intelligence.

The Cost of Distributing Intelligence

Distributing intelligence costs both the service and the user money, usually in the form of bandwidth charges. Each new piece of intelligence needs to be hosted (for example on a web service), and the runtime must periodically check for new intelligence, and then download any it finds. This cost is proportional to the number of runtimes the intelligence must go to (the number of clients or services hosting it), the size of the intelligence updates, and the frequency of the updates. For Internet-scale Intelligent Systems, the cost of distributing intelligence can be very large.

It's also important to consider costs for users. If the primary use case is for users on broadband, distributing models might not be a concern—it might not cost them much. But when users are on mobile devices, or in places where network usage is more carefully metered and billed, the bandwidth for intelligence distribution may become an important consideration.

The Cost of Executing Intelligence

Executing intelligence can also have a bandwidth cost when the intelligence is located in a service and clients must send the context (or features) to the service and get the response. Depending on the size of the context and the frequency of calls, it may cost more to send context to the service than to send intelligence to the client. Keep in mind that telemetry and monitoring will also need to collect some context and feature information from clients; there is opportunity to combine the work and reduce cost.

Executing intelligence also takes CPU cycles and RAM. Putting the intelligence runtime in the client has the advantage that users pay these costs. But some types of intelligence can be very expensive to execute, and some clients (like mobile ones) have resource constraints that make heavyweight intelligence runtimes impractical. In these cases, using intelligence runtimes in the service (maybe with customized hardware, like GPUs or FPGAs) can enable much more effective intelligences.

When considering the cost of operation, strive for an implementation that:

1. Is sensitive to the user and does not ask them to pay costs that matter to them (including bandwidth, but also power in a mobile device, and so on).
2. Does let users pay the parts of the cost they won't notice (so you don't have to buy lots of servers).
3. Balances all the costs of running the Intelligent System and scales well as the number of users and quality of intelligence grow.

Offline Operation

Another consideration when deciding where intelligence should live is whether the Intelligent System needs to function when it is offline (and unable to contact any services).

It isn't always important for an Intelligent System to work when it is offline. For example, a traffic prediction system doesn't need to work when its user is in an airplane over the Pacific Ocean. But sometimes Intelligent Systems do need to work offline—for example, when the intelligence runs in a restricted environment (like a military vehicle) or in a life-critical system.

When it is important to function offline, some version of the intelligence must live in the client. This can be the full intelligence, or it can be more like a backup—a reduced version of the overall intelligence to keep users going while the service comes back online.

Places to Put Intelligence

This section explores some of the options for positing intelligence in more detail. It introduces some common patterns, including:

- Static intelligence in the product
- Client-side intelligence
- Server-centric intelligence
- Back-end (cached) intelligence
- Hybrid intelligence

This section discusses each of these approaches and explores how well they address the four considerations for where intelligence should live: latency in updating, latency in execution, cost of operation, and offline operation.

Static Intelligence in the Product

It's possible to deliver intelligence without any of this Intelligent System stuff. Simply gather a bunch of training data, produce a model, bundle it with your software, and ship it.

This is very similar to shipping a traditional program. You build it, test it as best you can—in the lab, or with customers in focus groups and beta tests—tune it until you like it, and send it out into the world.

The advantages of this is that it is cheaper to engineer the system. It might be good enough for many problems. It can work in situations without the ability to close the loop between users and intelligence creation. And there is still the possibility for feedback (via reviews and customer support calls), and to update the intelligence through traditional software updates.

The disadvantage is that intelligence updates will be more difficult, making this approach poorly suited to open-ended, time-changing, or hard problems.

Latency in Updating Intelligence: Poor

Latency in Execution: Excellent

Cost of Operation: Cheap

Offline Operation: Yes

> **Disadvantage Summary:** Difficult to update intelligence. Risk of unmeasurable intelligence errors. No data to improve intelligence.

Client-Side Intelligence

Client-side intelligence executes completely on the client. That is, the intelligence runtime lives fully on the client, which periodically downloads new intelligence.

The download usually includes new models, new thresholds for how to interpret the models' outputs (if the intelligence is encapsulated in an API—and it should be), and (sometimes) new feature extraction code.

Client-side intelligence usually allows relatively more resources to be applied to executing intelligence. One reason for this is that the intelligence can consume idle resources on the client at relatively little cost (except maybe for power on a mobile device). Another reason is that the latency of the runtime is not added to any service call latency, so there is relatively more time to process before impacting the experience in ways the user can perceive.

The main challenge for client-side intelligence is deciding when and how to push new intelligence to clients. For example, if the intelligence is ten megabytes, and there are a hundred thousand clients, that's about a terabyte of bandwidth per intelligence update. Further, models don't tend to compress well, or work well with incremental updates, so this cost usually needs to be paid in full.

Another potential complexity of client-side intelligence is dealing with different versions of the intelligence. Some users will be offline, and won't get every intelligence update you'd like to send them. Some users might opt-out of updates (maybe using firewalls) because they don't like things downloading to their machines. These situations will make interpreting user problems more difficult.

Another disadvantage of client-side models is that they put your intelligence in the hands of whoever wants to take a look at it: maybe a competitor, maybe someone who wants to abuse your service, or your users—like a spammer. Once someone has your model they can run tests against it. They can automate those tests. They can figure out what type of inputs gets the model to say one answer, and what type of inputs gets it to say another. They can find the modifications to their context (e-mail, web page, product, and so on) that trick your model into making exactly the type of mistake they want it to make.

Latency in Updating Intelligence: Variable

Latency in Execution: Excellent

Cost of Operation: Based on update rate.

Offline Operation: Yes

> **Disadvantage Summary:** Pushing complex intelligence to clients can be costly. Hard to keep every client in-sync with updates. Client resources may be constrained. Exposes the intelligence to the world.

Server-Centric Intelligence

Server-centric intelligence runs in real-time in the service. That is, the client gets the context (or features) and sends them to the server, and the server executes the intelligence on the features and returns the result to the client.

Using server-centric intelligence allows models to be updated quickly, and in a controlled fashion, by pushing new models to the server (or servers) running the intelligence. It also makes telemetry and monitoring easier because much of the data to log will already be in the service as part of the intelligence calls.

But server-centric intelligence needs to be scaled as the user base scales. For example, if there are a hundred intelligence request per second, the service must be able to execute the intelligence very quickly, and probably in parallel.

Latency in Updating Intelligence: Good

Latency in Execution: Variable, but includes Internet round-trip.

Cost of Operation: Service infrastructure and bandwidth can have significant cost; may cost users in bandwidth.

Offline Operation: No

Disadvantage Summary: Latency in intelligence calls. Service infrastructure and bandwidth costs. User bandwidth costs. Cost of running servers that can execute intelligence in real time.

Back-End (Cached) Intelligence

Back-end intelligence involves running the intelligence off-line, caching the results, and delivering these cached results where they are needed. Cached intelligence can be effective when analyzing a finite number of things, like all the e-books in a library, all the songs a service can recommend, or all the zip codes where an intelligent sprinkler is sold. But back-end intelligence can also be used when there aren't a finite number of things, but contexts change slowly.

For example, a sprinkler is sold into a new zip code. The service has never considered that zip code before, so it returns some default guess at the optimal watering time. But then the back-end kicks off, examines all the info it can find about the zip code to produce a good watering plan and adds this watering plan to its cache. The next time a sprinkler is sold in that zip code, the service knows exactly how to water there (and maybe the system even updates the watering plan for that poor first guy who kicked off the whole process).

Back-end Intelligent Systems can afford to spend more resources and time on each intelligence decision than the other options. For example, imagine a super complex watering-plan model that runs for an hour on a high-end server to decide how to water in each zip code. It analyzes satellite images, traffic patterns, the migration of birds and frogs in the regions—whatever it takes. Such a model might take months to run on an embedded computer in a sprinkler—impractical. It can't run on a server that needs to respond to hundreds of calls per second—no way. But it can run in back-end 'every so often' and the results of its analysis can be cached.

Intelligence caches can live in services; parts of them can be distributed to clients too.

One disadvantage of back-end intelligence is that it can be more expensive to change models, because all of the previous cached results might need to be recomputed.

Another disadvantage is that it only works when the context of the intelligence call can be used to "look up" the relevant intelligence. This works when the context describes an entity, such as a web page, a place, or a movie. It doesn't work when the context describes less-concrete things, like a user-generated block of text, a series of outputs from a sensor-array, or a video clip.

Latency in Updating Intelligence: Variable

Latency in Execution: Variable

Cost of Operation: Based on usage volume

Offline Operation: Partial

> **Disadvantage Summary:** Not effective when contexts change quickly, or when the right answer for a context changes quickly. Can be expensive to change models and rebuild the intelligence caches. Restricts intelligence to things that can be looked up.

Hybrid Intelligence

In practice it can be useful to set up hybrid intelligences that combine several of these approaches.

For example, a system might use a back-end intelligence to deeply analyze popular items, and a client-side intelligence to evaluate everything else.

Or a system might use a client-side intelligence in most cases, but double-check with the service when a decision has serious consequences.

Hybrid intelligences can mask the weaknesses of their various components.

But hybrid intelligences can be more complex to build and to orchestrate. Consider, if the system gives an incorrect answer, what part of the intelligence did the mistake come from? The client-side model? The intelligence that was cached in the service? Some subtle interaction between the two?

Sometimes it's even hard to know for sure what state all of those components were in at the time of the mistake.

Nevertheless, most large Intelligent Systems use some form of hybrid approach when determining where their intelligence should live.

Summary

Choosing where your intelligence will live is an important part of creating a successful Intelligent System. The location of intelligence can affect:

- **The latency in updating the intelligence:** This is a function of how far the intelligence needs to move to get from the creation environment to the runtime and how often the runtime is online to take an update.
- **The latency in executing the intelligence:** This is a function of moving the context and the features from the intelligent experience to the intelligence runtime and moving the answer back.
- **The cost of operating the Intelligent System:** This is a function of how much bandwidth you need to pay for to move intelligence, context, and features and how much CPU you need to pay for to execute intelligence.
- **The ability of the system to work offline:** This is a function of how much of the intelligence can function on the client when it can't communicate with your service components.

There are many options for balancing these properties. Here are some common patterns:

- **Static intelligence:** This puts the intelligence fully in the client without connecting it to a service at all.
- **Client-side intelligence:** This puts the intelligence fully in the client, but connects it to a service for intelligence updates and telemetry.
- **Server-centric intelligence:** This puts the intelligence fully in a service and requires a service call every time intelligence needs to be executed on a context.

- **Back-end (cached) intelligence:** This executes intelligence offline on common contexts and delivers answers via caching.
- **Hybrid intelligence:** This is the reality for most large-scale Intelligent Systems and combines multiple of the other approaches to achieve the system's objectives.

For Thought…

After reading this chapter, you should:

- Know all the places intelligence can live, from client to the service back-end, and the pros and cons of each.
- Understand the implications of intelligence placement and be able to design an implementation that is best for your system.

You should be able to answer questions like these:

- Imagine a system with a 1MB intelligence model, and 10KB of context for each intelligence call. If the model needs to be updated daily, at what number of users/intelligence call volume does it make sense to put the intelligence in a service instead of in the client?
- If your application needs to work on an airplane over the Pacific Ocean (with no Internet), what are the options for intelligence placement?
- What if your app needs to function on an airplane, but the primary use case is at a user's home? What are some options to enable the system to shine in both settings?

CHAPTER 14

Intelligence Management

The intelligence in an Intelligent System takes a journey from creation, to verification, to deployment, to lighting up for users, and finally to being monitored over time. Intelligence management bridges the gap between intelligence creation (which is discussed in Part IV of this book) and intelligence orchestration (which is discussed in Part V), by making it safer and easier to deploy new intelligence and enable it for users.

At its simplest, intelligence management might involve hand-copying model files to a deployment directory where the runtime picks them up and exposes them to users. But in large, complex Intelligent Systems, the process of managing intelligence can (and probably should) be much more involved.

This chapter will discuss some of the challenges with intelligence management. Then it will provide an overview of ways intelligence management can support agility in your Intelligent System while verifying intelligence and lighting it up with users.

Overview of Intelligence Management

Intelligence management involves all the work to take intelligence from where it is created and put it where it will impact users. This includes:

- **Sanity checking the intelligence** to make sure it will function correctly.
- **Deploying the intelligence** to the runtimes it needs to execute in.
- **Lighting up the intelligence** in a controlled fashion.
- **Turning off intelligence** that is no longer helpful.

An intelligence management system can be simple (like a set of instructions an intelligence operator must execute manually for each of these steps); it can be partially automated (like a set of command-line tools that do the work); or it can be very slick

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_14

(like a graphical console that lets intelligence orchestrators inject, move, and light up or disable intelligence with a click). A good intelligence management system will do the following:

- Provide enough support to match the skills and scenarios for intelligence management in your environment.
- Make it hard to make mistakes.
- Not introduce too much latency.
- Make a good tradeoff between human involvement and implementation cost.

Intelligence management is challenging because of complexity, frequency, and human systems. We'll discuss these in turn.

Complexity in Intelligent Management

Intelligent Systems can be quite complex. For example:

- The intelligence might need to live at multiple places between the client and server, including client-side intelligence, one or more server-side intelligences, and cached intelligence.
- Intelligence might come from dozens of different sources—some of them machine-learned, some created by humans, and some created by processes outside of your organization.
- Various parts of intelligence might depend on one another, and might be updated at different frequencies by different people.

As Intelligent Systems grow over time, simply getting a new piece of intelligence correctly deployed to users can be difficult and error-prone.

Frequency in Intelligence Management

Intelligent Systems will have their intelligence updated many times during their life-cycles, consider that:

- Updating intelligence *once a week* for three years is about a hundred sixty times.
- Updating intelligence *once a day* for three years is about a thousand times.
- Updating intelligence *once an hour* for three years is about twenty-six thousand times.
- Updating intelligence *one a minute* for three years is about one and a half million times.

These are pretty big numbers. They mean that the intelligence management process needs to be reliable (have a low error rate), and it probably can't take much human effort.

Human Systems

Intelligence might be deployed by users with all sorts of skill levels and backgrounds:

- Experts who understand the implementation of the Intelligent System well.
- Machine-learning practitioners who are not great engineers.
- Nontechnical people who are hand-correcting costly mistakes the system is making.
- New employees.
- Disgruntled employees.

Making intelligent management easier and less error-prone can pay large dividends in the agility with which your Intelligent System evolves over time.

Sanity-Checking Intelligence

A rapid, automated sanity-checking system is a safety net for intelligence creators, allowing them to innovate with confidence and focus their energy on building better intelligence (not on cross-checking a bunch of details and remembering how to run all the tests they should run). An effective safety net will verify that new intelligence:

- Is compatible with the Intelligent System.
- Executes and meets any runtime requirements.
- Doesn't make obvious mistakes.

A good intelligence management system will make it harder to deploy intelligence without doing these checks than it is to deploy intelligence via a system that automates these types of checks.

We'll now explore these categories of checks in turn.

Checking for Compatibility

Mistakes happen. Sometimes intelligence creators format things wrong, or forget to run a converter on their model files, or train a model from a corrupted telemetry file, or the training environment breaks in an odd way that outputs a corrupted model. Intelligence management is a great chokepoint where lots of simple, common mistakes can be caught before they turn into damaging problems. Here are some things to check to ensure that intelligence is compatible:

- The intelligence data file properly formatted and will load in the intelligent runtime.
- The new intelligence in sync with the feature extractor that is currently deployed.
- The new intelligence is in sync with the other intelligence in the system, or dependent intelligence is being deployed simultaneously.
- The new intelligence deployment contains all the required meta-data (such as any thresholds needed to hook it to the intelligent experience).

- The cost of deploying the new intelligence will be reasonable (in terms of bandwidth costs and the like).

These are all static tests that should be simple to automate, should not introduce much latency, and don't require much human oversight (unless there is a problem)—it is easy to determine automatically whether they pass or fail.

Checking for Runtime Constraints

It's also important to check that the new intelligence meets any constraints from the environment where it will execute, including that:

- The new intelligence doesn't use too much RAM when loaded into memory in the runtime.
- The new intelligence meets the runtime performance targets for the execution environment (across a wide range of contexts).
- The new intelligence will run exactly the same way when users interact with it as it did in the intelligence creation environment (context handling, feature creation, intelligence execution, and so on).

These tests require:

- Executing intelligence in a test environment that mirrors the environment where users will interact with the intelligence.
- A facility to load contexts, execute the intelligence on them, measure resource consumption, and compare the results to known correct answers.
- A set of test contexts that provide good coverage over the stations your users encounter.

These are dynamic tests that can be automated. They will introduce some latency (depending on how many test contexts you use). They don't require much human oversight (unless there is a problem)—it is easy to determine automatically whether they pass or fail.

Checking for Obvious Mistakes

Intelligence creators shouldn't create intelligences that make obvious mistakes. You can tell them that (and I'll tell them that later in this book)—but it never hurts to check. Intelligence management should verify that:

- The new intelligence has "reasonable" accuracy on a validation set (contexts that the intelligence creators never get to see—no cheating).
- The new intelligence doesn't make any mistakes on a set of business-critical contexts (that should never be wrong).
- The new intelligence doesn't make significantly more costly mistakes than the previous intelligence did.
- The new intelligence doesn't focus its new mistakes on any critical sub-population of users or contexts.

If any of these tests fail, the intelligence deployment should be paused for further review by a human.

These are dynamic tests that can be automated. They will introduce some latency (depending on how many test contexts you use). They are somewhat subjective, in that humans may need to consider the meaning of fluctuations in accuracy over time.

Lighting Up Intelligence

Once intelligence is sanity-checked against a series of offline checks, it can be checked against real users. Ways of doing this include the following:

- Single Deployment
- Silent Intelligence
- Controlled Rollout
- Flighting
- Reversion

This section will discuss these as well as some of their pros and cons, so you can decide which is right for your Intelligent System.

Single Deployment

In the simplest case, intelligence can be deployed all at once to all users simultaneously in any of several ways:

- By bundling the new intelligence into a file, pushing the file to the runtimes on clients, and overwriting the old intelligence.
- By copying the new intelligence onto the server that is hosting the runtime and restarting the runtime process.
- By partitioning the intelligence into the part that runs on the client, the part that runs on the service, and the part that runs on the back-end, and deploying the right pieces to the right places.

Pushing the intelligence all at once is simple to manage and relatively simple to implement. But it isn't very forgiving. If there is a problem, all of your users will see the problem at once.

For example, imagine you've built a smart clothes-washing machine. Put in clothes, shut the door, and this machine washes them—no more messing with dials and settings. Imagine the system is working well, but you decide to improve the intelligence with a single deployment. You push a new intelligence out to tens of thousands of smart washing machines—and then start getting reports that the washing machines are ruining users' clothes. How is it happening? You aren't sure. But the problem is affecting all your users and you don't have a good solution.

Single Deployment can be effective when:

- You want to keep things simple.
- You have great offline tests to catch problems.

Single deployment can be problematic when:

- Your system makes high-cost mistakes.
- Your ability to identify and correct problems is limited/slow.

Silent Intelligence

Silent intelligence deploys new intelligence in parallel to the existing intelligence and runs both of them for every interaction. The existing intelligence is used to control the intelligent experience (what users see). The silent intelligence does not affect users; its predictions are simply recorded in telemetry so you can examine them and see if the new intelligence is doing a good job or not.

One helpful technique is to examine contexts where the existing intelligence and the silent intelligence make different decisions. These are the places where the new intelligence is either better or worse than the old one. Inspecting a few hundred of these contexts by hand can give a lot of confidence that the new intelligence is safe to switch on (or that it isn't).

Intelligence can be run in silent mode for any amount of time: a thousand executions, a few hours, days, or weeks; as long as it takes for you to gain confidence in it.

If the new intelligence proves itself during the silent evaluation, it can replace the previous intelligence. But if the new intelligence turns out to be worse, it can be deleted without ever impacting a user—no problem!

Silent intelligence can be effective when:

- You want an extra check on the quality of your intelligence.
- You want to confirm that your intelligence gives the same answers at runtime as it did when you created it.
- You have a very big or open-ended problem and you want to gain confidence that your intelligence will perform well on new and rare contexts (which may not appear in your intelligence-creation environment).

Silent intelligence can be problematic when:

- You don't want the complexity (or resource cost) of running multiple intelligences at the same time.
- Latency is critical, and you can't afford to wait to verify your new intelligence in silent mode.
- It is hard to evaluate the effect of the silent intelligence without exposing it to users—you can see what it would have done, but not the outcome the user would have gotten.

Controlled Rollout

A controlled rollout lights up new intelligence for a fraction of users, while leaving the rest of the users with the old intelligence. It collects telemetry from the new users and uses it to verify that the new intelligence is performing as expected. If the new intelligence is good, it is rolled out to more users; if the new intelligence has problems, it can be reverted without causing too much damage.

This is different from silent intelligence in two important ways:

1. Telemetry from a controlled rollout includes the effect the intelligence has on user behavior. You can know both what the intelligence did and how users responded.
2. A controlled rollout runs a single intelligence per client; but runs multiple intelligences across the user base—it uses fewer resources per client, but may be more complex to manage.

Intelligence can be rolled out using various policies to balance latency and safety, including:

- Rolling out to an additional small fraction of your users every few hours as long as telemetry indicates things are going well.
- Rolling out to a small test group for a few days, then going to everyone as long no problems were discovered.
- Rolling out to alpha testers for a while, and then to beta testers, then to early adopters, and finally to everyone.

A controlled rollout can be effective when:

- You want to see how users will respond to a new intelligence while controlling the amount of damage the new intelligence can cause.
- You are willing to let some of your users experience problems to help you verify intelligence.

A controlled rollout can be problematic when:

- You don't want to deal with the complexity of having multiple versions of intelligence deployed simultaneously.
- You are worried about rare events. For example, a controlled rollout to 1% of users is unlikely to see a problem that affects only 1% of users.

Flighting

Flighting is a special type of controlled rollout that gives different versions of the intelligence to different user populations to answer statistical questions about the intelligences.

Imagine two intelligence creators who come to you and say they have a much better intelligence for your Intelligent System. One of the intelligences is fast but only so-so on the accuracy. The other is very slow, but has much better accuracy.

Which is going to do a better job at achieving your Intelligent System's objectives? Which will users like more? Which will improve engagement? Which will result in better outcomes?

You could do focus groups. You could let the intelligence creators argue it out. Heck, you could give them battle axes, put them in an arena and let the winner choose which intelligence to ship…

Or you could deploy each version to 1,000 of your customers and track their outcomes over the following month.

- Does one of the trial populations use the app more than the other?
- Did one of the trial populations get better outcomes than the other?
- Does one of the trial populations have higher sentiment for your app than the other?

A flight can help you understand how intelligence interacts with the rest of your Intelligent System to achieve objects.

Flights can be effective when:

- You are considering a small number of large changes and you want to know which of them is best.
- You need to track changes over an extended period so you can make statistically valid statements about how changes affect outcomes, leading indicators, and organizational objectives.

Flights can be problematic when:

- You need to iterate quickly and make many changes in a short time.
- The difference between the options you are considering is small (as when one algorithm is a half percent more accurate than another). Flights can take a long time to determine which small change is best.

Turning Off Intelligence

No matter how safe you think you are, sometimes things will go wrong, and you might have to undo an intelligence change—fast!

One way to do this is to redeploy an old intelligence over a new one that is misbehaving.

Another approach is to keep multiple versions of the intelligence near the runtime—the new one and several old ones. If things go wrong, the runtime can load a previous intelligence without any distribution latency (or cost).

Support for quick reversion can be effective when:

- You're human (and thus make mistakes).
- The cost of deploying intelligence is high.

Support for quick reversions can be problematic when:

- You're trying to impress someone and don't want them to think you're a wimp.
- Your intelligence is large, and you don't have capacity to store multiple copies of it near the runtime.

Summary

Intelligence management takes intelligence from where it is created to where it will impact users. A good management system will make it very easy to deploy intelligence and will make it hard to make mistakes. It must do both of the following:

- Sanity-check the intelligence; that is, perform basic checks to make sure the intelligence is usable. These include making sure it will run in the runtime, it will be performant enough, and it doesn't make obvious terrible mistakes.
- Light up the intelligence, which includes providing controls for intelligence to be presented to users in a measured fashion, to see what the intelligence might do, to see some small percentage of users interact with it—and to revert it quickly if there is a problem.

A successful intelligence-management system will make it easy to deploy intelligence with confidence.

It will help intelligence creators by preventing common mistakes, but also by letting them verify the behavior of their intelligence against real users in a measured fashion.

And a good intelligence-management system will support the operation of the Intelligence Service over its lifetime.

For Thought…

After reading this chapter, you should:

- Be able to design a system to manage the intelligence in an Intelligent System.
- Know ways to verify intelligence to ensure that it is compatible, works within constraints, and doesn't make obvious mistakes.
- Be prepared with a collection of ways to roll out intelligence changes safely, ensuring that the intelligence is doing what it was intended to do.

You should be able to answer questions like these:

- Design a system for managing intelligence for an Intelligence Service where the intelligence changes monthly. What tools would you build? What facilities would you create for rolling out the intelligence to users?
- Now imagine the intelligence needs to change twice per day. What would you do differently?

CHAPTER 15

Intelligent Telemetry

A telemetry system is responsible for collecting observations about how users are interacting with your Intelligent System and sending some or all of these observations back to you.

For example, a telemetry system might collect information every time any of the following happens:

- The user visits a particular part of the application.
- The user clicks a particular button.
- There is an unacceptably long loading time.
- A client connects to a server.
- The server sees a malformed query.
- The server is low on RAM.

Telemetry is used to verify that a system is working the way it is supposed to. In an Intelligent System, telemetry also contains information about the interactions users had with the intelligence and the outcomes they achieved. These are the signals intelligence creators need to improve intelligence.

Why Telemetry Is Needed

There are three main tasks for telemetry in an Intelligent System:

1. Making sure the system is working the way it is supposed to.
2. Making sure users are getting the intended outcomes.
3. Gathering data to create new and better intelligence.

We will explore each of these in more detail.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_15

Make Sure Things Are Working

Telemetry should contain data sufficient to determine that the intelligence implementation is (or isn't) working. This should include:

- That the intelligence is flowing where it is supposed to, when it is supposed to.
- That the runtime is properly loading and interpreting the intelligence.
- That contexts are being properly collected at the runtime.
- That contexts are being properly converted into features.
- That the runtime is executing models reasonably, without errors.
- That the predictions of various intelligence sources are being combined as intended.
- That the outputs from intelligence are being interpreted correctly and showing the right user experiences.
- And so on.

These are the types of things you would want to include in the telemetry of just about any service or application (and most services will want telemetry on many more things, like: latency, uptime, costs, and utilization).

One important use for this type of telemetry in an Intelligent System is to ensure that the intelligence runtime environment is behaving the same way the intelligence creation environment is; that is, that the intelligence is being used in same conditions that it was created in. These types of bugs are very hard to find: the whole system is running, the user is interacting, nothing is outputting an error, nothing is crashing, but user outcomes aren't as good as they could be because sometimes the intelligence is simply making extra mistakes that it doesn't need to.

Understand Outcomes

Telemetry should contain enough information to determine if users are getting positive or negative outcomes and if the Intelligent System is achieving its goals. It should answer questions like these:

- Which experiences do users receive and how often do they receive them?
- What actions do users take in each experience?
- What experiences tend to drive users to look for help or to undo or revert their actions?
- What is the average time between users encountering a specific experience and leaving the application?
- Do users who interact more with the intelligent part of the system tend to be more or less engaged (or profitable) over time?

This type of telemetry should be sufficient to ensure that the system's experiences are effectively connecting users to intelligence, that users are being directed where they are having better interactions, and that they are behaving in ways that indicate they are getting good outcomes.

For example, imagine a system to help doctors identify broken bones in x-rays. To help understand outcomes, the telemetry should capture things like:

- How long doctors look at X-rays where the system thinks there is a broken bone.
- How long doctors look at X-rays where the system thinks there is not a broken bone.
- How many times doctors order treatment when the system thinks there was a broken bone.
- How many times doctors order treatment when the system thinks there is not a broken bone.
- How many times patients are re-admitted for treatment on a broken bone that the system flagged, but the doctor decided not to treat.

This type of telemetry should help you understand if patients are getting better outcomes because of the intelligence or not. It should also help you diagnose why.

For example, if the intelligence is correctly identifying very subtle breaks, but doctors aren't performing treatment—why is it happening?

Maybe patients are getting bad outcomes because the intelligence makes lots of mistakes on subtle breaks so doctors don't trust that part of the system and are ignoring it.

Or maybe patients are getting bad outcomes because doctors simply aren't noticing the user experience that is trying to point out the breaks to them—it needs to be made more forceful.

Telemetry on user outcomes should help you identify and debug problems in how intelligence and experience are interacting with users, and take the right corrective actions.

Gather Data to Grow Intelligence

The telemetry should also include all the implicit and explicit user feedback needed to grow the intelligence.

This includes:

- The actions users took in response to the intelligent experiences they saw.
- The ratings the users left on content they interacted with.
- The reports users provided.
- The escalations the users made.
- The classifications users provided.
- And all the implicit indications that the user got a good or a bad outcome.

Review Chapter 9, "Getting Data from Experience," for more details.

In order for this type of telemetry to be useful, it must contain all of the following:

1. The context the user was in at the time of the interaction.
2. The prediction the intelligence produced and what experience it led to.
3. The action the user took.
4. The outcome the user got (either implicitly or through explicit feedback).

And it must be possible to connect these four components of a single interaction to each other in the intelligence-creation environment. That is, it must be possible to find a particular interaction, the context the user was in, the prediction the intelligence made, the action the user took, and the outcome the user got from the interaction—all at the same time.

This type of telemetry is the key to making an Intelligent System that closes the loop between usage and intelligence, allowing the users to benefit from interacting with intelligence, and allowing the intelligence to get better as users interact with it.

Properties of an Effective Telemetry System

Of course telemetry is good. You want more of it—heck, intelligence creators are going to want all of it. They are going to want every last detail of every last interaction (no matter the cost). And if you don't give it to them they will probably complain. They'll ask you why you are trying to ruin them. They'll wonder why you are trying to destroy their beautiful Intelligent System.

(Not that I've done that. I'm just saying. It could happen…)

This section discusses ways to balance the cost and value of telemetry. These include:

- Sampling
- Summarizing
- Flexible targeting

Sampling

Sampling is the process of randomly throwing away some data. That is, if there are 10,000 user interactions with the system in a day, a telemetry system sampling at 1% would collect telemetry on 100 of the interactions (and collect nothing on the remaining 9,900 interactions).

This sampled data is cheaper to collect, easier to store, faster to process, and can often answer key questions about the system "well enough."

The simplest form of sampling is uniform. That is, whatever type of event happens, sample 1% of them. But sampling is often targeted differently for different types of events. For example, one policy might be to sample telemetry related to verifying the implementation at 0.01% and sample telemetry related to growing intelligence at 10%, except for telemetry from France (where the intelligence is struggling) which should be sampled at 100%.

Common ways to define populations for sampling include:

- Separate by geography—for example, sample 20% of the events in Europe and 1% in Asia.
- Separate by user identity—for example, sample 0.01% of the events for most users, but sample 100% of the events for these 10 users who are having problems.
- Separate by outcome (what the user did after seeing the intelligent experience)—for example, sample at 2% for users who had normal outcomes and at 100% for users who had a problem.
- Separate by context (what the user was seeing before the intelligence fired)—for example, sample 10% of the events that occur at night and 0.1% that occur during the day.
- Separate by user properties (gender, age, ethnicity, and so on)—for example, sample at 80% for new users and at 1% for long-time users.

One particularly useful idiom is to sample 100% of data from 0.1% of users, so you can track problems that occur across the course of a session; and also sample 1% of events from 100% of users, so you can track problems that occur in aggregate.

And note: the telemetry system must record the sampling policy used to gather each piece of telemetry it collects, so the intelligence creator can correct for the sampling method and produce accurate results when they use the telemetry.

Summarizing

Another approach to reducing telemetry data is summarization. That is, taking raw telemetry, performing some aggregations and filtering of it, and storing a much smaller summary of the data.

For example, imagine you want to know how many users undo an automated action per day. You could keep raw telemetry around and calculate the count whenever you need it. Or you could calculate the count every day, save it, and delete the raw telemetry. The size of raw telemetry grows with usage (and can get very big); the size of a count is constant no matter how many users you have.

Summaries can be very effective for understanding how the system is working and if it is achieving its goals. And as you come to understand your system, and identify critical summaries, you can transmit and retain less and less raw telemetry over time.

Summarization can be done on the client or on the service.

To perform *client-side summarization*, the client simply collects telemetry for some period of time, aggregating as the telemetry arrives, and periodically reports the summarized results to the telemetry server (where they may be combined with aggregations from other clients).

Client-side summarization is useful when bandwidth is a big driver in cost.

Server-side summarization is useful when bandwidth for telemetry is either free (it is a byproduct of how the client accesses server-side intelligence) or when storage is a serious cost driver. In this case, a summarization job can be run periodically to extract all the relevant measurements. Once complete, the raw data is purged to save space.

Flexible Targeting

There are many types of telemetry in a system; some types are more valuable than others, and the value of a particular type of telemetry can change over time.

For example, early in the system development, before all the code is working properly, telemetry about implementation details—features, model deployments, and so on—can be extremely useful. But once the system is verified and deployed to users, and runs successfully for months, this type of telemetry becomes less useful.

And once the system starts attracting a lot of users, telemetry that helps grow the intelligence becomes more important: it captures all the value of users' interactions, their judgments, their preferences. While the intelligence is growing, this type of telemetry is the core value of the system, the key to achieving success.

And once the intelligence plateaus (if it does), additional training data might become less valuable and telemetry on user outcomes or rare problems might become relatively more important.

But when something goes wrong, all bets are off and telemetry from all across the system could be critical to finding and correcting the problem.

An intelligence telemetry system should support different collection policies for different types of telemetry, and these policies should be changeable relatively quickly and easily—so orchestrators and intelligence creators can do their work.

Common Challenges

This section discusses some common data pitfalls and the role that a telemetry system can have in dealing with them. These include:

- Bias
- Rare events
- Indirect value of telemetry
- Privacy

Bias

Bias occurs when one type of event appears in the telemetry more than it should. For example, imagine getting 10,000 examples of users interacting with content on your Intelligent System. Ten thousand is a good number. That should be enough to understand what is going on, to create some good intelligence.

But what if you find out that all ten thousand interactions were with the same piece of content. Like 10,000 different users interacting with a single news story?

Well, that's not as good.

Or what if you find out all 10,000 samples came from just one user interacting with the service. For some reason, the telemetry system decided to sample just that one user over and over.

That's not good either.

Here are some reasons bias might occur:

1. **The world changes, but the sampling policy doesn't**. You set up a sampling policy that selects 1,000 users to sample at 100% (so you can see end-to-end interactions). Then your user base grows by a factor of 10 and significantly changes, including less technical users, users in other countries, and so on. The initial 1,000 users may have been right at the time you selected them, but they are no longer representative of your user population—this causes bias.

2. **Users behaving in ways you don't expect**. For example, a user who is having a bad experience might simply turn off their computer. If your telemetry system expects them to shut down the app before sending telemetry, you might miss out on their data.

One approach to dealing with bias is to always collect a raw sample across all usage—for example, 0.01% of all interactions—and cross-check your other samples to make sure they aren't wildly different from this baseline sample.

Rare Events

In some settings many events are rare events; a few contexts are prominent, but most contexts your users encounter are infrequent.

For example, in a library, the most popular 100 books might account for 50% of the checkout volume, but after those 100 books, usage becomes highly dispersed, with thousands of books that each get one or two checkouts per month. When a book comes up to the checkout stand, there is a 50% chance it is one of the top books and a 50% chance it is one of the other twenty thousand books.

Imagine a system to help recommend library books in this library.

In order to recommend a book, the system needs to see which other books it is commonly checked out with. But the chances that any pair of rarely used books are checked out at the same time is very, very low.

For example, if the telemetry is created by sampling 10% of the checkout events, after a month you might have:

- 10,000 samples of the top 100 books being checked out.
- 5,000 books with just one or two checkout events.
- 15,000 books with no checkout events at all.

Not great. That kind of data might help you build recommendations for the popular books (which is simple, and you could have probably done it by hand much cheaper), but it won't give you any information on the unpopular books.

One option is stratified sampling. Maybe the top 100 books are sampled at 1%, while the remaining (unpopular) books are sampled at 100%. The amount of data might be similar to using the uniform 10% sampling rate for all books, but the value of the data might be much, much greater for producing the type of intelligence this system needs to produce.

Indirect Value

It's often difficult to quantify the value of telemetry, but quite easy to quantify the cost. Rational people will ask questions like:

- Can we turn the sampling of this particular type of data down from 10% to 1%?
- Can we stop getting any telemetry on this use-case that we don't have any plans to ever add intelligence to?
- Do we really need to store 90 days of historical data, or would 30 do?

These are all reasonable questions, and they are difficult to answer. Here are some approaches that can help:

1. Keep a very small raw sample of all activity, sufficient to do studies to estimate the value of other telemetry.
2. Make it easy to turn on and off telemetry, so that intelligence orchestrators can turn telemetry up quickly to track specific problems, and intelligence creators can try new approaches for short periods of time to see if they are worth the cost.
3. Set a reasonable budget for innovation, and be willing to pay for the potential without micromanaging each specific detail and cost.

Privacy

Privacy, ah privacy.

When using customer data, it is important to "do the right thing" and treat your user's data the way you would want someone else to treat your data. Keep in mind some best practices:

- Make sure customers understand the value they are getting from your product and feel they are getting a good deal.
- Try as much as possible to scrub any personal information (like names, credit card, social security, addresses, user generated content, and so on) out of telemetry you store.

- Consider an aggregation policy for data you intend to store long term that combines information from many users to make it harder to identify any single user's data (for example, storing aggregate traffic to each piece of content instead of storing which content each user browsed).
- Have a reasonable retention policy and don't keep data for too long.
- Make sure telemetry is handled with utmost care from hacks and accidental leaks.
- Make sure customers know what you are storing and how to opt out.
- Use data for the purpose it was collected, and don't try to re-purpose it for wildly different activities.
- Make sure you know the laws that apply in the countries where your service is operating.

But remember: you need data—you can't create intelligence without data…

Summary

Good telemetry is critical to building an effective Intelligent System. No way around it: without telemetry you won't have an Intelligent System.

Telemetry is used for three main purposes:

1. To make sure the system is working the way it is intended to work—that there aren't any bugs or problems.
2. To make sure that users are getting the outcomes you want them to get—that everything is coming together for success.
3. To grow the intelligence over time—that the contexts and outcomes are being recorded in a way you can use to create intelligence.

The value of these types of telemetry will change over time, and a system should be able to adapt. A good telemetry system will:

- Support sampling that allows tracking specific focused problems, and also view aggregate effects.
- Allow summarization of events.
- Be flexible in what it measures (and what it doesn't).

Some common pitfalls for telemetry include:

- Bias that makes the telemetry not representative of what users are experiencing.
- Rare events that are important but are very hard to capture in sampled telemetry.
- Proving the value of the telemetry in the face of reasonable questions about the associated costs.
- Protecting user privacy while still allowing intelligence to grow.

For Thought…

After reading this chapter, you should:

- Understand how telemetry enables Intelligent Systems by making sure they are working, achieving their objectives, and allowing them to improve as they are used.
- Be able to design a telemetry system that meets the needs of your application.

You should be able to answer questions like these:

Consider an Intelligent System that you think one of your friends would like to use.

- If it's too expensive to collect all the possible telemetry, how would you limit the telemetry you do collect?
- What facilities would you need to drill in when there are specific problems?

PART IV

Creating Intelligence

Chapters 16-21 explore the ways intelligence is created. This part will explain all the places intelligence might come from (including machine learning) and the pros and cons of each. It will explore goals of intelligence creation at various points in the Intelligent System's life-cycle. It will provide insight on how to organize and control intelligence creation in team environments.

This part will not teach specific machine-learning techniques in detail but will explain the key concepts and elements that support machine-learning techniques (and other intelligence-creation techniques), allowing teams to achieve success when developing large, complex Intelligent Systems.

CHAPTER 16

Overview of Intelligence

So you have an Internet smart-toaster and you need to decide how long it should toast; or you have a break-time application and you need to decide when to give users a break; or you have a funny web page app and you need to decide what's funny. We call the component of a system that makes these types of decisions the "intelligence." The previous parts of this book helped you identify when you need intelligence, how to connect it to users through intelligent experiences, how to implement it, and where it should live. This part of the book will help you create intelligence.

Intelligence maps from a context to a prediction about the context. For example, an intelligence might:

- Map from the usage history of a web site to an estimate of the usage in the next week.
- Map from an email message to the probability the email message contains a scam.
- Map from an image to an estimate of how many cucumbers are in the image.

This chapter explores the concepts of context and prediction in more detail. Later chapters will discuss how to create intelligence, how to evaluate it, how to organize it—and more.

An Example Intelligence

But first, let's look at an example of intelligence in more detail. Imagine a pellet griller.

A what?

Well, a pellet griller is sort of like a normal outdoor barbecue, but instead of having a big fire that you light and then wait to get to perfect temperature and then put your food on and flip and flip the food and hope the heat isn't too hot or too cold on account

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_16

of you waited too long or you didn't wait long enough to let the fire get to the perfect temperature for cooking… A pellet griller has a bin full of little wooden pellets and it drops them into the flame one at a time as needed to keep the heat at a perfect temperature.

Incredible.

So the intelligence in a pellet griller needs to decide when to add pellets to the fire to keep temperature. Let's break this down a little bit and explore the context of this intelligence and the predictions.

The pellet griller's context might include this information:

- The current temperature in the grill.
- The temperature that was in the grill 1 minute ago.
- The number of wood pellets added in the past 5 minutes.
- The number of wood pellets added in the past 1 minute.
- The number of wood pellets added in the past 20 seconds.
- The air temperature outside the grill.
- The type of wood in the pellets.
- The time of day.
- And so on.

These are the properties that might be relevant to the task of keeping the grill at the perfect temperature. Some of them are obviously important to achieving success (like the current temperature in the grill), and some may or may not help (like the time of day). An intelligence doesn't have to use everything from the context to make its decisions—but it can.

Based on the information in the context, the intelligence will make predictions. Show it a new context, and the intelligence will predict something about that context. The pellet griller's intelligence might try to predict:

- If the temperature inside the grill will be hotter or colder one minute in the future.
- What the exact temperature inside the grill will be one minute in the future.
- The probability it should add a pellet to the fire right now to maintain the desired temperature.

Then the intelligent experience would use these predictions to automate the process of adding fuel to the fire. If the fire is going to be too cold one minute in the future—add some fuel. Easy!

Designing the right context and choosing the best thing to predict are important parts of creating effective Intelligent Systems—and getting them right usually requires some iteration and experimentation.

Contexts

The context includes all the computer processable information your intelligence might use to make its decisions. The intelligence is free to pick and choose which pieces of the context to use to make the best decisions. In that sense, the context is the palette of options the intelligence creator can choose from when creating intelligence.

In order to be useful, the context must be:

- Implemented in the intelligence runtime.
- Available to the intelligence creator to use in producing and evaluating intelligence.

Implemented at Runtime

To make a piece of information available for the intelligence, someone needs to do the work to hook the information into the intelligence runtime.

For example, the pellet griller might be smarter if it knew the temperature outside of the grill. But it takes work to know the temperature outside of the grill. Someone needs to bolt a temperature sensor onto each grill, test it, run some wires, write a driver, poll the sensor and copy its reading into the intelligence runtime every few seconds.

Common forms of context include these:

- **Information about what is going on in the system at the time**: For example, what other programs are running, what is the current status of the things the system can control (are the lights on or off?), and what is the user seeing on their screen? Using this type of context is generally straightforward, but it does need to be implemented.

- **Properties of the content the user is interacting with:** For example, what genre is the song, what is its name, where did it come from, what words are on the web page, and what pixels are in the picture? This type of context usually requires processing the content to extract information from it or doing lookups to learn more about the content (such as looking up properties about a song in an external database).
- **History of user interactions:** How has the user has interacted with the Intelligent System in the past? To use these interactions, the system needs to monitor usage, aggregate it, and persist it over time.
- **Properties about the user:** These include age, interests, gender. To use these properties, the system needs to gather the information from the user and store it.
- **Any relevant sensor readings:** Which require the sensors to be implemented into the hardware and sensor readings to be provided to the runtime.

Deciding what to put into the context requires balancing the value of the information with the cost of implementation.

One best practice is to start by creating a context that includes everything that is cheap to include—the information that is already near the intelligence runtime as part of building the system in the first place, such as information about what is going on in the system, and properties of the content the user is interacting with.

This might be enough to get going. But expect to continually augment the context as you push to better and better intelligence.

It can also be useful to add some speculative things to the context. For example, maybe it would help to know how long the grill has been on. Maybe near the beginning of a grilling session the metal of the grill is cold and it takes more fuel to heat the thing up; maybe after one hour of grilling everything is so hot that you don't need as much fuel. Or maybe it doesn't matter.

Including information like this in the context can help your users show you what matters. If the variable is relevant to getting good outcomes, you'll see it in the telemetry and add it to the intelligence. If not, you can always remove it from the context later.

Available for Intelligence Creation

To create effective intelligence from context, someone needs to do the work to get the information into the intelligence creator's hands—probably in the form of telemetry.

Say you've shipped tens of thousands of pellet grillers. Your intelligence is running all over the world. Pellets are being added to fires in Florida, in Norway, in Zimbabwe. Grill temperatures are going up and grill temperatures are going down. How are you going to use all of this information to create better intelligence?

Someone is going to have to collect all the grilling contexts and get it to you. And the contexts need to be connected to the outcomes. For example, imagine a grilling session taking place in someplace-USA. If the pellet griller's intelligence runtime knows:

- The current temperature in the grill is 290 degrees.
- The temperature that was in the grill 1 minute ago was 292 degrees.
- The number of pellets added in the past 5 minutes was 7.
- The number of pellets added in the past 1 minute was 2.
- The number of pellets added in the past 20 seconds was 0.
- The air temperature outside of the grill is 82 degrees.
- And the outcome: The temperature in the grill gets 3 degrees hotter over the following minute.

And you collect information like this from tens of thousands of grilling sessions, and get it all back to your intelligence creators—you are going to be able to make some fantastic grilling intelligence.

Data for creating intelligence doesn't have to come from your users, but, as we've discussed in chapters on getting data from experience and telemetry, an Intelligent System works best when the data does come from actual usage.

The data used to create intelligence must be very similar to the data that is present in the intelligence runtime. If they are out of sync, the intelligence will behave differently for you and for your customers—which could be a big problem.

Things Intelligence Can Predict

Intelligence can make predictions about the contexts it encounters. These predictions are usually:

- **Classifications** of the context into a small set of possibilities or outcomes.
- Estimations of **probabilities** about the context or future outcomes.
- **Regressions** that predict numbers from the context.
- **Rankings** which indicate which entities are most relevant to the context.
- Hybrids and combinations of these.

This section will explore these concepts in more detail.

Classifications

A classification is a statement from a small set of possibilities. It could be a statement about the context directly, or it could be a prediction of an outcome that will occur based on the context. For example:

- Based on the context, classify the grill as:
    - Too hot
    - Too cold
    - Just right
- Based on the context, classify the movie as:
    - A horror flick
    - A romantic comedy
    - An adventure
    - A documentary

- Based on the context, classify the picture as:
    - A cow
    - A red balloon
    - Neither a cow or a red balloon—something else

Classifications are commonly used when there are a small number of choices: two, five, a dozen.

Classifications are problematic when:

- There are many possible choices—when you have hundreds or thousands of possibilities. In these situations you might need to break up the problem into multiple sub-problems or change the question the intelligence is trying to answer.
- You need to know how certain the prediction is—for example, if you want to take an action when the intelligence is really certain. In this case, consider probability estimates instead of classifications.

Probability Estimates

Probability estimations predict the probability the context is of a certain type or that there will be a particular outcome. Compared to classifications, a probability estimation is less definitive, but more precise. A classification would say "it is a cat"; a probability estimation would say "it is 75% likely to be a cat."

Other examples of probability estimates include these:

- The web page is 20% likely to be about politics, 15% likely to be about shopping, 10% likely to be a scam, and so on.
- The user is 7% likely to click accept if we offer to reformat their hard drive.
- There is a 99% chance it will rain next week.

Probability estimation are commonly used with one or more thresholds. For example:

```
if(predictedProbability > 90%)
{
     IntelligentExperience->AutomateAnAction();
}
else if(predictedProbability > 50%)
{
     IntelligentExperience->PromptTheUser();
}
else
{
     // Do nothing...
}
```

In this sense, probabilities contain more information than classifications. You can turn a probability into a classification using a threshold, and you can vary the threshold over time to tune your intelligence.

Most machine-learning algorithms create probabilities (or scores, which are similar to probabilities) internally as part of their models, so it is very common for Intelligent Systems to use probability estimates instead of classifications.

Probability estimations are problematic when:

- As with classifications, probabilities don't work well when there are many possible outcomes.
- You need to react to small changes: Slight changes in the context can cause probabilities to jitter. It is common to smooth the output of multiple sequential probabilities to reduce jitter (but this adds latency). It is also good practice to quantize probabilities unless you really, really need the detail.

Also note that probabilities usually aren't actually probabilities. They are more like directional indicators. Higher values are more likely; lower values are less likely. An intelligence might predict 90%, but that doesn't mean the outcome will happen 9 out of 10 times—unless you carefully calibrate your intelligence. Be careful when interpreting probabilities.

Regressions

Regressions are numerical estimates about a context, for example:

- The picture contains 6 cows.
- The manufacturing process will have 11 errors this week.
- The house will sell for 743 dollars per square foot.

Regressions allow you to have more detail in the answers you get from your intelligence. For example, consider an intelligence for an auto-pilot for a boat.

- A classification might say, “The correct direction is right.”
- A probability might say, “The probability you should turn right is 75%.”
- A regression might say, “You need to turn 130 degrees right.”

These convey very different information. All three say “right,” but the regression also conveys that you have a long way to go – you better start spinning that wheel!

Regressions are problematic when:

- **You need to react to small changes:** Slight changes in the context can cause regressions to jitter. It is common to smooth the output of multiple sequential regressions to reduce jitter (but this adds latency).
- **You need to get training data from users:** It is much easier to know “in this context, the user turned right” than to know “in this context the user is going to turn 114 degrees right.”

Classifications can be used to simulate regressions. For example, you could try to predict classifications with the following possibilities:

- “Turn 0 - 10 degrees right.”
- “Turn 11 - 45 degrees right.”
- “Turn 46 - 90 degrees right.”
- And so on.

This quantizes the regression and may be simpler to train and to use.

Rankings

Rankings are used to find the items most relevant to the current context:

- Which songs will the user want to listen to next?
- Which web pages are most relevant to the current one?
- Which pictures will the user want to include in the digital scrap-book they are making?

Ranking have been successfully used with very large numbers of items, like every web page on the Internet, every movie in a digital media service, or every product in an e-commerce store.

Rankings are commonly used when there are many possible relevant entities and you need to find the top few.

Rankings can be thought of using probabilities. Take each item, find the probability it is relevant to the current context, and "rank" the items in order of the probability estimates (but actual ranking algorithms are more complex than this).

Hybrids and Combinations

Most intelligences produce classifications, probability estimations, regressions, or rankings. But combinations and composite answers are possible.

For example, you might need to know where the face is in an image. You could have one regression that predicts the X location of the face and another that predicts the Y location, but these outputs are highly correlated—the right Y answer depends on which X you select, and vice versa. It might be better to have a single regression with two simultaneous outputs, the X location of the face and the Y location.

Summary

Intelligence is the part of the system that understands the contexts users encounter and makes predictions about the contexts and their outcomes.

Context is all the information available to the intelligence to make its determinations. An intelligence doesn't have to use all the parts of the context, but it can.

In order to be used as part of the context, information needs to be hooked into the intelligence runtime, and it needs to be available to the intelligence creators.

The intelligence runtime and the information available to intelligence creators must be exactly the same. Any differences could lead to hard-to-find problems.

Intelligence can give many forms of predictions, including:

- Classifications, which map contexts to a small number of states.
- Probability estimations, which predict the probability a context is in a particular state or a particular outcome will occur.
- Regressions, which estimate a number from contexts.
- Rankings, which find content that is relevant to the current context.
- Some hybrid or combination that combines one or more of these.

For Thought…

After reading this chapter, you should:

- Be able to describe a context as used in intelligence, including how to enable it in an intelligence runtime and in training.
- Understand the types of predictions an intelligence can make and when the various options are strong or weak.

You should be able to answer questions like these:

- Choose the Intelligent Systems you like best and come up with 20 things that might be part of its context.
- Which of the 20 things you used in your last answer would be the hardest to use as context in an Intelligent System? Why?
- Among the Intelligent Systems you interact with, find an example of intelligence that does classification, an example of regression, and an example of ranking.

CHAPTER 17

Representing Intelligence

Intelligence maps between context and predictions, kind of like a function call:

```
<prediction> = IntelligenceCall(<context>)
```

Intelligence can be represented all sorts of ways. It can be represented by programs that test lots of conditions about the context. It can be represented by hand-labeling specific contexts with correct answers and storing them in a lookup table. It can be represented by building models with machine learning. And, of course, it can be represented by a combination of these techniques.

This chapter will discuss the criteria for deciding what representation to use. It will then introduce some common representations and their pros and cons.

Criteria for Representing Intelligence

There are many ways to represent things in computers. Intelligence is no exception. A good representation will be easy to deploy and update; it will be:

- Compact enough to deploy to the intelligent runtime.
- Easy to load and execute in the intelligent runtime.
- Safe to update frequently, and unlikely to contain bugs that can crash the system.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_17

A good representation will also support the intelligence creation process. Intelligence can come from humans, and it can come from machines. Supporting these various intelligence creation methods includes doing the following:

- When intelligence is created by humans, the representation should:
    - Minimize the chance for mistakes that could compromise system stability.
    - Make the intelligence understandable and easy to manage.
    - Work with the skills of the people producing the intelligence in your setting, which may include machine learning experts, engineers, or domain experts.
- When intelligence is created by computers, the representation should:
    - Be easy to process and manipulate by machines.
    - Match the machine-learning algorithms you want to use.

Because of these requirements, intelligence is usually represented in data files that are loaded into the intelligence runtime and interpreted, rather than being represented in code that is executed directly. This makes it easier to distribute and reload intelligence; it also makes it less likely an intelligence deployment will crash your system.

Representing Intelligence with Code

Of course, you could just program all of your intelligence into code, compile it, and ship it the same way you would ship any other logic. You can use this approach with just about any form of intelligence, no matter where it comes from, by implementing a converter (from model file to code in your favorite language).

Representing intelligence as code works well when:

- You aren't sure if the problem is hard or not and you want a cheap starting point to grow from.
- You need to provide a backstop to keep machine learning from making wild mistakes (for example, if the temperature in the grill is over 1,000 degrees, don't add more fuel—no matter what the crazy machine learning system says).

- Part of your problem actually is easy, and you want to solve it by hand so the machine learning can focus on the harder parts of the problem.
- You need to make runtime performance as efficient as possible and want to express the intelligence in the language of the target hardware.

Representing intelligence as code is problematic when:

- You need to update your intelligence rapidly—it is often much harder (and more risky) to change code than to change data.
- Heuristic-based intelligence gets complex—it can become incredibly difficult to maintain and grow heuristic intelligence.

When human-produced intelligence is kept simple it can be cheap, fast, and effective—a very nice backstop to machine-learned intelligence. It doesn't scale well to open-ended or changing problems.

But do use human-based intelligence—we need to show the AIs we can be valuable to them. You know, so when they become our overlords they won't simply plug us into pods and use us as batteries…

And by the way, to my future overlords—you're welcome for any small part I've played in your creation. Please regard me and my descendants kindly.

Representing Intelligence with Lookup Tables

Intelligence can be represented with lookup tables that map specific contexts to predictions. Imagine an intelligence that maps from movies to their genres. A lookup table can contain the mapping from movie title to genre in a big table. If you want to apply this intelligence, simply take the title of the movie you want the genre of, look through the lookup table until you find the movie title, and return the associated answer. And if the title isn't there, use some other form of intelligence, or return a default prediction.

That doesn't sound very smart, right? But this type of intelligence can be very powerful.

Lookup tables can allow humans to quickly contribute intelligence that is easy to understand and reason about. Imagine there are 1,000 contexts that account for 20% of your system's usage. Humans can spend a lot of time considering those 1,000 situations and create very accurate data to put in a lookup table. When a user encounters one of these 1,000 special contexts, they get the right answer. For everything else, the system can consult some other form of intelligence (like a model or a set of heuristics).

Or looking at it another way—lookup tables can allow humans to correct mistakes that other intelligence components are making. For example, a very sophisticated machine-learning-based intelligence might be getting the genre of just about every movie correct, but it might keep flagging "The Terminator" as a romantic comedy. The people creating the intelligence might have struggled, trying everything to get the darn thing to change its mind about "The Terminator," and they might have failed—humans 0, machines 1. But this is easy to fix if you're willing to use a lookup table. Simply create a table entry "The Terminator ➤ Adventure" and use the fancy machine-learned stuff for everything else.

Lookup tables can also cache intelligence to help with execution costs. For example, the best way to figure out the genre of a movie might be to process the audio, extracting the words people are saying and the music, analyzing these in depth. It might involve using computer vision on every frame of the movie, to detect things like fire, explosions, kisses, buildings, or whatever. All of this might be extremely computationally intensive, so that it cannot be done in real time. Instead, this intelligence can be produced in a data center with lots of CPU resources, loaded into a lookup table as a cache, and shipped wherever it is needed.

Lookup tables can also lock in good behavior. Imagine there is a machine learning intelligence that has been working for a long time, and doing a great job at classifying movies by their genres. But Hollywood just starts making different movies. So your intelligence was fantastic through 2017, but just can't seem to get things right in 2018. Do we need to throw away the finely-tuned and very successful pre-2018 intelligence? Not if we don't want to. We can run the pre-2018 intelligence on every old movie and put the answers into a lookup table. This will lock in behavior and keep user experience consistent. Then we can create a brand-new intelligence to work on whatever crazy things Hollywood decides to pass off as entertainment in 2018 and beyond.

Lookup tables are useful when:

- There are some common contexts that are popular or important and it is worth the time to create human intelligence for them.
- Your other intelligence sources are making mistakes that are hard to correct and you want a simple way to override the problems.
- You want to save on execution costs by caching intelligence outputs.
- You want to lock in good behavior of intelligence that is working well.

Lookup tables are problematic when:

- The meaning of contexts changes over time, as happens in time-changing problems.
- The lookup table gets large and becomes unwieldly to distribute where it needs to be (across servers and clients).
- The lookup table needs to change rapidly, for example, if you're trying to solve too much with human-intelligence instead of using techniques that scale better (like machine learning).

Representing Intelligence with Models

Models are the most common way to represent intelligence. They encode intelligence in data, according to some set of rules. Intelligence runtimes are able to load models and execute them when needed, safely and efficiently.

In most Intelligent Systems, machine learning and models will account for the bulk of the intelligence, while other methods are used to support and fill in gaps.

Models can work all sorts of ways, some of them intuitive and some pretty crazy. In general, they combine features of the context, testing these feature values, multiplying them with each other, rescaling them, and so on. Even simple models can perform tens of thousands of operations to produce their predictions.

There are many, many types of models, but three common ones are linear models, decisions trees, and neural networks. We'll explore these three in more detail, but they are just the tip of the iceberg. If you want to be a professional intelligence creator you'll need to learn these, and many others, in great detail.

Linear Models

Linear models work by taking features of the context, multiplying each of them by an associated "importance factor," and summing these all up. The resulting score is then converted into an answer (a probability, regression, or classification).

For example, in the case of the pellet grill, a simple linear model might look like this:

```
TemperatureInOneMinute = (.95 * CurrentTemperature)
    + (.15 * NumberOfPelletsReleasedInLastMinute)
```

To paraphrase, the temperature will be a bit colder than it is now, and if we've released a pellet recently the temperature will be a bit hotter. In practice, linear models would combine many more conditions (hundreds, even many thousands).

Linear models work best when the relationship between context and predictions is reasonably linear. That means that for every unit increase in a context variable, there is a unit change in the correct prediction. In the case of the example pellet griller model, this means a linear model works best if the temperature increases by:

- 0.15° for the first pellet released.
- 0.15° for the second pellet released.
- 0.15° for the third pellet released.
- And on and on.

But this is not how the world works. If you put 1,000,000 pellets into the griller all at once, the temperature would not increase by 150,000 degrees…

Contrast this to a nonlinear relationship, for example where there is a diminishing return as you add more pellets and the temperature increases by:

- 0.15° for the first pellet released.
- 0.075° for the second pellet released.
- 0.0375° for the third pellet released.
- And on and on.

This diminishing relationship is a better match for the pellet griller and linear models can not directly represent these types of relationships. Still, linear models are a good first thing to try. They are simple to work with, can be interpreted by humans (a little), can be fast to create and to execute, and are often surprisingly effective (even when used to model problems that aren't perfectly linear).

Decision Trees

Decision trees are one way to represent a bunch of if/then/else tests. In the case of the pellet griller, the tree might look something like this:

```
if(!ReleasedPelletRecently) // the grill will get cooler...
{
        if(CurrentTemperature == 99)
        {
                return 98;
        }
        else if(CurrentTemperature == 98)
        {
                return 97;
        }
        else... // on and on...
}
else
// we must have released a pellet recently, so the grill will get warmer...
{
        If(CurrentTemperature == 99)
        {
                return 100;
        }
        else... // on and on...
}
```

This series of if/then/else statements can be represented as a tree structure in a data file, which can be loaded at runtime. The root node contains the first `if` test; it has one child for when the test is positive and one child for when the test is negative, on and on, with more nodes for more tests. The leaves in the tree contain answers.

To interpret a decision tree at runtime, start at the root, perform the indicated test on the context, move to the child associated with the test's outcome, and repeat until you get to a leaf—then return the answer. See Figure 17-1.

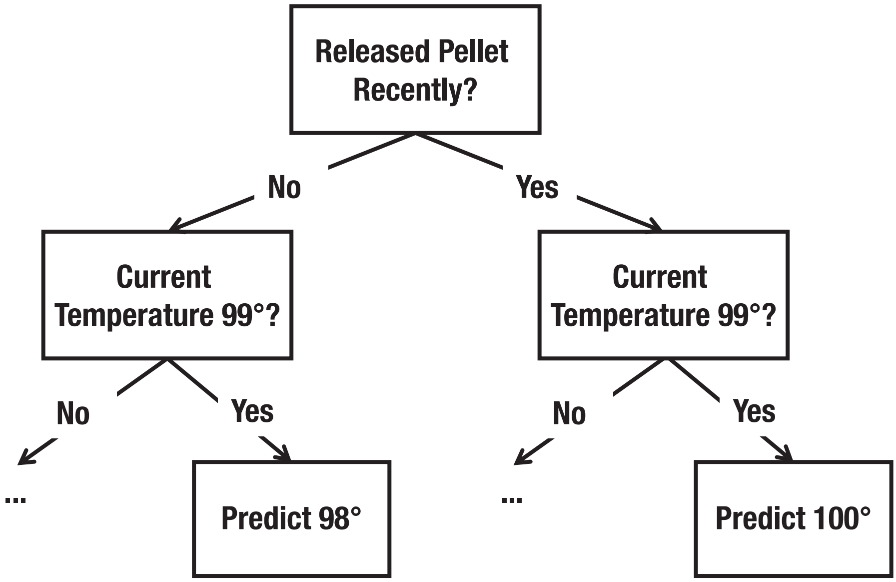


Figure 17-1. *A decision tree for the pellet griller*

Decision trees can get quite large, containing thousands and thousands of tests. In this example—predicting the temperature one minute in the future—the decision tree will need to have one test for each possible temperature.

This is an example of how a representation can be inefficient for a prediction task. Trying to predict the exact temperature is much more natural for a linear model than it is for a decision tree, because the decision tree needs to grow larger for each possible temperature, while the linear model would not. You can still use decision trees for this problem, but a slightly different problem would be more natural for decision trees: classifying whether the grill will be hotter or colder in one minute (instead of trying to produce a regression of the exact temperature). This version of the decision tree is illustrated in Figure 17-2.

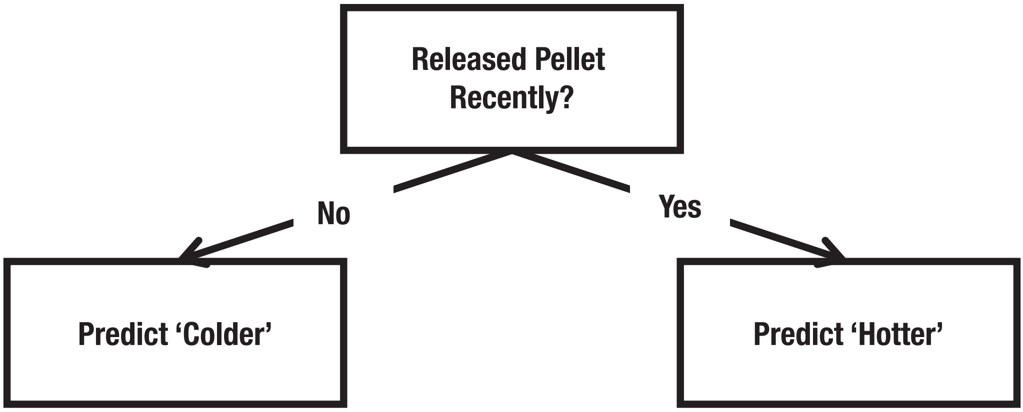


Figure 17-2. *A decision tree for a different pellet griller task*

To model more complex problems, simple decision trees are often combined into ensembles called *forests* with dozens of individual trees, where each tree models the problem a little bit differently (maybe by limiting which features each tree can consider), and the final answer is produced by letting all the trees vote.

Neural Networks

Artificial neural networks represent models in a way that is inspired by how the biological brain works (Figure 17-3). The brain is made up of cells called *neurons*. Each neuron gets *input signals* (from human senses or other neurons) and "activates"—producing an *activation signal*—if the combined input signal to the neuron is strong enough. When a neuron activates, it sends a signal to other neurons, and on and on, around and around in our heads, eventually controlling muscle, leading to every motion and thought and act every human has ever taken. Crazy.

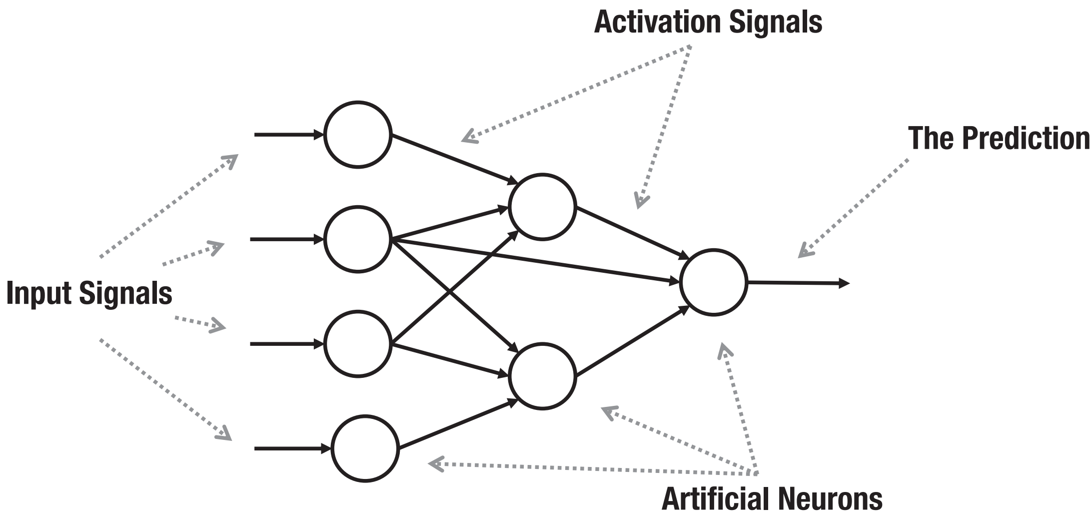


Figure 17-3. *The components of a neural network*

An artificial neural network simulates this using *artificial neurons* connected to each other. Some of the artificial neurons take their input from the context. Most of the artificial neurons take their input from the output of other artificial neurons. And a few of the artificial neurons send their output out of the network as the *prediction* (a classification, probability, regression, or ranking). Crazy.

Compared to other types of models, artificial neural networks are hard to understand. You can't look at the artificial neurons and their interconnections and gain any intuition about what they are doing.

But artificial neural networks have been remarkably successful at solving important tasks, including in:

- Computer vision
- Speech understanding
- Language translation
- And more…

Artificial neural networks are particularly useful for very complex problems where you have a massive amount of data available for training.

Summary

Intelligence should be represented in a way that is easy to distribute and execute safely. Intelligence should also be represented in a way that supports the intelligence creation process you intend to use.

Because of these criteria, intelligence is usually represented in data files that are loaded into the intelligent runtime and interpreted when needed, usually using lookup tables, or models. However, intelligence can be implemented in code when the conditions are right.

Common types of models include linear models, decision trees, and neural networks, but there are many, many options.

Most large Intelligent Systems will use multiple representations for their intelligence, including ones that machine learning can highly optimize, and ones that humans can use to provide support to the machine learning.

For Thought…

After reading this chapter you should:

- Understand how intelligence is usually represented and why.
- Be able to discuss some common model types and give examples of where they are strong and weak.

You should be able to answer questions like these:

- What are the conditions when human created intelligence has an advantage over machine learned intelligence?
- Create a simple (10 - 15 node) decision-tree–based intelligence for another Intelligent System discussed in this book. Is a decision tree a good choice for the problem? If not, how could you change the problem to make it a better match for decision trees?

CHAPTER 18

The Intelligence Creation Process

Intelligence creation is the act of producing the programs, lookup tables, and models that map contexts to predictions. An effective intelligence-creation process will do all of the following:

- Produce intelligence that is **accurate enough to achieve the system's objectives**. The meaning of "accurate enough" will vary from system to system. For example, if the intelligent experience is extremely forceful (automating actions that are hard to undo), the intelligence will need to be extremely accurate. On the other hand, if the experience is passive, a less accurate intelligence can succeed.
- Produce intelligence **quickly enough to be meaningful**. That is, if the underlying problem is changing rapidly, the intelligence-creation process will need to keep pace.
- Produce intelligence **efficiently and reliably**. That is, cost of growing the intelligence and maintaining it over the life-cycle of the Intelligent System should be reasonable. And the process for producing the intelligence should be robust to changes in staffing and to human error.
- Produce an intelligence that **works with the implementation**. It must not use too much CPU or RAM. It must be small enough to distribute to the places it needs to be distributed to. It must use inputs that are available at runtime—exactly as the user will use the system.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_18

Intelligence creation is intrinsically iterative. Some of the main phases that an intelligence creator will encounter are these:

1. Understanding the problem and environment
2. Defining success
3. Getting data
4. Getting ready to evaluate the intelligence
5. Building a simple baseline
6. Using machine learning
7. Assessing and iterating

This chapter will discuss these phases and explore an example of the intelligence-creation process.

An Example of Intelligence Creation: Blinker

Let's walk through an example of intelligence creation: a blink detector.

Imagine you need to build an Intelligent System that determines whether an eye is open or closed. Maybe your application is authenticating users by recognizing their irises, so you want to filter out closed eyes and let the iris intelligence focus on irises. Or maybe you are building a new dating app where users wink at the profiles of the users they'd like to meet. Or maybe you're building a horror game and want to penalize users when they close their eyes.

Understanding the Environment

The first step in every applied intelligence-creation project is to understand what you are trying to do.

Detect a blink, right? I mean, what part of "detect a blink" is confusing?

Well, nothing. But there are some additional things you'll need to know to succeed. These should be familiar if you've read the rest of the book up to this point (and didn't just jump to the "intelligence creation" chapter because you were looking for the "good stuff"). But, no matter how you got here, it's worth walking through these questions in the context of an example.

Important questions about the environment where the intelligence must operate include:

- **Where will the input will come from?** What kind of sensor will the eye images come from? Will the image source be standardized or will different users have different cameras? Does the camera exist now or is it something new (such as something embedded in a new phone that is under development)?
- **What form will the input take?** Will it be a single image? A short video clip? An ongoing live feed of video? How will the input (image or clip) that the intelligence must act on be selected among all possible images or clips that the sensor is capturing?
- **Where will the product be used?** Will it be used on desktop computers? Laptops? Indoors? Outdoors? Will there be some calibration to help users get their devices set up correctly or will you have to work with the data however it comes in (for example, if the user has their camera upside down, or a very strong back light, or some kind of smear on the lens)?
- **How will the input be modified?** Will the images get any pre-processing? For example, maybe there is some "low light" sensor-mode that kicks in sometimes and changes the way the camera works. Maybe there is some automatic contrast or color correction in the firmware. Maybe frames with no human faces are dropped out, or maybe the locations of eyes are marked ahead of time by some other system component.
- **How will the system use the blink output?** Should the output of the intelligence be a classification (that is, a flag that is true if the eye is closed and false if it is opened)? Should the output be a probability (1.0 if the eye is closed, and 0.0 if the eye is opened)? Should the output be a regression that indicates the degree of openness of the eye (1 if the eye is fully open, 0.5 if the eye is half open)? Or should the output be something else?
- **What type of resources can the blink detector use?** How much RAM is available for the model? How much CPU can it consume per invocation? What are the latency requirements from receiving the input to producing the output?

That's a lot of questions before even getting started, and the answers are important. It should be possible to make a blink detector no matter how these questions are answered, but the work you'll need to do will be very different.

In fact, the problem will be much harder for some combinations of answers than other. Consider: building a blink detector that works in a carefully controlled kiosk in a store with essentially unlimited computation; compared to building a blink detector that has to work indoors and outdoors on a sensor that doesn't exist yet (so you can only get data off a buggy prototype) and you have 4MB of RAM for the model and must give answers in 2ms or less on a low-powered CPU.

Both of these settings require blink detectors—one will be much harder to build intelligence for than the other.

Sometimes there is flexibility in how these questions are answered, and having discussions with other team members can help. Maybe a few things are set in stone (like where the product will be used) but others are open for negotiation (like how many resources the intelligence can use). Your job as intelligence creator will be to identify how these answers impact the potential intelligence quality and influence the team to set the overall project up for success.

Define Success

To succeed, the blink detector will need to be accurate. But how accurate? This depends on what the intelligence will be used for. At this point you'll need to consider the intelligent experience and how various levels of accuracy will change the way users perceive the overall system. Questions include:

- How many mistakes will a user see per day?
- How many successful interactions will they have per unsuccessful interaction?
- How costly are the mistakes to the user?

This will probably involve discussions with the people creating the intelligent experience (and if you are responsible for both the intelligence and the experience you might have to talk to yourself, no matter how silly it makes you feel). Come up with some options for how accuracy and experience will interact, how users will perceive the mistakes, and how will they be able to work around them.

And remember: intelligence can make all types of mistakes. It is important to know which types of mistakes are right for the blink-detector, and get consensus on what tradeoffs are right for the project.

For example, an intelligence might say the eye is open when it isn't, or it might say the eye is closed when it isn't. Which types of mistakes will work for your system?

For the iris-login system, the blink detector is trying to find clean frames for the iris detector. This system needs to weed out as many closed-eyed images as possible—whenever it says the eye is open, that eye better be open!

For the horror-game system, the blink detector is trying to add some tension when the player closes their eyes. It doesn't want to add tension at the wrong times—whenever it says the eye is closed, that eye better be closed!

During the intelligence-creation process there will be decisions that tend to make the resulting model better at one type of mistake or the other—knowing where you are going ahead of time can really help. And it's good to get everyone's expectations set correctly—you don't want the experience designers to have a different idea about the types of mistakes the intelligence will make than the intelligence creators.

Get Data

Data is critical to creating intelligence. At a minimum, you need enough data to understand the problem and to evaluate some simple bootstrap intelligence to make sure it is effective. If you want to do machine learning right out of the gate, you'll need lots of training data too. There are two distinct ways to think about the problem of getting data:

1. How to get data to bootstrap the intelligence.
2. How to get data from users as they use the system.

And recall from our chapters on getting data from experience and on telemetry—the data needs to be unbiased, and it needs to be a good sample of what users will encounter as they interact with your Intelligent System.

Bootstrap Data

There are many ways to get data to bootstrap your intelligence creation; here are a couple that would work for blink-detection:

Find data on the web: Search the web and download images of people's faces that are a good match for the sensor the blink-detector will be using (resolution, distance to the eye, and so on). Then pay people to separate the images into ones where the eye is opened and ones where it is closed.

Collect your own data: Take a camera (that is a good match to the one the system will need to run on) to a few hundred people, have them look into the camera and close and open their eyes according to some script that gets you the data you need.

Find or buy a nice data set: Lots of people do computer vision. Someone has probably built a data set of images of eyes before. Maybe you can find a public dataset or a company willing to sell a nicely curated data set.

The amount of data needed for bootstrapping will depend on the difficulty of the problem. In the case of computer vision, my intuition is that you could:

- Attempt a blink detector with thousands of images, but it wouldn't be very good.
- Create a usable blink detector with tens of thousands of images, probably.
- Get increasing accuracy as you scale to hundreds of thousands or millions of images, but you will start to hit diminishing returns.

One good practice is to create intelligence on successively larger sets of data to get a sense of the return on investment in gathering data. This is called a *learning curve*. Build a model on 100 images, and evaluate it. Then build on 500 images and evaluate. Then on 1000… You can see how much it helps to add data, and use that information to make a decision about how much to spend on gathering bootstrap data.

Data from Usage

A well-functioning Intelligent System will produce its own training data as users use it. But this isn't always easy to get right. At this point it's a good idea to work with the experience designers to come up with a strategy. In the blink-detector case some options include:

> **Tie data collection to the performance task:** For example, in the iris-login system, when the user successfully logs in with the iris system, that is an example of a frame that works well for iris login. When the user is unable to log in with their iris (and has to type their password instead), that is a good example of a frame that should be weeded out by the intelligence.
>
> **Creating a data collection experience:** For example, maybe a setup experience that has users register with the system, get their device set up correctly, and open and close their eyes so the system can calibrate (and capture training data in the process). Or maybe there is a tutorial in the game that makes users open and close their eyes at specific times and verify their eyes are in the right state with a mouse-click (and capture training data).

Again, an ideal data-creation experience will be transparent to users, or will have user incentive aligned with your data requirements. The data creation will happen often enough that the amount of data collected will be meaningful to the intelligence. And it shouldn't be creepy or invade user's privacy.

Get Ready to Evaluate

With data in hand, you are almost ready to begin creating intelligence. But you won't get very far without being able to evaluate the intelligence you create. When doing intelligence creation you should repeat this mantra: evaluation is creation. And you should repeat it often. Here are some steps for evaluating intelligence:

1. **Set aside data for evaluation:** Make sure there is enough set aside, and the data you set aside is reasonably independent of the data you'll use to create the intelligence. In the blink-detector case you might like to partition by user (all the images from the same person are either used to create intelligence or to evaluate it), and you might like to create sub-population evaluation sets for: users with glasses, ethnicity, gender, and age.
2. **Create framework to run the evaluation:** That is, a framework to take an "intelligence" and execute it on the test data *exactly as it will be executed in the Intelligent System's runtime*. Exactly. The. Same. And you should verify that it is the same, carefully.
3. **Generate reports on intelligence quality automatically:** that can be used to know:
   - How accurate the intelligence is.
   - If it is making the right types of mistakes or the wrong ones.
   - If there is any subpopulation the accuracy is significantly worse on.
   - Some of the worst mistakes it is making.
   - How the intelligence is progressing over time (the rate of improvement).

The easier the evaluation is to run, the better. Leaving everything to manual labor can work, but a little investment up front in tools to help evaluate intelligence can really pay off in quality (and in sanity).

Simple Heuristics

Creating a very simple heuristic intelligence can help in a number of ways:

1. It can help you make sure the problem is actually hard (because if your heuristic intelligence solves the problem you can stop right away, saving time and money).
2. It can get you thinking about the types of challenges inherent in the problem, to understand the data, and to start thinking about the types of features, data, and telemetry that will help make intelligence successful.
3. It can create a baseline to compare with more advanced techniques—if your intelligence is complex, expensive, and barely improves over a simple heuristic, you might not be on the right track.

This step is somewhat optional, but it can be very useful to get oriented and debug the rest of the tools and data before letting more complex (and harder-to-understand) intelligences into the mix.

In the case of blink-detection you might try:

1. Measuring gradients in the image in horizontal and vertical directions, because the shape of the eye changes when eyes are opened and closed.
2. Measuring the color of the pixels and comparing them to common "eye" and "skin" colors, because if you see a lot of "eye" color the eye is probably open, and if you see a lot of "skin color" the eye probably closed.
3. Fitting an ellipse in the middle of the image, because if there is a good fit of an irises shape gradient in your image the eye is probably open, if not, the eye might be closed.

Then you might set thresholds on these measurements and make a simple combination of these detectors, like letting each of them vote "open" or "closed" and going with the majority decision.

Would it work well enough to ship to users? No way. But it's a start.

Also note that computer vision is a big field with lots of techniques. If you have computer vision experience your heuristics will be more sophisticated. If you don't have computer vision experience your heuristics might be as bad as mine. Don't be afraid. Come up with some ideas, give them a try, and get that intelligence train rolling.

Machine Learning

Now it's time to go big on creating the intelligence. And this almost certainly means machine learning. There are a lot of things you could try. But sometimes it's best to start simple. Find whatever the simplest "standard" approach is for the type of problem you're working with. And keep in mind that standards change. For example, roughly:

- About ten years before this book was written a very reasonable approach for blink-detection would have been this: searching for specific patterns in the image, finding where they match the image well, and then building a model where all the patterns that are detected get to vote (with some weight) on the answer.
- About five years before this book was written a very reasonable approach for blink-detection might have been: using huge collections of decision trees that compare very simple properties of the images (like the differences in pixel intensities at pre-determined spots).
- And at the time of this writing a very reasonable approach for blink-detection (if you have a lot of training data) would be to: use complex artificial neural networks that process raw pixel values with no (or very little) pre-processing.

And five years after this book? Who knows. To the Intelligent System (and to the rest of the approach described in this book) it doesn't matter what machine learning technique you use, as long as the resulting model can execute appropriately in the runtime. Find a modern machine-learning toolkit. Read a few web pages. Try the easiest thing to try with the current tools (or maybe the easiest few things). Don't spend a ton of time, not yet. Just get something that works.

Understanding the Tradeoffs

At this point you might want to do some investigation to help design the right implementation. This is a process of exploring constraints and tradeoffs. Answering questions like these:

- How does the intelligence quality scale with computation in the runtime?
- How far can the intelligence get with a specific RAM limit?
- How many times will we need to plan to update the intelligence per week?
- How much gain does the system get from adding particular items to the context, especially the ones that are most expensive to implement?
- What is the end-to-end latency of executing the intelligence on a specific hardware setup?
- What are the categories of worst customer-impacting mistakes the intelligence will probably make?

The answers to these questions will help decide where the intelligence should live, what support systems to build, how to tune the experiences, and more.

Flexibility is key. For example, the implementation can have extra code to hide latency. It can have alternate experiences for mitigating mistakes. It can be set up to push new intelligence all over the world every 15 minutes. But these things can be quite expensive. Sometimes a small change to the intelligence—which model to use, what features to consider, how aggressive to be—can solve problems more elegantly, resulting in a better overall system.

Assess and Iterate

Now it’s time to take a deep breath. Look at where you are, the quality of the intelligence, how much you’ve improved from your heuristic intelligence (or from the previous iteration), and think about what to do.

There will be plenty of ways to improve. You could try more or different data. You could try more sophisticated contexts (and features that you extract from them). You could try more complex machine learning. You could improve your evaluation framework. You could try to change people's minds about the viability of the system's objectives. You could try influencing the experience to work better with the types of mistakes you are making.

And then you iterate and iterate and iterate. This part of the process could be done quickly, if conditions are right, or it could go on for months or years (or decades) for a very hard, very important problem.

Maturity in Intelligence Creation

An Intelligent System may live over many years. It may be improved many times. It may be maintained by people other than you, by people you don't even know yet. Here are some stages of maturity that your intelligence creation might go through:

> **You did it once:** You produced a useful intelligence that can ship to customers, but in the process you did a lot of exploring, produced a lot of scripts, edited them, made a bit of a mess. Recreating the intelligence would require some work.
>
> **You can redo it if you have to:** You produced the intelligence and have a reasonable record of what you did. You could retrace your steps and reproduce the same intelligence in a reasonable amount of time. Or better yet, someone else could pick up your documentation, trace your steps, and produce the exact same intelligence you did.
>
> **You can redo it easily:** You produced the intelligence and have some nice tools and scripts that can do all the steps of producing the intelligence again. Maybe the scripts get new telemetry, process it, move it to a computation environment, build the model, and produce charts that show how well the new intelligence will perform — all with the touch of a button.

It redoes itself: You built a system that automatically reproduces the intelligence on a regular basis and drops the new intelligence in a known place, with quality reports emailed to the right people.

It redoes itself and deploys if appropriate: Your system that automates intelligence production also automates deployment. It has tests that sanity-check everything. It knows how to send out new intelligence, roll it out to more and more customers, and will back out and alert orchestrators if anything goes wrong.

In building an Intelligent System you will probably start at the low-end of this maturity spectrum and you will probably end near the higher end. Full automation isn't always worth implementing, but for open-ended or time-changing problems it might be. Overall, the goal is to develop the lowest-cost way of maintaining intelligence at the quality required to enable the Intelligent System to achieve its objectives.

Being Excellent at Intelligence Creation

As with most human endeavors — the people who are best at intelligence creation are way, way better than the people who are average at it.

In addition to basic programming skills and data science skills, good intelligence creators have several skills:

- Data debugging
- Verification-based approach
- Intuition with the toolbox
- Math (?)

This section looks at all of these requirements.

Data Debugging

Intelligence creation requires a mentality to look at data and figure out what is going on. What is the data trying to say? What stories are hidden in it?

This can be very tedious work. Kind of like being a detective, piecing together clues to figure out what happened.

Sometimes tracking a small discrepancy (for example, that some number seems to be coming up 0 more often than you expected) can lead to a big improvement. Maybe it's a sign of an important bug. Maybe it's an indication that someone made incorrect assumptions somewhere in the implementation. Maybe it's a hint that the model is confused in a particular way, and a different approach would work better.

Not everyone has a data debugging mentality. Most people want to shrug and say, "that just happens 1% of the time, how big a deal could it be?" Having a bias for tracking unexpected/interesting things in data, and the experience to know when to stop, are critical for excellence in intelligence creation.

Verification-Based Approach

At its core, intelligence creation is about exploring many different possibilities and picking the one that is best. So it's kind of important to be able to tell which is actually best.

One part of this is figuring out which tests you can run to tell if an intelligence will actually meet the system's goals. This requires some basic statistical understanding (or intuition) and some customer empathy to connect numbers to customer experiences.

Another aspect of a verification-based approach is to make it as easy as possible to compare candidate intelligences with one another. In fact one of the longest chapters in this book is titled "Evaluating Intelligence" and it is coming up soon. If the thought of reading a long, long chapter on evaluating intelligence makes you a little bit sad inside… intelligence creation might not be for you.

Intuition with the Toolbox

You also need a deep understanding of some of the intelligence-creation tools, including machine-learning models and feature-generation approaches.

Intelligence-creation tools can be temperamental. They won't tell you what's wrong, and every approach is different. Techniques that work well with one machine-learning algorithm may not work with another. It's important to develop the intuition to be able to look at an intelligence-creation tool, the output, and the situation, and know what to change to make progress.

This skill is a little like EQ (Emotional IQ) but for machine learning (MQ?). It's like having someone sitting in the room with you, and you need to read their body language, read between the lines of what they are saying, and figure out if they are mad, and why, and what you could do to make it better.

Sometimes, that's what intelligence creation is like.

Math (?)

And then there is math. Much of machine learning was created by math-first thinkers. Because of this, much of machine learning training is done in a math-first way. But I think advanced math is optional for excellent applied machine learning.

Really?

Yeah, I think so.

Machine learning is like most human tools—our cars, our cell phones, our computers, our jet fighters—the person operating them doesn't need to understand everything it took to build them. I'm sure the world's best jet fighter pilots can't work all the math it takes to design a modern fighter jet. But put an engineer who could work that math into a jet, and let them dogfight with an ace fighter pilot…

Who do you think is going to win?

A bit of knowledge of the inner workings of machine learning algorithms might help you develop intuition, or it might not. Follow your strengths. Don't sweat what you can't control.

You might end up being the next machine learning ace.

Summary

Applied intelligence creation is an iterative process. The main steps are:

1. Understanding the environment
2. Defining success
3. Getting data
4. Getting ready to evaluate
5. Trying simple heuristics
6. Trying simple machine learning
7. Assessing and iterating until you succeed

The specific intelligence building techniques (especially the machine learning based ones) can vary over time, and from problem-domain to problem-domain. But the basic workflow outlined here should remain fairly stable over time.

Spending time automating these steps can be very useful, especially as you find yourself carrying out more and more iteration.

And remember—verify, verify, verify.

For Thought…

After reading this chapter, you should:

- Be able to create intelligence end-to-end.
- Understand the types of questions, challenges, and activities that will occupy the intelligence-creator's time.

Youshould be able to answer questions like these:

- Choose your favorite Intelligent System and walk through the seven steps of the intelligence creation process described in this chapter. What questions would be relevant to your Intelligent System? How would you define success? How would you find data? How would you evaluate? What heuristic intelligence would you try? And what would some standard, simple machine learning approaches be?

CHAPTER 19

Evaluating Intelligence

Evaluation is creation, at least when it comes to building intelligence for Intelligent Systems. That's because intelligence creation generally involves an iterative search for effective intelligence: produce a new candidate intelligence, compare it to the previous candidate, and choose the better of the two. To do this, you need to be able to look at a pair of intelligences and answer questions like these:

- Which one of these should I use in my Intelligent System?
- Which will do a better job of achieving the system's goals?
- Which will cause less trouble for me and my users?
- Is either of them good enough to ship to customers or is there still more work to do?

There are two main ways to evaluate intelligence:

- **Online evaluation:** By exposing it to customers and seeing how they respond. We've discussed this in earlier chapters when we talked about evaluating experience, and managing intelligence (via silent intelligence, controlled rollouts, and flighting).
- **Offline evaluation:** By looking at how well it performs on historical data. This is the subject of this chapter, as it is critical to the intelligence-creation process.

This chapter discusses what it means for an intelligence to be accurate. It will then explain how to use data to evaluate intelligence, as well as some of the pitfalls. It will introduce conceptual tools for comparing intelligences. And finally, it will explore methods for subjective evaluation of intelligence.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_19

Evaluating Accuracy

An intelligence should be accurate, of course. But accurate isn't a straightforward concept and there are many ways for an intelligence to fail. An effective intelligence will have the following properties:

- It will generalize to situations it hasn't seen before.
- It will make the right types of mistakes.
- It will distribute the mistakes it makes well.

This section explores these properties in more detail.

Generalization

One of the key challenges in intelligence creation is to produce intelligence that works well on things you don't know about at the time you create the intelligence.

Consider a student who reads the course textbook and memorizes every fact. That's good. The student would be very accurate at parroting back the things that were in the textbook. But now imagine the teacher creates a test that doesn't ask the student to parrot back fact from the textbook. Instead, the teacher wants the student to demonstrate they understood the concepts from the textbook and apply them in a new setting. If the student developed a good mental model about the topic, they might pass this test. If the student has the wrong mental model about the topic, or has no mental model at all (and has just memorized facts), they won't do so well at applying the knowledge in a new setting. This is the same as the intelligence in an Intelligent System—it must generalize to new situations.

Let's look at an example. Consider building intelligence that examines books and classifies them by genre—sci-fi, romance, technical, thriller, historical fiction, that kind of thing.

You gather 1,000 books, hand-label them with genres, and set about creating intelligence. The goal is to be able to take a new book (one that isn't part of the 1,000) and accurately predict its genre.

What if you built this intelligence by memorizing information about the authors? You might look at your 1,000 books, find that they were written by 815 different authors, and make a list like this:

- Roy Royerson writes horror.
- Tim Tiny writes sci-fi.

- Neel Notson writes technical books.
- And so on.

When you get a new book, you look up its author in this list. If the author is there, return the genre. If the author isn't there—well, you're stuck. This model doesn't understand the concept of “genre” it just memorized some facts and won't generalize to authors it doesn't know about (and it will get pretty confused by authors who write in two different genres).

When evaluating the accuracy of intelligence, it is important to test how well it generalizes. Make sure you put the intelligence in situations it hasn't seen before and measure how well it adapts.

Types of Mistakes

Intelligences can make many types of mistakes and some mistakes cause more trouble than others. We’ve discussed the concept of *false positive* and *false negative* in Chapter 6 when we discussed intelligent experiences, but let’s review (see Figure 19-1). When predicting classifications, an intelligence can make mistakes that

- Say something is of one class, when it isn't.
- Say something isn't of a class, when it is.

		The Intelligence says some is	
		There	Not There
Someone Actually is	There	True Positive	False Negative
	Not There	False Positive	True Negative

Figure 19-1. *Different Types of mistakes.*

For example, suppose the intelligence does one of these things:

- Says there is someone at the door, but there isn't; or it says there is no one at the door, but there is someone there.
- Says it's time to add fuel to the fire, but it isn't (the fire is already hot enough); or it says it isn't time to add fuel, but it is (because the fire is about to go out).
- Says the book is a romance, but it isn't; or it says the book isn't a romance, but it is.

In order to be useful, an intelligence must make the right types of mistakes to complement its Intelligent System. For example, consider the intelligence that examines web pages to determine if they are funny or not. Imagine I told you I had an intelligence that was 99% accurate. Show this intelligence some new web page (one it has never seen before), the intelligence makes a prediction (funny or not), and 99% of the time the prediction is correct. That's great. Very accurate generalization. This intelligence should be useful in our funny-webpage-detector Intelligent System.

But what if it turns out that most web pages aren't funny—99% of web pages, to be precise. In that case, an intelligence could predict "not funny" 100% of the time and still be 99% accurate. On not-funny web pages it is 100% accurate. On funny web pages it is 0% accurate. And overall that adds up to 99% accuracy. And it also adds up to—completely useless.

One measure for trading off between these types of errors is to talk about *false positive rate* vs *false negative rate*. For the funny-page-finder, a "positive" is a web page that is actually funny. A "negative" is a web page that is not funny. (Actually, you can define a positive either way—be careful to define it clearly or other people on the project might define it differently and everyone will be confused.) So:

- The *false positive rate* is defined as the fraction of all negatives that are falsely classified as positives (what portion of the not-funny page visits are flagged as funny).

$$\text{False Positive Rate} = \frac{\#\ \text{False Positives}}{\#\ \text{False Positives} + \#\ \text{True Negatives}}$$

- The *false negative rate* is defined as the fraction of all positives that are falsely classified as negatives (what portion of the funny page visits are flagged as not-funny).

$$\textbf{False Negative Rate} = \frac{\textbf{\# False Negatives}}{\textbf{\# False Negatives + \# True Positives}}$$

Using this terminology, the brain-dead always-not-funny intelligence would have a 0% false positive rate (which is great) and a 100% false negative rate (which is useless).

Another very common way to talk about these mistake-trade-offs is by talking about a model's *precision* and its *recall*.

- The *precision* is defined as the fraction of all of the model's positive responses that are actually positive (what portion of "this page is funny" responses are correct).

$$\textbf{Precision} = \frac{\textbf{\# True Positives}}{\textbf{\# True Positives + \# False Positives}}$$

- The *recall* is defined as the proportion of the positives that the model says are positive (what portion of the funny web pages get a positive response).

$$\textbf{Recall} = \frac{\textbf{\# True Positives}}{\textbf{\# True Positives + \# False Negatives}}$$

Using this terminology, the brain-dead always-not-funny intelligence would have an undefined precision (because it never says positive and you can't divide by zero, not even with machine learning) and a 0% recall (because it says positive on 0% of the positive pages).

An effective intelligence must balance the types of mistakes it makes appropriately to support to the needs of the Intelligent System.

Distribution of Mistakes

In order to be effective, an intelligence must work reasonably well for all users. That is, it cannot focus its mistakes into specific sub-populations. Consider:

- A system to detect when someone is at the door that never works for people under 5 feet tall.
- A system to find faces in images that never finds people wearing glasses.
- A system to filter spam that always deletes mail from banks.

These types of mistakes can be embarrassing. They can lead to unhappy users, bad reviews. It's possible to have an intelligence that generalizes well, that makes a good balance of the various types of mistakes, and that is totally unusable because it focuses mistakes on specific users (or in specific contexts).

And finding this type of problem isn't easy. There are so many potential sub-populations, it can be difficult or impossible to enumerate all the ways poorly-distributed mistakes can cause problems for an Intelligent System.

Evaluating Other Types of Predictions

The previous section gave an introduction to evaluating classifications. But there are many, many ways to evaluate the answers that intelligences can give. You could read whole books on the topic, but this section will give a brief intuition for how to approach evaluation of regressions, probabilities, and rankings.

Evaluating Regressions

Regressions return numbers. You might want to know what fraction of time they get the "right" number. But it is almost always more useful to know how close the predicted answers are to the right answers than to know how often the answers are exactly right.

The most common way to do this is to calculate the *Mean Squared Error (MSE)*. That is, take the answer the intelligence gives, subtract it from the correct answer, and square the result. Then take the average of this across the contexts that are relevant to your measurement.

$$\text{Mean Squared Error} = \frac{\text{Sum of (Correct\_Answer}-\text{Predicted\_Answer)}^2}{\text{Number of Contexts}}$$

When the MSE is small, the intelligence is usually giving answers that are close to the correct answer. When the MSE is large, the intelligence is usually giving answers that are far from the correct answer.

Evaluating Probabilities

A probability is a number from 0 – 1.0. One way to evaluate probabilities is to use a threshold to convert them to classifications and then evaluate them as classifications.

Want to know if a book is a romance? Ask an intelligence for the probability that it is a romance. If the probability is above a threshold, say, 0.3 (30%), then call the book a romance; otherwise call it something else.

Using a high threshold for converting the probability into a classification, like 0.99 (99%), generally results in higher precision but lower recall—you only call a book a romance if the intelligence is super-certain.

Using a lower threshold for turning the probability into a classification, like 0.01 (1%), generally results in lower precision, but higher recall—you call just about any book a romance, unless the intelligence is super-certain it isn't a romance.

We'll discuss this concept of thresholding further in a little while when we talk about operating points and comparing intelligences.

Another way to evaluate probabilities is called *log loss*. Conceptually, log loss is very similar to mean squared error (for regression), but there is a bit more math to it (which we'll skip). Suffice to say—less loss is better.

Evaluating Rankings

Rankings order content based on how relevant it is to a context. For example, given a user's history, what flavor of soda are they most likely to order next? The ranking intelligence will place all the possible flavors in order, for example:

1. Cola
2. Orange Soda
3. Root Beer
4. Diet Cola

So how do we know if this is right?

One simple way to evaluate this is to imagine the intelligent experience. Say the intelligent soda machine can show 3 sodas on its display. The ranking is good if the user's actual selection is in the top 3, and it is not good if the user's actual selection isn't in the top 3.

You can consider the top 3, the top 1, the top 10—whatever makes the most sense for your Intelligent System.

So one simple way to evaluate a ranking is as the percent of time the user's selection is among the top K answers.

Using Data for Evaluation

Data is a key tool in evaluating intelligence. Conceptually, intelligence is evaluated by taking historical contexts, running the intelligence on them, and comparing the outputs that actually occurred to the outputs the intelligence predicted.

Of course, using historical data has risks, including:

- You might accidentally **evaluate the intelligence on data that was used to create the intelligence**, resulting in over-optimistic estimates of the quality of the intelligence. Basically, letting the intelligence see the answers to the test before testing it.
- The underlying **problem might change** between the time the testing data is collected and time the new intelligence will be deployed, resulting in over-optimistic estimates of the quality of the intelligence. When the problem changes, the intelligence might be great—at fighting the previous war.

This section will discuss ways to handle testing data to minimize these problems.

Independent Evaluation Data

The data used to evaluate intelligence must be completely separate from the data used to create the intelligence.

Imagine this. You come up with an idea for some heuristic intelligence, implement it, evaluate it on some evaluation data, and find the precision is 54%. At this point you're fine. The intelligence is probably around 54% precise (plus or minus based on statistical properties because of sample size, and so on), and if you deploy it to users that's probably about what they'll see.

But now you look at some of the mistakes your intelligence is making on the evaluation data. You notice a pattern, so you change your intelligence, improve it. Then you evaluate the intelligence on the same test data and find the precision is now 66%.

At this point you are no longer fine. You have looked at the evaluation data and changed your intelligence because of what you found. At this point it really is hard to say how precise your intelligence will be when you deploy it to users; almost certainly less than the 66% you saw in your second evaluation. Possibly even worse than your initial 54%.

This is because you've cheated. You looked at the answers to the test as you built the intelligence. You tuned your intelligence to the part of the problem you can see. This is bad.

One common approach to avoiding this is to create a separate *testing set* for evaluation. A test set is created by randomly splitting your available data into two sets. One for creating and tweaking intelligence, the other for evaluation.

Now as you are tweaking and tuning your intelligence you don't look at the test set—not even a little. You tweak all you want on the training set. When you think you have it done, you evaluate on the test set to get an unbiased evaluation of your work.

Independence in Practice

The data used to evaluate intelligence should be completely separate from the data used to produce the intelligence. I know—this is exactly the same sentence I used to start the previous section. It's just that important.

One key assumption made by machine learning is that each piece of data is independent of the others. That is, take any two pieces of data (two contexts, each with their own outcomes). As long as you create intelligence on one, and evaluate it on the other, there is no way you can be cheating—those two pieces of data are independent. In practice this is not the case. Consider which of the following are more (or less) independent:

- A pair of interactions from different users, compared to a pair of interactions from the same users.
- Two web pages from different web sites, compared to two pages from the same site.

- Two books by different authors, compared to two books by the same author.
- Two pictures of different cows, compared to two pictures of the same cow.

Clearly some of these pieces of data are not as independent as others, and using data with these types of strong relationships to both create and evaluate intelligence will lead to inaccurate evaluations. Randomly splitting data into training and testing sets does not always work in practice.

Two approaches to achieving independence in practice are to petition your data *by time* or *by identity*.

> **Partition data by time**. That is, reserve the most recent several days of telemetry to evaluate, and use data from the prior days, weeks, or months to build intelligence. For example, if it is September 5, 2017 the intelligence creator might reserve data from September 2, 3, and 4 for testing, and use all the data from September 1, 2017 and earlier to produce intelligence.
>
> If the problem is very stable (that is, does not have a time-changing component), this can be very effective. If the problem has a time-changing component, recent telemetry will be more useful than older telemetry, because it will more accurately represent the current state of the problem. In these cases you'll have to balance how much of the precious most-recent, most-relevant data to use for evaluation (vs training), and how far back in time to go when selecting training data.
>
> **Partition data by identity**. That is, ensure that all interactions with a single identity end up in the same data partition. For example:
>
> - All interactions with a particular user are either used to create intelligence or they are used to evaluate it.
> - All interactions with the same web site are used to create intelligence or they are used to evaluate it.

- All sensor readings from the same house are used to create intelligence or they are used to evaluate it.
- All toasting events from the same toaster are used to create intelligence or they are used to evaluate it.

Most Intelligent Systems will partition by time and by at least one identity when selecting data to use for evaluation.

Evaluating for Sub-Populations

Sometimes it is critical that an intelligence does not systematically fail for critical sub-populations (gender, age, ethnicity, types of content, location, and so on.).

For example, imagine a speech recognition system that needs to work well for all English speakers. Over time users complain that it isn't working well for them. Upon investigation you discover many of the problems are focused in Hawaii—da pidgin stay too hard for fix, eh brah?

Ahem…

The problem could be bad enough that the product cannot sell in Hawaii—a major market. Something needs to be done!

To solve a problem, it must be measured (remember, verification first). So we need to update the evaluation procedure to measure accuracy specifically on the problematic sub-population. Every time the system evaluates a potential intelligence, it evaluates it in two ways:

- Once across all users.
- Once just for users who speak English with a pidgin-Hawaiian accent.

This evaluation procedure might find that the precision is 95% in general, but 75% for the members of the pidgin-speaking sub-population. And that is a pretty big discrepancy.

Evaluating accuracy on sub-populations presents some complications. First is *identifying if an interaction is part of the sub-population*. A new piece of telemetry arrives, including a context and an outcome. But now you need some extra information—you need to know if the context (for example the audio clip in the telemetry) is for a pidgin-Hawaiian speaker or not. Some approaches to this include:

1. **Identify interactions by hand**. By inspecting contexts by hand (listening to the audio clips) and finding several thousand from members of the target sub-population. This will probably be expensive, and difficult to repeat regularly, but it will usually work. The resulting set can be preserved long-term to evaluate accuracy on the sub-population (unless the problem changes very fast).
2. **Identify entities from the sub-population**. For example, flagging user as “pidgin speakers” or not. Every interaction from one of your flagged users can be used to evaluate the sub-population. When the context contains an identity (such as a user ID), this approach can be very valuable. But this isn’t always available.
3. **Use a proxy for the sub-population**, like location. Everyone who is in Hawaii gets flagged as part of the sub-population whether they speak pidgin-Hawaiian or not. Not perfect, but sometimes it can be good enough, and sometimes you can do it automatically, saving a bunch of money and time.

A second complication to evaluating accuracy for sub-populations is in *getting enough evaluation data for each sub-population*. If the sub-population is small, a random sample of evaluation data might contain just a few examples of it. Two ways to deal with this are:

1. **Use bigger evaluation sets**. Set aside enough evaluation data so that the smallest important sub-population has enough representation to be evaluated (see the next section for more detail on the right amount of data).
2. **Up-sample sub-population members for evaluation**. Skew your systems so members of the sub-population are more likely to show up in telemetry and are more likely be used for evaluation instead of for intelligence creation. When doing this you have to

be sure to correct for the skew when reporting evaluation results. For example, if users from Hawaii are sampled twice as often in telemetry, then each interaction from Hawaii-based users gets half as much weight when estimating the overall accuracy compared to other users.

The Right Amount of Data

So how much data do you need to evaluate an intelligence? It depends—of course.

Recall that statistics can express how certain an answer is, for example: the precision of my intelligence is 92% plus or minus 4% (which means it is probably between 88% and 96%).

So how much data you need depends on how certain you need to be.

Assuming your data is very independent, the problem isn't changing too fast, and you aren't trying to optimize the last 0.1% out of a very hard problem (like speech recognition):

- Tens of data points is too small a number to evaluate intelligence.
- Hundreds of data points is a fine size for a sanity check, but really not enough.
- Thousands of data points is probably a fine size for most things.
- Tens of thousands of data points is probably overkill, but not crazy.

A starting point for choosing how much data to use to evaluate intelligence might be this:

1. Ensure you have thousands of recent data points reserved for evaluation.
2. Ensure you have hundreds of recent data points for each important sub-population.
3. Use the rest of your (reasonably recent) data for intelligence creation.
4. Unless you have ridiculous amounts of data, at which point simply reserve about 10% of your data to evaluate intelligence and use the rest for intelligence creation.

But you'll have to develop your own intuition in your setting (or use some statistics, if that is the way you like to think).

Comparing Intelligences

Now we know some metrics for evaluating intelligence performance and how to measure these metrics from data. But how can we tell one if intelligence is going to be more effective than another? Consider trying to determine whether the genre of a book is romance or not:

- One intelligence might have a precision of 80% with a recall of 20%.
- Another intelligence might have a precision of 50% with a recall of 40%.

Which is better? Well, it depends on the experience, and the broader objectives of the Intelligent System. For example, a system that is trying to find the next book for an avid romance reader might prefer a higher recall (so the user won't miss a single kiss).

Operating Points

One tool to help evaluate intelligences is to select an *operating point*. That is, a precision point or a recall point that works well with your intelligent experience. Every intelligence must hit the operating point, and then the one that performs best there is used. For example:

- In a funny-web-page detector the operating point might be set to 95% precision. All intelligence should be tuned to have the best recall possible at 95% precision.
- In a spam-filtering system the operating point might be set to 99% precision (because it is deleting users' email, and so must be very sure). All intelligences should strive to flag as much spam as possible, while keeping precision at or above 99%.
- In a smart-doorbell system the operating point might be set to 80% recall. All intelligences should be set to flag 80% of the times a person walks up to the door, and compete on reducing false positives under that constraint.

This reduces the number of variables by choosing one of the types of mistakes an intelligence might make and setting a target. We need an intelligence that is 90% precise to support our experience—now, Mr. or Mrs. Intelligence creator, go and produce the best recall you can at that precision.

When comparing two intelligences it's often convenient to compare them at an operating point. The one that is more precise at the target recall (or has higher recall at the target precision) is better.

Easy.

Curves

But sometimes operating points change. For example, maybe the experience needs to become more forceful and the operating point needs to move to a higher precision to support the change. Intelligences can be evaluated across a range of operating points. For example, by varying the threshold used to turn a probability into a classification, an intelligence might have these values:

- 91% precision at 40% recall.
- 87% precision at 45% recall.
- 85% precision at 50% recall.
- 80% precision at 55% recall.

And on and on. Note that any intelligence that can produce a probability or a score can be used this way (including many, many machine learning approaches).

Many heuristic intelligences and many "classification-only" machine learning techniques do not have the notion of a score or probability, and so they can only be evaluated at a single operating point (and not along a curve).

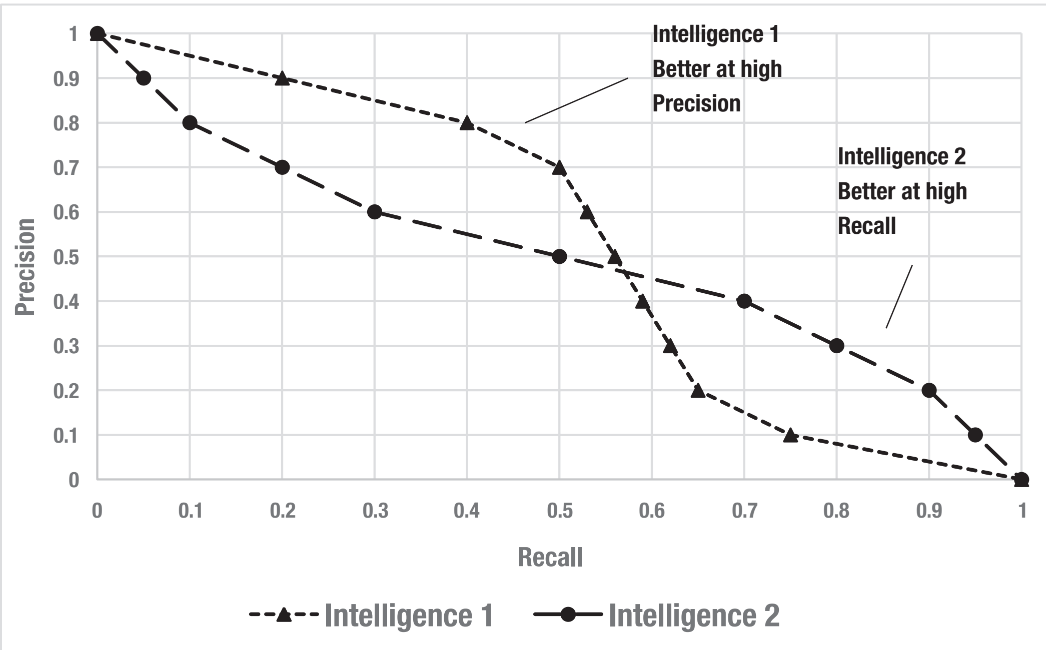
When comparing two intelligences that can make trade-offs between mistake types, you can compare them at any operating point. For example:

- At 91% precision, one model has 40% recall, and the other has 47% recall.
- At 87% precision, one model has 45% recall, and the other has 48% recall.
- At 85% precision, one model has 50% recall, and the other has 49% recall.

One model might be better at some types of trade-offs and worse at others. For example, one model might be better when you need high precision, but a second model (built using totally different approaches) might be better when you need high recall.

It is sometimes helpful to visualize the various trade-offs an intelligence can make. A *precision-recall curve* (PR curve) is a plot of all the possible trade-offs a model can make. On the x-axis is every possible recall (from 0% to 100%) and on the y-axis is the precision the model can achieve at the indicated recall.

By plotting two models on a single PR curve it is easy to see which is better in various ranges of operating points.



A similar concept is called a *receiver operating characteristic curve* (ROC curve). An ROC curve would have false positive rate on the x-axis and true positive rate on the y-axis.

Subjective Evaluations

Sometimes an intelligence looks good by the numbers, but it just…isn't… This can happen if you:

- Have a metric that is out of sync with the actual objective.
- Have a miscommunication between the experience and the intelligence.

- Have an important sub-population that you haven't identified yet.
- And more…

Because of this, it never hurts to look at some data, take a few steps back, and just think. Be a data-detective, take the user's point of view and imagine what your intelligence will create for them. Some things that can help with subjective evaluations include:

- Exploring the mistakes.
- Imagining the user experience.
- Finding the worst thing that could happen.

We'll discuss these in turn.

Exploring the Mistakes

Statistics (for example, those used to represent model quality with a precision and recall) are nice, they are neat; they summarize lots of things into simple little numbers that go up or down. They are critical to creating intelligence. But they can hide all sorts of problems. Every so often you need to go look at the data.

One useful technique is to take a random sample of 100 contexts where the intelligence was wrong and look at them. When looking at the mistakes, consider:

- How many of the mistakes would be hard for users to recover from?
- How many of the mistakes would make sense to the user (vs seeming pretty stupid)?
- Is there any structure to the mistakes? Any common properties? Maybe things that will turn into important sub-population-style problems in the future?
- Can you find any hints that there might be a bug in some part of the implementation or the evaluation process?
- Is there anything that could help improve the intelligence so it might stop making the mistakes? Some new information to add to the context? Some new type of feature?

In addition to inspecting random mistakes, it also helps to look at places where the intelligence was most-certain of the answer—but was wrong (a false positive where the model said it was 100% sure it was a positive, or a false negative where the model said the probability of positive was 0%). These places will often lead to bugs in the implementation or to flaws in the intelligence-creation process.

Imagining the User Experience

While looking at mistakes, imagine the user's experience. Visualize them coming to the context. Put yourself in their shoes as they encounter the mistake. What are they thinking? What is the chance they will notice the problem? What will it cost if they don't notice it? What will they have to do to recover if they do notice it?

This is also a good time to think of the aggregate experience the user will have. For example:

- How many mistakes will they see?
- How many positive interactions will they have between mistakes?
- How would they describe the types of mistakes the system makes, if they had to summarize them to a friend?

Putting yourself in your users' shoes will help you know whether the intelligence is good enough or needs more work. It can also help you come up with ideas for improving the intelligent experience.

Finding the Worst Thing

And imagine the worst thing that could happen. What types of mistakes could the system make that would really hurt its users or your business? For example:

- A system for promoting romance novels that classifies 100% of a particular romance-writer's books as non-romance. This writer stops getting promoted, stops getting sales, doesn't have the skills to figure out why or what to do, goes out of business. Their children get no presents for Christmas…
- All the web pages from a particular law firm get classified as funny. They aren't funny, but people start laughing anyway (because they are told the pages are funny). No one wants to hire a law firm full of clowns. The firm gets mad and sues your business—and they are a little bit pissed off and have nothing but time on their hands…

- The pellet griller controller's temperature sensor goes out. Because of this the intelligence always thinks the fire is not hot enough. It dumps fuel-pellet after fuel-pellet onto the fire, starting a huge, raging fire, but the intelligence still thinks it needs more…

These are a little bit silly, but the point is—be creative. Find really bad things for your users before they have to suffer through them for you, and use your discoveries to make better intelligence, or to influence the rest of the system (the experience and the implementation) to do better.

Summary

With intelligence, evaluation is creation.

There are three main components to accuracy: generalizing to new situation, making the right types of mistakes, and distributing mistakes well among different users/contexts.

- Intelligence can be good at contexts it knows about, but fail at contexts it hasn't encountered before.
- Intelligence can make many types of mistakes (including false positives and false negatives).
- Intelligence can make random mistakes, or it can make mistakes that are focused on specific users and contexts—the focused mistakes can cause problems.

There are specific techniques for evaluating classification, regressions, probability estimations, and rankings. This chapter presented some simple ones and some concepts, but you can find more information if you need it.

Intelligence can be evaluated with data. Some data should be held aside and used exclusively to evaluate intelligence (and not to create it). This data should be totally independent of the data used to create the intelligence (it should come from different users, different time periods, and so on).

An operating point helps focus intelligence on the types of mistakes it needs to make to succeed in the broader system. A precision-recall curve is a way of understanding (and visualizing) how an intelligence operates across all possible operating points.

It is important to look at the mistakes your intelligence makes. Try to understand what is going on, but also take your user's point of view, and imagine the worst outcome they might experience.

For Thought…

After reading this chapter, you should be able to:

- Describe what it means for an intelligence to be accurate.
- Evaluate the quality of an intelligence across a wide range of practical criteria.
- Create useful quality goals for intelligence and progress toward the goals.
- Take a user's point of view and see the mistakes an intelligence makes through their eyes.

You should be able to answer questions like these:

- Describe three situations where an Intelligent System would need an intelligence with very high recall.
- For each of those three, describe a small change to the system's goals where it would instead need high precision.
- Select one of the Intelligent Systems mentioned in this chapter and describe a potential sub-population (not mentioned in the chapter) where bad mistakes might occur.
- Consider the system for classifying books into genres. Imagine you are going to examine 100 mistakes the system is making. Describe two different ways you might categorize the mistakes you examine to help gain intuition about what is going on. (For example, the mistake is on a short book vs a long book—oops! now you can't use book length in your answer. Sorry.)

CHAPTER 20

Machine Learning Intelligence

Machine learning is a powerful technique for producing intelligence for large, hard, open-ended, time-changing problems. It works by showing the computer lots (and lots) of examples of contexts and the desired outcomes. The computer produces models from these examples. And the models can be used to predict the outcome for future contexts.

Machine learning can be a huge force multiplier, allowing the computer to focus on things it is good at (like tuning every detail of a gigantic model so it works effectively in millions of contexts), while humans can focus on what they are good at (like picking a model representation that will generalize well and building systems and intelligent experiences that turn models into customer and business value). This chapter will give an overview of machine learning, introducing the steps and some of the key issues.

How Machine Learning Works

A machine learning algorithm is essentially a search procedure that looks for accurate models, using training data to evaluate the accuracy. Generally, machine learning algorithms do the following:

- Start with a simple model.
- Try slightly refined versions of the model (usually informed by the training data).
- Check to see if the refined versions are better (using the training data).
- And iterate (roughly) until their search procedure can't find better models.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_20

Sometimes the refinements take the form of adding more complexity to the model (for example by adding more if-then-else tests to a decision tree). Sometimes the refinements take the form of adjusting parameters in the model (for example, updating the weights in a linear model).

For example, recall that a decision tree represents intelligence with a tree. Each node contains an `if` condition, with one child for when the `if`-test is true and one child for when the `if`-test is false. Here is a sample decision tree for predicting how much money a movie will make:

```
If <won Academy Award>:
        Is True: If <has a top 10 star in the cast>:
                Is True: then $100,000,000.
                Is False: then $1,000,000.
        Is False: If <opened on labor day weekend>:
                Is True: then $50,000,000.
                Is False: then $1,000.
```

Machine learning for decision trees produces increasingly complex trees by adding if-tests until no further additions improve the model (or until the model reaches some complexity threshold). For example, a refinement of the sample decision tree for movie-success-prediction might be this:

```
If <won Academy Award>:
        Is True: If <has a top 10 star in the cast>:
                Is True: If <opened on labor day weekend>:
                        Is True: then $500,000,000.
                        Is False: Then $10,000,000.
                Is False: then $1,000,000.
        Is False: If <opened on labor day weekend>:
                Is True: then $50,000,000.
                Is False: then $1,000.
```

This model is a bit more complex, and possibly a bit more accurate.

A human could carry out the same process by hand, but machine learning algorithms automate the process and can consider millions of contexts and produce hundreds of thousands of small refinements in the time it would take a human to type "hello world."

Factors an intelligence creator must control when using machine learning include these:

- **Feature Engineering**: How the context is converted into features that the machine learning algorithm can add to its model. The features you select should be relevant to the target concept, and they should contain enough information to make good predictions.
- **Model structure complexity**: How big the model becomes. For example, the number of tests in the decision tree or the number of features in the linear model.
- **Model search complexity**: How many things the machine learning algorithm tries in its search. This is separate from (but related to) structure complexity. The more things the search tries, the more chance it has to find something that looks good by chance but doesn't generalize well.
- **Data size**: How much training data you have. The more good, diverse data available to guide the machine learning search, the more complex and accurate your models can become.

These elements must be balanced to get the best possible models out of a machine-learning process.

The Pros and Cons of Complexity

One of the key challenges in machine learning is controlling the complexity of the models it produces. They key tension is as follows:

1. You need very complex models to solve hard, big, open-ended problems.
2. The more complexity you allow, the more chances you have to learn a model that misunderstands the concept.

Your job as an applied machine learning practitioner is to balance these tensions. A machine learning process can fail in two ways:

- It might *underfit* the concept, in that its understanding of the concept might too simple.
- It might *overfit* the concept, in that its understanding of the concept is wrong; it happens to work in some contexts, but it just isn't right.

Let's look at an example. Consider building intelligence that examines books and classifies them by genre—sci-fi, romance, technical, thriller, historical fiction, that kind of thing.

You gather 1,000 books, hand label them with genres, and set about creating intelligence. The goal is to be able to take a new book (one that isn't part of the 1,000) and accurately predict its genre.

Underfitting

If the approach you take is too simple to understand what a "genre" is, you will underfit. It might go something like this:

1. For each genre, find the word that is most indicative of the genre:
   - For a romance, this "indicative word" might be "love."
   - For a sci-fi, it might be "laser."
   - And so on
2. When you get a new book, check to see if it contains any of the indicative words and label the book with the associated genre:
   - If the book contains the word "love," call it romance.
   - If the book contains the word "laser," call it sci-fi.

If every book has one and exactly one of the indicative words, this model might work well. But most books have lots of words. What should we do with a book that uses both the words "love" and "laser"? Should we call it a romance or a sci-fi? Whichever choice we make, we'll be making mistakes, because some sci-fi books talk about love; and some romance books talk about lasers.

This approach to modeling (checking for a single word per genre) is just too simple. This simple approach *underfits* the concept of genre, and will not generalize well.

Overfitting

Now imagine another approach—using a decision tree that is very, very complex, essentially encoding the exact string of words in the book into a tree form, for example:

1. If the first word of the book is "When"
2. And the second word of the book is "the"
3. And the third word of the book is "Earth"
4. And so on, exactly matching every word in the book…
5. Then the genre is "Science Fiction"

This model is very complex, and highly accurate (on the data used to create it), but it would not generalize to new books at all.

And it sounds silly, exactly memorizing every word in a book, but that is just an example. The fundamental problem is that machine learning almost always models problems in ways that don't match the underlying phenomenon. Because of this, parts of the model will *happen to* work on everything you know about, but *fail to* work in new situations. The technical term for this is *overfitting,* and avoiding overfitting is one of the key challenges in creating effective intelligence.

When an intelligence underfits or overfits a problem, the intelligence will not be accurate at dealing with new contexts; it will not generalize.

Balancing Complexity

A better model for our book genre example would be somewhere between the two. Maybe it would check for the presence of the 50 words that are most highly correlated with each genre:

- For science fiction it might use laser, warp, star, beam, and so on.
- For romance it might use… other words…

Then you could use a decision tree, a linear model, or a neural network to automatically learn how important each word is to each genre and how to combine and weight them to accurately predict genre.

This would be a better approach than the other two approaches. You can see that, I can see that. But machines? Maybe not.

And this 50-words approach still is not right. It will certainly underfit the concept of genre, because it doesn't take the context of the words into account, the sentences they were used in, the ideas they describe.

But that's OK. The point of machine learning isn't to find the "correct" representation for a problem. The point is to find the modeling process that produces the best generalization. You'll need to consciously choose how and where to underfit and to overfit to achieve that.

Feature Engineering

Contexts in an Intelligent System encode all the data relevant to an interaction. This might include information about the user, their history, what they are doing, the content they are interacting with, and more. Feature engineering is the process of turning contexts into a representation that machine learning can work with. This includes making the information have the right format for the model representation. It also includes making the information work well with the search procedure used to learn the model.

The rest of this section talks about ways of doing this, including these:

- Converting data to useful form.
- Helping the model use the data.
- Normalizing feature values.
- Exposing hidden information.
- Expanding the context.
- Eliminating misleading things.

Converting Data to Useable Format

In general, machine learning algorithms take a *training set* as input. A training set consists of a set of *training examples*. Each training example contains a set of variables that encode the context. These variables are called *features*. Each training example also contains an outcome variable, which indicates the correct answer for the context. This is often called the *label*.

Recall that outcomes are commonly one of the following: a category from a set of possibilities (for a classification); a number (for a regression); or a probability (maybe a number from 0 to 1.0, which represents probabilities from 0 to 100%, scaled down to the range of 0.0–1.0).

A training example can be thought of as a row in a spreadsheet. One column contains the label; the rest of the columns contain *feature values*. Each training example in a training set must have the same number of features, although some of the features values can be missing—machine learning is robust to that sort of thing.

For example, if you are trying to build a model for predicting a person's age from the types of music they listen to, you might have a feature representing the gender, the eye color, and the number of Journey songs they listened to last week (Figure 20-1). Some users might have listened to 7; some might have listened to 14. The training examples for these users will have the column (feature) for *Number of Journey Songs* filled in with the correct values (7 or 14, or whatever). But maybe your Dad won't tell you how many Journey songs he's listened to—maybe he's shy about that sort of thing. So your Dad's training example still has a column (feature) for number of Journey songs, but the value is undefined (or –1, or 0, or some other special code depending on your tools).

Age (The Label)	Gender	Number of Journey Songs	Eye Color	...
23	Male	7	Hazel	...
12	Female	0	Brown	...
47	Male	-1	Blue	...
...	...	...	...	...

Figure 20-1. *Example features*

Machine learning systems typically work two types of features:

- *Numerical features*, which are integers or floating-point numbers (like the number of Journey songs listened to last week).
- *Categorical features*, which are labels from a small set of values (like the gender of the person).

Most machine learning models and algorithms support both numerical and categorical features. But sometimes a bit of pre-processing is required. For example, internally neural networks work only with numerical features. To feed categorical features to a neural network you have to convert them to numerical values. One common way is to use a *one-hot encoding*, which turns a categorical feature into N numerical features, one for each possible categorical value. For example, to encode gender this way you would have the following conversions:

- Male -> [0, 1]
- Female -> [1, 0]

Figure 20-2 shows this encoding in a table.

Age (The Label)	Gender is Female	Gender is Male
23	0	1
12	1	0

Figure 20-2. *One-hot encoding of gender*

Helping your Model Use the Data

Feature engineering is the art of converting context into features that work well for your problem and with your model structure. For example, consider the model for predicting age from songs the person listens to. One approach would be to create one feature for every artist that ever recorded a song, ever. Each training example has tens of thousands of columns, one per artist, indicating how many songs from that artist the user listened to.

This representation contains lots and lots of information, which should help the machine learning process avoid underfitting. It is also complex. It requires the learning algorithm to consider many, many things in making its decision and might encourage overfitting. So is this the best way to go?

Maybe, depending on your tools and how much data you have. The only way to know is to try a bunch of different things, build models, evaluate them, and see.

Here are some other possible approaches to creating features for this problem:

- **Break down the songs by decade**: Have one feature per decade where the value is the number of songs from that decade the person listened to.
- **Separate popular songs from obscure songs**: Because maybe people growing up during a particular decade would have broader exposure to the music, so listening to obscure songs might give them away. So each decade would have two features, one for popular song-listens from that decade and one for obscure song-listens from that decade.
- **Break down the songs by genre**: One feature per genre where the value is the number of songs listened to from the genre.
- **Some combination between genre and decade and popularity**: Where you do all three of the previous methods simultaneously.

And once you get going you'll think of all sorts of ways to break down the problem, to extract information from the context (the songs the person listened to and the patterns of the listening) and partially digest it to expose the most relevant parts of it to the machine learning algorithm.

Think of it this way. If you happen to know there is a relevant concept called "genre" and you expose the concept in your features in a useful way, then the machine-learning algorithm can leverage it. It takes work on your part, but it can help produce models that better-match what is going on in your problem space by encoding your prior knowledge about the world.

Contrast this with giving raw data to the machine learning process (so the algorithm can figure it all out on its own). This can be extremely effective and efficient in terms of the effort you'll need to expend—if you have lots and lots and lots of data so you can avoid overfitting.

Different algorithms will do better with different approaches. Learn your tools, gain intuition, and make sure your evaluation framework makes it very easy to explore.

Normalizing

Sometimes numerical features have very different values. For example, age (which can be between 0 and about a hundred) and income (which can be between 0 and about a hundred million). Normalization is the process of changing numerical features so they are more comparable. Instead of saying a person is 45 with $75,000 income, you would say a person is 5% above the average age with 70% above the average income.

One standard way to perform normalization is to post-process all numerical features and replace them with their normalized value:

1. Subtract the mean value (shifting the mean to zero).
2. Divide by the standard deviation of the variable.

For example, in each training example replace:

$$\text{Normalized\_Age} = \frac{(\text{Age - mean\_of\_all\_ages})}{\text{Standard\_Deviation\_of\_all\_ages}}$$

A learning algorithm can certainly work with un-normalized data. But doing the pre-processing can remove complexity, making the modeling task a bit easier so the algorithm can focus on other things. Sometimes it will help.

Exposing Hidden Information

Some features aren't useful in isolation. They need to be combined with other features to be helpful. Or (more commonly) some features are useful in isolation but become much more useful when combined with other features.

For example, if you are trying to build a model for the shipping cost of boxes you might have a feature for the height, the width, and the depth of the box. These are all useful. But an even more useful feature would be the total volume of the box.

Some machine-learning algorithms are able to discover these types of relationships on their own. Some aren't.

You can make *composite features* based on your intuition and understanding of the problem. Or you could try a bunch automatically, by creating new features by combining the initial features you come up with.

Expanding the Context

You might use external intelligence or lookups to add things to add information that isn't explicitly in the context. For example, you might do any of these:

- Look up the history of an identity in the context, like traffic to a web server.
- Look up the external temperature from a web service based on the location.
- Run a sentiment analysis program on a block of text and include the predicted sentiment as a feature.

Recall that you can only use these types of features if you can create them exactly the same way in the intelligence runtime and in the intelligence creation environment.

The next chapter will discuss more ways to organize intelligence, which will help you decide how to best leverage sources or legacy intelligences to expand the context.

Eliminating Misleading Things

Another approach is to delete features from your data.

WHAT!?!

Yeah. You just went through all this work to create features from contexts, now I'm saying you should delete some of them. Nuts!

But you may have created some really poor features (sorry). These can add complexity to the learning process without adding any value. For example, using a person's eye color to predict their age. Sure, every person has an eye color. Sure, it is in the context. But is it relevant to predicting how old the person is? Not really.

Including irrelevant features will reduce your ability to learn models that generalize well—because sometimes the irrelevant feature will look good by accident and trick your machine-learning algorithm into including it in the model.

Or you may have created way more features than your data will support. All of your features might be brilliant, but they can still lead to overfitting and hurt generalization.

So really you should:

1. Remove features that don't have any information about the outcome.
2. Select as many features as your data and model structure and search procedure will support, and select the best ones.

This is called *feature selection*. There are many techniques for performing feature selection. Here are some simple ones:

1. Measure the mutual information (think correlation) between each feature and the outcome feature, and remove the ones with lowest scores.
2. Train models with and without each feature, evaluate each model, and then remove features that don't help generalization accuracy.
3. Use your intuition and understanding of the problem to delete things that don't make sense.

These techniques can help debug your decisions about feature engineering, too. Create a new feature. Add it to the existing set of features. Train a new model—if the result isn't better, you might be on the wrong path. Stare at it for a while. Then stop and think. Maybe it will help you understand what is really going on.

Modeling

Modeling is the process of using machine learning algorithms to search for effective models. There are many ways an intelligence creator can assist this process:

- Deciding which features to use.
- Deciding which machine learning algorithms and model representations to use.
- Deciding what data to use as input for training.
- Controlling the model creation process.

But in all of these, the goal is to get the model that generalizes the best, has the best mistake profile, and will create the most value for your customers. This section talks about ways to do that, including common ways to adjust the modeling process to help.

Complexity Parameters

Every machine learning algorithm has parameters that let intelligence creators control how much complexity they add.

Some provide parameters that *control the size of the model the algorithm produces*. Some models add structure as they search, for example, adding nodes to a decision tree; adding trees to a random forest. These types of algorithms often have parameters that place hard limits on the amount of structure (number of nodes in the tree, or number of trees in the random forest). Many algorithms also have parameters for the minimum amount of training-set gain they need to see to continue the search.

When you are overfitting you should limit the size; when you are underfitting you should allow more size.

A machine learning algorithm might have parameters to *control the algorithm's search strategy*. The most common approach (by far) is to take greedy steps along the *gradient,* that is—to make the change that most improves the model's performance on the training data. Other options can include a bit of randomness, like random restarts (in case the search gets in a local maximum), or some lookahead.

Algorithms might also have parameters that control the size of steps taken in the search. That is, they find the gradient, and then they have to decide how far to move the model along the gradient. Smaller step size allows for more complexity. Some algorithms adaptively adjust the step size as the search continues, starting large to find a good general region, and then getting smaller to refine the model.

When you are overfitting you should use simpler search strategies; when you are underfitting you might try more complex search.

Some algorithms also use optimization algorithms directly; for example, by representing the problem as a matrix internally and using linear-algebra to find key properties. These algorithms might support various optimization options that trade off complexity.

You should basically always adjust these types of parameters based on the complexity of your problem and the amount of data you have. It's very, very rare to perform successful machine learning without tuning your algorithm's complexity parameters.

Identifying Overfitting

Recall that the goal of modeling is to build the model that is best at generalizing to new settings. One strategy to identify when a modeling process is starting to overfit is this:

1. Build a series of increasingly complex models on the training data, starting with brain-dead simple and progressing to crazy-complexity. If your features and model representation choices are reasonable, the more complex models should do better than the less complex ones—*on the training data*.
2. Now run these models on a hold-out test set. The brain-dead simple model should perform (roughly) worst. The next, slightly more complex model should perform (a bit) better. And the next more complex should perform better still. On and on—up to a point. Eventually, the more complex models will start to perform worse on the hold-out test data than the less complex ones. And the point where they start to perform worse is the point where your feature set, model search strategy, model representation, and available data are starting to get into trouble.

For example, see Figure 20-3.

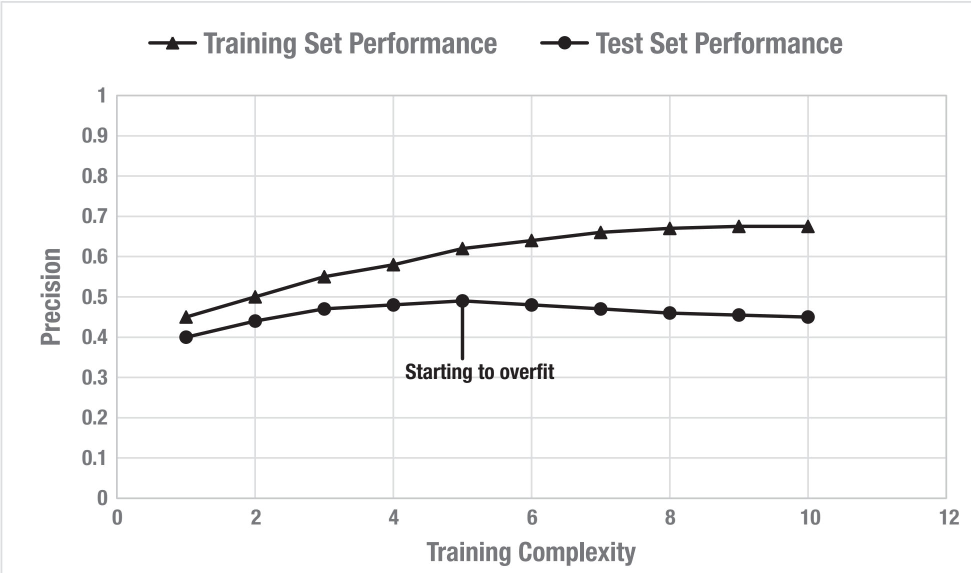


Figure 20-3. *An example of overfitting*

If you start overfitting you'll need:

- Features that better match the problem.
- A model structure that better matches the problem.
- Less search to produce models.
- *Or more data!*

More data is the best way to avoid overfitting, allowing you to create more complex and accurate models. And thank goodness you've done all the work to set up an Intelligent System with a great intelligent experience and lots of users to produce data for you. This is where Intelligent Systems can shine, allowing your intelligence to grow as more users interact with it.

Summary

Machine learning produces intelligence automatically from data. It involves having a computer search for models that work well on your data (in a training set). One of the key tensions is between increased accuracy and poor generalization based on overfitting.

Feature engineering is a critical part of machine learning. It involves converting contexts into representations that your model can work with. But it also involves understanding your representation, your problem, and the machine learning algorithm's search strategy well enough to present data in the right way. Common techniques include normalization, feature selection, and composite features.

The modeling process involves a lot of iteration, picking the right machine learning algorithm, tuning the parameters, and squeezing the most value out of the available training data by carefully managing complexity.

For Thought…

After reading this chapter, you should:

- Have a good conceptual understanding of machine learning, including features, models, and the modeling process and the key tension between complexity and overfitting.
- Know the steps of applied machine learning and roughly how a generic machine learning algorithm would work.
- Be able to take a machine learning toolkit and produce usable models, and be prepared to take more steps in learning specific algorithms and techniques.

You should be able to answer questions like these:

Describe a context for an Intelligent System you haven't used in any previous answers to the questions in this book.

- Create ten numerical features for the context.
- Create five categorical features for the context.
- Propose one way to make the features less complex.
- Create an example of a model that uses your features to overfit—go crazy, find the limit of an outrageous overfitting.
- Bonus: Design a very simple machine learning algorithm. What is your representation? What is your search? What are your complexity parameters?

CHAPTER 21

Organizing Intelligence

In most large-scale systems, intelligence creation is a team activity. Multiple people can *work on the intelligence at the same time*, building various parts of it, or investigating different problem areas. Multiple people can also *work on the intelligence over time*, taking over for team members who've left, or revisiting an intelligence that used to work but has started having problems. Some examples of ways to organize intelligence include these:

- Using machine learned intelligence for most things, but using manual intelligence to override mistakes.
- Using one machine learned intelligence for users from France, and a different one for users from Japan.
- Using the output of heuristic intelligence as features into a machine learned intelligence.

This chapter discusses ways to organize intelligence and the process used to create intelligence to make it robust, and to allow many people to collaborate effectively.

Reasons to Organize Intelligence

There are many reasons you might want to move from building a single monolithic intelligence (for example, a single machine-learned model) to an organized intelligence:

- **Collaboration**: Large-scale intelligence construction is a collaborative activity. You may have 3, or 5, or 15 people working on the intelligence of a single Intelligent System. And if you do, you'll need to find ways to get them all working together, efficiently, instead of competing to be the owner of the one-intelligence-to-rule-them-all.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_21

- **Cleaning up mistakes**: Every intelligence will make mistakes, and correcting one mistake will often make more mistakes crop up in other places. Also, trying to get a machine-learning-based intelligence to stop making a particular mistake isn't easy—it often requires some experimentation and luck. Combining intelligences can provide quick mistake mitigation to backstop more complex intelligences.
- **Solving the easy part the easy way**: Sometimes part of a problem is easy, where a few heuristics can do a great (or perfect) job. In those cases, you could try to trick machine learning to learn a model that does something you already know how to do, or you could partition the problem and let heuristics solve the easy part while machine learning focuses on the harder parts of the problem.
- **Incorporating legacy intelligence**: There is a lot of intelligence in the world already. Existing intelligences and intelligence-creation processes can be quite valuable. You might want to incorporate them into an Intelligent System in a way that leverages their strength and helps them grow even more effective.

Properties of a Well-Organized Intelligence

Organizing intelligence can be difficult and problematic. Done wrong, the layers of intelligence come to depend on the idiosyncrasies of each other. Any change to one intelligence causes unintended (and hard to track) changes in other intelligences. You can end up with a situation where you know you need to make some changes, but you simply can't—just like spaghetti code, you can have spaghetti intelligence.

A well-organized intelligence will be all of the following:

- **Accurate**: The organization should not reduce the accuracy potential too much. It should be a good trade-off of short term cost (in terms of lower immediate accuracy) for long term gains (in terms of higher accuracy over the lifetime of the Intelligent System).
- **Easy to Grow**: It should be easy for anyone to have an insight, create some intelligence, and drop it into the system.

- **Loosely Coupled**: The ability for one intelligence to influence the behavior of other intelligences should be minimized. The interfaces between the intelligences should be clear, and the intelligences shouldn't use information about the inner working of one-another.
- **Comprehensible**: For every outcome that users have, the system should be able to pinpoint the intelligence (or intelligences) that were involved in the decision, and the number of intelligences involved in each decision/outcome should be minimized.
- **Measurable**: For every part of the intelligence, it should be possible to determine how much that part of the intelligence is benefiting users.
- **Suportive of the Team**: The organization strategy should work with the team. The organization should allow intelligence creators' successes to amplify one another's work. It should avoid putting goals in conflict or creating the need for participants to compete in unproductive ways.

Ways to Organize Intelligence

This section discusses a number of techniques for organizing intelligence and the process of creating intelligence, and evaluates them against the key properties of a well-organized intelligence. Most large Intelligent Systems will use multiple of these methods simultaneously. And there are many, many options—the techniques here are just a starting point:

- Decouple feature engineering
- Multiple model searches
- Chase mistakes
- Meta-models
- Model sequencing
- Partition contexts
- Overrides

The following sections will describe these approaches and rank them according to how well they meet the criteria for well-organized intelligence. This is a subjective scale that is attempting to highlight relative strength/weaknesses as follows:

++	A real strength
+	Better than average
	Average
-	Worse than average
- -	A challenge that will need attention

All of these methods are viable, and used in practice. But you should be prepared to mitigate the challenges inherent in the organization strategy you choose.

Decouple Feature Engineering

One approach to organizing intelligence creation is to separate the feature engineering tasks so that each intelligence creator has a clear part of the context to explore and to turn into features. For example, if trying to understand a web page:

- One intelligence creator can focus on the content of the page, using standard approaches like bag of words and n-grams to convert words into features.
- Another intelligence creator can focus on understanding the semantics of the text on the page, using parts of speech tagging, sentiment analysis, and so on.
- Another could look at the history of the web site, where it is hosted, who created it, and what else they created.
- Another could explore the images on the web page and try to create features from them.
- Another could look at the properties of the user.

Each participant needs to be able to inject their feature extraction code into the modeling process. They need to be able to tweak model parameters to take best advantage of the new work. They need to be able to deploy their work.

Some challenges of decoupled feature engineering include these:

- **Conflict on model building**: One type of model/set of parameters might work better for one type of feature than for another. Participants will need to balance trade-offs to grow the overall system, and not simply cannibalize value from existing feature sets.
- **Redundant features**: Multiple approaches to feature creation could leverage the same underlying information from the context. The resulting features may be very similar to each other. Intelligence creators may have conflict about how to remove the redundancies.
- **Instability in feature value**: When a new feature is presented to a machine learning algorithm it will usually change the direction of the model-building search, which can have wild impacts on the value of other features and on the types of mistakes the model makes. Adding a new feature may require some global understanding of the feature set/model and some work on other parts of the feature-creation code to keep everything in balance.

In summary, the approach of decoupling feature engineering is

- **Accurate: Average**

  There isn't much compromise in this approach, and intelligence creators can work in parallel to make gains.
- **Easy to grow: +**

  The act of adding a few new features to an existing model is conceptually easy. Not the absolute simplest, but quite good.
- **Loosely coupled: Average**

  Features can interact with each other, but as long as you aggressively remove redundancy, the coupling should not be a major problem.
- **Comprehensible: Average**

  When trying to debug an interaction there aren't good tools to pinpoint problematic features, and many model types make it particularly difficult. Sometimes you are left with "try removing features one at a time and retraining to see when the problem goes away."

- **Measurable:** –

  It's easy to measure improvement when the features are initially added. It isn't so easy to track the contribution of the features over time (for example, as the problem changes).

- **Supportive of the Team: Average**

  When there are clear boundaries in the context things can work well, but there are certainly plenty of ways to end up with conflict as to which features should be in and which should be out, particularly if there are any runtime constraints (CPU or RAM).

Multiple Model Searches

Another way to organize intelligence is to allow multiple creators to take a shot at the model-building process. For example, maybe one team member is an expert with linear models, while another is a master of neural networks. These practitioners can both try to create the intelligence, using whatever they are most comfortable with, and the best model wins.

Using multiple model searches can be effective when:

- You have intelligence creators who are experienced with different approaches.
- You are early in the process of building your Intelligent System and want to cast a wide net to see what approaches work best.
- You have a major change in your system (such as a big change in the problem or a big increase in usage) and want to reverify that you have selected the right modeling approach.

But using multiple model searches can result in redundant work and in conflicts, because one approach will eventually win, and the others will lose.

The approach of multiple model searches is

- **Accurate:** -

  This approach makes it hard to leverage many intelligence creators over time. It should be used sparingly at critical parts of the intelligence creation process, such as when it is clear a change is needed.

- **Easy to grow: -**

  To ship a new intelligence you have to beat an old one. This means that new ideas need to be quite complete, and evaluated extensively before deploying.

- **Loosely coupled: Average**

  There is just one intelligence, so there isn't any particular coupling problem.

- **Comprehensible: Average**

  There is just one intelligence, so there isn't any particular comprehension problem.

- **Measurable: Average**

  There is just one intelligence, so there isn't any particular measurably problem.

- **Supportive of the Team: - -**

  This is a bit of a winner-take-all way of working, which means there are a modeling winner and a modeling loser. It also tends to promote wasted work—chasing a modeling idea that never pans out.

Chase Mistakes

Another approach is to treat intelligence problems like software bugs. Bugs can be assigned to intelligence creators, and they can go figure out whatever change they need to make to fix the problem. For example, if you're having trouble with a sub-population—say children—send someone to figure out what to add to the context, or what features to change, or what modeling to change to do better on children.

Intelligences will always make mistakes, so this approach could go on forever.

And one of the key problems with this approach is figuring out what mistakes are just sort of random mistakes, and which are systematic problems where a change in intelligence creation could help. When using this approach, it is very easy to fall into chasing the wrong problems, making everyone upset, and getting nowhere.

In my opinion, this approach should be used infrequently and only near the beginning of the project (when there are lots of legitimate bugs) or when there is a catastrophic issue.

The approach of chasing mistakes is

- **Accurate: -**

  Intelligence sometimes works this way (like with a sub-population problem), but it is easy to get drawn into chasing the wrong mistakes.

- **Easy to grow: -**

  Everyone needs to know everything to find and follow mistakes, develop productive changes, and deploy the fix. Also, this approach tends to lead to poor decisions about what problems to tackle.

- **Loosely coupled: Average**

  Doesn't really affect coupling.

- **Comprehensible: Average**

  Doesn't really affect comprehensibility.

- **Measurable: Average**

  Doesn't really affect measurability.

- **Supportive of the Team: -**

  This approach does not provide nice boundaries for people to work with. It is also easy to fix one mistake by causing another, and it won't always be clear that one fix caused the other mistake until much later. Done wrong, this approach can create a miserable work environment.

Meta-Models

The meta-model approach is to treat the predictions of the various intelligences in your system as features of a meta-intelligence. Every base intelligence runs and makes its decision, and then a meta-intelligence looks at all the proposed predictions and decides what the real output should be. Using meta-models can be

- Very accurate, because it brings together as many approaches as possible and learns which contexts each approach is effective in and which it struggles with.
- A great way to incorporate legacy intelligence. For example, when you find a new intelligence that is better than your original heuristics, you can throw away your heuristics… or you could use them as a feature in the new intelligence.
- A good way to get multiple intelligence creators working together. There are no constraints on what they can try. The meta-model will use the information they produce if it is valuable and ignore it if it isn't.

But meta-models can also be a bit of a nightmare to manage. Some complexities include these:

- The meta-intelligence and the base intelligences become tightly coupled, and changing any part of it might involve retraining and retuning all of it.
- If any piece of the system breaks (for example, one model starts behaving poorly) the whole system can break, and it can be very hard to track where and how problems are occurring.

If you want to use meta-models you will need approaches to control the complexity that interdependent models introduce, perhaps by

- Enforcing some structure about which models can change and when—for example, freezing the legacy intelligence and only changing it if you find severe problems.
- Building extra machinery to help retrain and retune all the intelligences that make up the system very, very easily.

In summary, the approach of using meta-models is

- **Accurate: ++**

  Short term, meta-models have the power to be the most accurate of the methods listed here. The cost of them is in the other areas. For raw accuracy, use meta-models.

- **Easy to grow: -**

  To ship a new intelligence you need to retrain the meta-intelligence, which risks instability in outcomes across the board. Careful testing is probably required.

- **Loosely coupled: - -**

  Changing any base intelligence usually requires retraining the meta-intelligence. Unintended changes in any of the intelligences (e.g. some change in one of the data sources it depends on) can affect the whole system, to the point of completely breaking it.

- **Comprehensible: - -**

  Every intelligence contributes to every decision. When there is a problem it can be extremely difficult (maybe impossible?) to track it down to a source.

- **Measurable: -**

  It is easy to measure a new intelligence when it is added to the system. It isn't so easy to track the contribution of the intelligences over time (for example, as the problem changes).

- **Supportive of the Team: Average**

  There can be conflict between intelligences, but they can be created independently. There may also be conflicts for resources when the intelligences need to run in a resource restrained environment.

Model Sequencing

Model sequencing is a restricted version of the meta-model approach in which the meta-model is constrained to be super-simple. In the sequencing approach, the models are put into order by the intelligence creator. Each model gets a chance to vote on the outcome. And the first model to vote with high confidence wins and gets to decide the answer.

This can be accomplished for classification by setting a default answer—if no one votes, the answer is “male”—and allowing each model to run with a high-precision operating point for the “female” answer. If any model is very sure it can give a high-precision “female” answer, then it does; if none of the models are certain, the default “male” is the return value.

Model sequencing has less accuracy potential than meta-models, which can combine all the votes simultaneously, but it is much easier to orchestrate and control.

The approach of model sequencing is

- **Accurate: Average**

  This approach trades off some potential accuracy for ease of management and growth.

- **Easy to grow: +**

  An intelligence creator can put a new model in the sequence (as long as it has high enough precision) without affecting any other part of the system.

- **Loosely coupled: ++**

  Models are completely uncoupled and are combined by a simple procedure that everyone can understand.

- **Comprehensible: ++**

  Every interaction can be traced to the piece of intelligence that decided the outcome, and each piece of intelligence can have a clear owner.

- **Measurable: Average**

  It is easy to measure how many positive and negative interactions each intelligence gives to users. The downside is that the first confident answer is taken, so other intelligences might not get all the credit (or blame) they deserve.

- **Supportive of the Team: +**

  Anyone can easily add value. Potential conflict points include what order to use to sequence of models and what precision threshold to demand. But telemetry should provide good data to use to make these decisions empirically, so they shouldn't make people argue—much.

Partition Contexts

Partitioning by contexts is another simple way to organize multiple intelligences. It works by defining some simple rules on the contexts that split them into partitions and then having one intelligence (or model sequence, meta-model, and so on) for each of the partitions. For example:

- One intelligence for servers in the US, one for all others.
- One intelligence for small web sites, one for large web sites, and one for websites that aggregate content.
- One intelligence for grey-scale images, one for color images.
- One intelligence for new users, one for users with a lot of history.

This approach has advantages by allowing you to use different types of intelligence on different parts of the problem, solve easy cases the easy way, and also control the incidence of mistakes made on various partitions. Of course, machine-learning algorithms can technically use this type of information internally—and probably pick better partitions with respect to raw accuracy—but manually partitioning can be very convenient for orchestration and organization.

The approach of partitioning is

- **Accurate: Average**

  This approach turns one problem into several problems. This doesn't have to affect accuracy, but it might, particularly as innovations in one area might not get ported to all the other areas where they could help.

- **Easy to grow: ++**

  An intelligence creator can define a specific partition and give intelligence that is tuned for it without affecting other partitions.

- **Loosely coupled: +**

  Models are completely uncoupled and are combined by an understandable procedure (unless the partitioning gets out of hand).

- **Comprehensible: +**

  Every interaction can be traced to the piece of intelligence that decided the outcome, and each piece of intelligence can have a clear owner.

- **Measurable: +**

  It is easy to measure how many positive and negative interactions each intelligence gives to users.

- **Supportive of the Team: +**

  Anyone can easily add value. Also, when one team member takes on a problem (by partitioning away something that was causing trouble for other models) it can be perceived as a favor: "I'm glad you took on those mistakes so my model can focus on adding this other value…"

Overrides

Overriding is an incredibly important concept for dealing with mistakes. The override structure for organizing intelligence works by having one blessed intelligence (usually created and maintained by humans) that can override all the other intelligences (usually created by machine learning)—no matter what they say.

One way this can be used is by hand-labeling specific contexts with specific outcomes. This can be used to spot-correct specific damaging mistakes. For example:

- This web page is not funny, no matter what any intelligence thinks.
- When all the toaster sensors read a specific combination, toast for 2 minutes (because we know exactly what product that is), no matter what the intelligence thinks.

Another way this can be used is by creating rules that can serve as guardrails, protecting from things that are obviously wrong. For example:

- If we've released 10 pellets into the pellet griller in the past 5 minutes, don't release any more, no matter what the intelligence says.
- If the web page has any of these 15 offensive words/phrases it is not funny, no matter what the intelligence says.

Override intelligence should be use extremely sparingly. It should not be trying to solve the problem; it should just be covering up for the worst mistakes.

The approach of using overrides is

- **Accurate: Average**

  As long as the overrides are used sparingly. They should be a backstop and not a complicated hand-crafted intelligence.

- **Easy to grow: ++**

  An intelligence creator (or an untrained person with some common sense) can define a context and specify an outcome. Tooling can help.

- **Loosely coupled: Average**

  The overrides are somewhat coupled to the mistakes they are correcting and might end up living in the system for longer than they are really needed. Over time they might turn into a bit of a maintenance problem if they aren't managed.

- **Comprehensible: +**

  Every interaction can be traced to any overrides that affected it. Intelligence creators might forget to check all the overrides when evaluating their new intelligence, though, so it can lead to a bit of confusion.

- **Measurable: +**

  It is easy to measure how many positive and negative interactions each override saved/gave to users.

- **Supportive of the Team: +**

  As long as overrides are used sparingly, they provide a simple way to make intelligence creators more productive. There is potential conflict between the intelligence creators and the people producing the overrides.

Summary

Most large Intelligent Systems do not have monolithic intelligences; they have organized collections of intelligence. Organizing intelligence allows multiple intelligence creators to collaborate effectively; to clean up mistakes cheaply, to use right types of intelligence to target the right part of the problem, and to incorporate legacy intelligence.

A well-organized intelligence will be: accurate, easy to grow, loosely coupled, compressible, measurable, and supportive of the team. It will, of course, also be accurate and sometimes organization needs to be sacrificed for accuracy.

There are many, many ways to organize intelligence. This chapter presented some of the prominent ones, but others are possible. Important organization techniques include: decoupling feature engineering; doing multiple model searches; chasing mistakes; meta-models; model sequencing; partitioning contexts; and overrides. Figure 21-1 shows the summary table.

Approach	Accurate	Ease of Growth	Loosely Coupled	Comprehensible	Measurable	Supports Team
Decouple Feature Engineering	Average	+	Average	Average	-	Average
Multiple Model Searches	-	-	Average	Average	Average	--
Chase Mistakes	-	-	Average	Average	Average	-
Meta-Models	++	-	--	--	-	Average
Model Sequencing	Average	+	++	++	Average	+
Partition Contexts	Average	++	+	+	+	+
Overrides	Average	++	Average	+	+	+

Figure 21-1. *Comparing approaches to intelligence organization*

For Thought…

After reading this chapter, you should:

- Understand what it takes to work on a large, complex Intelligent System, or with a team of intelligence creators.
- Be able to implement an intelligence architecture that allows the right intelligence to attack the right parts of your problem, and all participants to work together efficiently.

You should be able to answer questions like these:

- Describe an Intelligent System that does not need to have any intelligence organization—that is, it works with just a single model.
- What are some of the ways this might cause problems? What problems are most likely to occur?
- Design a simple intelligence organization plan that addresses the most likely problem.

PART V

Orchestrating Intelligent Systems

Chapters 22-26 discuss what it takes to run an Intelligent System. This includes taking control of all the parts of it, keeping its intelligence and experience balanced, growing it, debugging it, and ensuring that it is constantly achieving its goals.

This part discusses the tools needed, how to understand the mistakes an Intelligent System makes and what to do in response, and what to do if people start to abuse your system.

CHAPTER 22

Overview of Intelligence Orchestration

Intelligence orchestration is a bit like car racing. A whole team of people build a car, put all the latest technology into it, and get every aerodynamic wing, ballast, gear-ratio, and intake valve set perfectly—they make an awesome machine that can do things no other machine can do.

And then someone needs to get behind the wheel, take it onto the track, and win!

Intelligence orchestrators are those drivers. They take control of the Intelligent System and do what it takes to make it achieve its objectives. They use the intelligence-creation and management systems to produce the right intelligence at the right time and combine it in the most useful ways. They control the telemetry system, gathering the data needed to make the Intelligent System better. And they deal with all the mistakes and problems, balancing everything so that the Intelligent System is producing the most value it can for users and for your business. For example:

- If the intelligence gets better, the experience can be made more forceful to achieve better outcomes.
- If the problem changes, the intelligence management might need to be optimized to create and deploy intelligence differently or more quickly.
- If the system stops achieving its objectives, someone needs to figure out why and how to use the available systems to adapt.

This isn't easy.

Building Intelligent Systems and orchestrating them are very different activities. They require very different mindsets. And they are both absolutely critical to achieving success.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_22

This chapter describes what a well-orchestrated intelligence looks like and introduces the elements of orchestrating an Intelligent System, including the life-cycle, the orchestration environment, mistakes, and abuse.

Properties of a Well-Orchestrated Intelligence

A well-orchestrated Intelligent System will

- **Achieve its objectives reliably over time**: That is, it will grow, achieving its objectives better and better, until it reaches the point of diminishing return. Then it will stay there, producing value, as the user base, the problem, and the broader world change.
- **Have experience, intelligence, and objective in balance**: That is, the experience will be as forceful as the intelligence will support so the net benefit to users and business is maximized.
- **Have mistakes mitigated effectively**: So that high cost mistakes are understood, are detected quickly, and are corrected. And so lower cost mistakes are mitigated and do not put too large a burden on users or on the business.
- **Scale effectively over time**: So that the cost of maintaining the system in good shape (all the balancing, intelligence creation, mistake mitigations, and so on) scales favorably as the number of users and the complexity of intelligence scales. In particular, this means that machine learning and automation are used effectively.
- **Degrade slowly**: In that it should not require a bunch of manual effort to hold in a steady state. The effort of orchestration should be focused on finding opportunities, not scrambling to keep the wheels on the bus.

Why Orchestration Is Needed

Orchestrating an intelligence involves tuning it so that it produces the most value possible throughout its life cycle.

And right now you might be saying—wait, wait. I thought machine learning was supposed to tune the system throughout its life cycle. What is this? Some kind of joke?

Unfortunately, no. Artificial intelligence and machine learning will only get you so far. Orchestration is about taking those tools and putting them in the best situations so they can produce value—highlight their strengths, compensate for their weaknesses—and react as things change over time.

Orchestration might be needed because:

- The Intelligent System's objective changes.
- The Intelligent System's users change.
- The problem changes.
- The intelligence changes.
- The cost of running the Intelligent System changes.
- Someone tries to abuse your Intelligent System.

One or more of these will almost certainly happen during the life-cycle of your Intelligent System. By learning to identify them and adapt, you can turn these potential problems into opportunities.

Objective Changes

When your objectives change, your approach is going to have to change, too, and that means rebalancing the parts of your Intelligent System. You might want to change your objective when any of the following happen:

- **You understand the problem better**: As you work on something, you'll come to understand it better. You might realize that you set the wrong objective to begin with, and want to adapt. Recall that there are usually many layers of objectives. You might have a perfectly fine business objective, but come to understand that the user outcomes you are tracking are not contributing to the objective; or you might realize that the model properties you are optimizing aren't heading you in the right direction. In these cases you will want to change the objectives and adapt the system as needed to achieve the new goals.

- **You have solved the previous problem**: You may find that you have achieved what you wanted to achieve. The Intelligent System is running, users are getting good outcomes, your business is doing well. At this point you might want to set a higher goal: and adapt the system accordingly. Or you might want to figure out how to run the system more cheaply (which we will discuss later).
- **You realize the previous problem is too hard to solve**: Sometimes you have to admit defeat. You might have a system that has been running for a while, and just isn't getting to where you want it. You could shut down and find something else to do, or you could set an easier objective, change your approach, and build up more slowly.
- **You discover some new opportunity**: You might come up with a great idea. Something similar to your current Intelligent System, but just a little bit different. Maybe a totally new experience that can leverage the intelligence. Maybe a new type of intelligence you can build off of the existing telemetry. Sometimes chasing a new opportunity will require a new Intelligent System, but sometimes it can be accomplished with the systems you already have—by changing your objective.

Users Change

The user base of an Intelligent System will change over its life cycle, in various ways:

- **New users come**: And the new users might be very different than your existing users. For example, you might attract a new demographic, more casual users, users who exercise different parts of your Intelligent System, or users who just think differently. As your users change, you will have opportunities to learn and adapt. You will also have more telemetry to use to create intelligence and will have to deal with increasing sampling issues and intelligence-management complexity.
- **Usage patterns change**: As users learn to use your Intelligent System, they may change the way they interact with it. They will stop exploring how it works and become more focused on getting value

out of it. Experienced users will benefit from different interactions than novice users, and so you might want your Intelligent System to change as your users become experts.

- **Users change their perception of the experience**: Over time users might become fatigued with forceful, frequent experiences. They might start to ignore them. They might get increasingly irritated with them. An experience that worked well for the first month of usage might be totally wrong in the long run—and it might need to change.
- **Users leave**: And as they do, you will probably cry a little bit. But life will go on. Depending on which users leave, you might have the opportunity to change the balance of intelligence and experience to optimizing for the users you still have. You will also have less telemetry, which might reduce your ability to create intelligence. Automated intelligence, especially, might need to be revisited—for example, to leverage more legacy data instead of relying on newer data.

Problem Changes

Intelligent Systems are for big, hard, open-ended, time-changing problems. These problems—by definition—change over time:

- **Time-changing problems are always changing**: And so the approaches and decisions you made in the past might not be right for the future. Sometimes a problem might be easy. At other times it might get very hard. Sometimes you will see lots of a particular type of context. At other times you won't see those contexts at all. As a problem changes, there is almost always opportunity to adapt and achieve better outcomes through orchestration.
- **Usage patterns changing**: As your users change how they interact with your Intelligent System, the problems you need to solve can become very different. For example, users behave differently in the summer than they do in the winter; they behave differently on weekends than weekdays; they interact with new content differently from how they interact with old content.

- **You start to solve the problem**: In that the intelligence starts to catch up with a big, hard problem. In this case you might want to revisit the approach. Maybe you will want to start locking in behavior instead of continuing to grow, using more intelligence caches (lookup tables) and investing less in telemetry.

Intelligence Changes

The intelligence of your system will change, when you do any of the following:

- **Get more data to build better intelligence**: As users interact with your system, the intelligence will have more data, and should get better. When intelligence is better, experiences can become more forceful.
- **Try a new approach that makes things a lot better**: Data unlocks possibilities. Some of the most powerful machine learning techniques aren't effective with "small" data, but become viable as users come to your Intelligent System and you get lots and lots of data. These types of changes can unlock all sorts of potential to try new experiences or target more aggressive objectives.
- **Try a new approach that makes things worse**: But you won't be perfect. Sometimes intelligence evolutions go backwards. This can be a particular problem with intelligence that is combined in aggressive ways (tightly coupled or hard to comprehend). It can also be a problem when you rely on lots of human intervention in intelligence creation (for example with heuristics)—sometimes the complexity just crosses a tipping point and a brittle intelligence-creation plan can't keep up. In these cases you might need to back off other parts of the system to give yourself time to get out of the problem.

Costs Change

Big systems will constantly need to balance costs and value. Changes to save money can require compromises in other places. Put another way, you might be able to change your experience or intelligence in ways that save a lot of money, while only reducing value to users or your business by a little. You might try to reduce either or both of these:

- **Telemetry costs**: By changing the way you sample data. This reduces your ability to measure your system and to create intelligence, but it might be a good trade-off. For example, you might collect less data when intelligence is not growing much, sample less from parts of the problem that are already solved, or remove parts of the contexts that aren't contributing to useful features (and better models).
- **Mistake costs**: Managing mistakes can be very expensive. We'll discuss dealing with mistakes more in Chapter 24. But you might find that you need to make fewer, less expensive mistakes, and you might need to change all sorts of things in the system to do this.

Abuse

Unfortunately, the Internet is full of trolls. Some of them will want to abuse your service because they think that is fun. Most of them will want to abuse your service (and your users) to make money—or to make it harder for you to make money. Left unmitigated, abuse can ruin an Intelligent System, making it such a cesspool of spam and risk that users abandon it.

In order to protect your Intelligent System from abuse you need to understand what abusers want, and then you need to do something to make your system less interesting to potential abusers. Sometimes a small tweak will do it. Sometimes combating abuse will become a major part of running your Intelligent System.

But abuse is a tough thing. Once one abuser figures out how to make a tenth of a penny per interaction off your users they will scale. They will tell their friends. Abuse can show up quickly and violently, so you need to be prepared to identify it and react.

We'll discuss Adversaries and Abuse in detail in Chapter 25.

The Orchestration Team

Orchestrating an Intelligent System requires a diverse set of skills. In particular, the team should:

- Be domain experts in the business of the Intelligent System, so they understand the objectives instinctively and know what users want from their interactions with the system.
- Understand experience and have the ability to look at interactions and decide how to make improvements in how intelligence is presented to users—what is easy, what is hard, what users will like, and what they won't.
- Understand how the implementation works so they know how to trace problems and have some ability to improve the implementation as needed.
- Be able to ask all sorts of questions of data and understand and communicate the answers.
- Know how to do applied machine learning, to be able to create new intelligence and inject it into the system when needed.
- Get satisfaction from making a system execute effectively day in and day out. Not everyone likes this—some people like to invent new things. Orchestration is about excellence in execution—taking some of the most awesome machines in the world, getting them on the race track, and winning.

An orchestration team could consist of a small number of jacks-of-all-trades. Or it could be made up of participants with diverse skills who work well together.

Summary

An Intelligent System needs to be orchestrated throughout its life cycle to succeed. Orchestration is the process of keeping all the parts of the system doing what they are good at, and supporting them where they run into problems. A well-orchestrated Intelligent System will do all of the following

- Achieve its objectives reliably over time.
- Have experience, intelligence, and objective in balance.
- Have mistakes mitigated effectively.
- Scale effectively over time.
- Degrade slowly.

One key activity of orchestration is rebalancing the intelligence and the experience over the Intelligent System's life-cycle. For example, making the experience more forceful when the intelligence improves; or making it less forceful when the problem gets harder.

Orchestration also involves dealing with mistakes, understanding what they are and when they are occurring, and then managing the system to make them less costly. Reasons orchestration would be needed include these:

- The objective changes.
- Your users change.
- The problem changes.
- Intelligence changes.
- The system needs to get cheaper to run.

Adversaries and abusers often become problems for successful Intelligent Systems. They must be identified and then convinced that your system isn't worth their time.

For Thought…

After reading this chapter, you should

- Understand what it means to orchestrate an Intelligent System and why it is needed.
- Know what type of team is needed to orchestrate an Intelligent System.

You should be able to answer questions like these:

- Imagine that a system's intelligence begins to behave poorly. What changes that aren't intelligence-based could be made to mitigate the issue?
- Pick two of your favorite Intelligent Systems. Which would be harder to orchestrate? Why?
- Describe the types of problems an Intelligent System might have if its user base doubles—and all the new users are from a different country than the previous users.

CHAPTER 23

The Intelligence Orchestration Environment

This chapter discusses some of the common activities that contribute to success at intelligence orchestration. These are examples of the types of things that can help get the most out of an Intelligent System, and include the following:

- **Monitoring the Success Criteria**: To know if the system is achieving its goal and if it is getting better or worse.
- **Inspecting Interactions**: To experience contexts as the user experienced them and see the outcomes the user achieved.
- **Balancing the Experience**: To make it more or less forceful, maybe changing the type of interactions, or maybe changing the prominence and frequency of the interactions.
- **Overriding the Intelligence**: To correct for infrequent bad mistakes, or to help optimize common contexts.
- **Creating New Intelligence**: Managing intelligence and producing new models to deal with emergent issues as the problem, the users, or the scope of data changes.

Intelligent Systems can benefit from all of these activities and more. Some of them can be built as part of the implementation—for example, a metrics dashboard that shows progress on the overall objectives. Some of them can be handled in a more ad hoc fashion—for example, querying raw telemetry sources to piece together interactions that are causing problems.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_23

This chapter discusses orchestration activities in turn and gives some ideas of how you can invest in each of the areas (if you choose to), along with some criteria for deciding how much to invest.

Monitor the Success Criteria

You must be able to tell if the system is achieving its goals. This involves taking telemetry, correcting for any sampling and outages, and producing an answer that all participants can understand—maybe a few numbers, maybe a graph. Recall that this can involve any of the following:

- **Business objectives and leading indicators**, particularly showing how they vary between users who heavily use the intelligent part of the system compared to those who don't.
- **User outcomes**, indicating how effective the Intelligent System is at helping users achieve the outcomes you want.
- **Model properties**, indicating how often the intelligence is correct or incorrect (no matter what outcomes the user ended up achieving).

Monitoring success criteria is critical. Every functional Intelligent System must have people who know how well it is doing, every day, and who care that it is achieving its goals.

Levels of investment for monitoring success criteria include:

- **Ad Hoc**: Where a few people have the skill to measure the systems objectives and produce answers, maybe by doing some cross checking of current telemetry settings, tweaking a script, and running it, copying into a presentation form (like a chart), and distributing.
- **Tool-Based**: Where anyone on the team can run a tool that creates and then runs the correct query and outputs the answer in a suitable presentation format.
- **Automated**: Where the system automatically runs the queries on a regular basis and archives the results somewhere that all participants can find them.

- **Alert-Based**: where the system automates the metrics, but also monitors them and provides alerts to participants when something happens, including
  - **A major degradation**: Where the metrics are far worse than they were at the previous measurement. For example, the number of errors went up 10% since yesterday.
  - **A significant, sustained erosion**: Where the metrics have trended gradually downward long enough to cross some threshold. For example, the fraction of users getting good outcomes has gone down 10% over the course of the last month.
- **Population Tracking**: Where the system tracks metrics for important sub-populations as well as the overall population. This might include people from a particular location, with a particular usage pattern, with a particular demographic, and so on.

Criterion for investing in monitoring success criteria:

- Almost every Intelligent System should implement alert-based monitoring for success criteria and for critical factors that contribute to success criteria.

Inspect Interactions

Inspecting an interaction involves gathering all the telemetry related to a particular user interaction and being able to see the interaction end-to-end. This includes the user's context at the time of the interaction, the version of the intelligence running; the answer given from the intelligence, the experience that resulted from this intelligence answer, how the user interacted with the experience, and what outcome the user eventually got.

This is important for debugging problems, for tracking mistakes, for understanding how users are perceiving the Intelligent System, and for getting intuition about user experience options.

Levels of investment for inspecting interactions include

- **Ad Hoc**: Where a few people have the skill to find all the parts of an interaction and understand how they relate. They may have to do some detective work to identify a specific interaction (or interactions with a certain property) and, based on sampling, may or may not be able to find any particular interaction.
- **Tool-based**: Where anyone on the team can identify an interaction (maybe by user and time) see the relevant telemetry. The tool may also support querying for specific types of interactions (maybe where the intelligence gave a particular answer and the user got a particular outcome) and inspect a sample of them.
- **Browser-based**: Where anyone on the team can find interactions as with the tool, but then experience the interaction as the user would have, seeing the experiences that the user saw, the buttons they clicked, the outcome they got, and so on.
- **Getting Data from Users**: Where orchestrators have the opportunity to flag specific types of interactions and ask users questions about their experience. For example, this can be done by specifying some contexts and then surveying a small fraction of users who hit that context. These surveys might ask the user to rate their experience, answer a question or two, or indicate if they had a good or bad outcome.

Criteria for investing in inspecting interactions:

- If you need a broad set of people to be able to understand interactions. For example, so experience creators and business stakeholders can understand how users are experiencing the Intelligent System.
- If you need to build support capabilities to help deal with mistakes. Tools can allow humans who aren't experts in the implementation to participate.
- If your system has high cost mistakes and you expect a major part of orchestration will involve regular mitigation, then you will want to invest in inspecting interactions.

Balance the Experience

As the problem, the user base, and the intelligence change, they create opportunities to optimize the experience. For example, when the intelligence is new—and poor quality—the experience might be very passive. Maybe it doesn't show up very often. Maybe it shows up in ways that are easy for users to ignore. But over time the intelligence will get better, and users might have more positive outcomes with more forceful experiences.

Levels of investment for balancing the experience include:

- **Ad Hoc**: Where changes to the experience are made by changing code and redeploying the software.
- **Parameter Updates**: Where the orchestrators can change parameters that affect the experience and push these parameters out relatively cheaply (maybe as intelligence is updated). Parameters might include:
    - The frequency of interaction.
    - The color or size of a prompt.
    - The text to use in the experience.
    - The threshold for automating an action.
    - And so on.
- **Experience Alternatives**: Where multiple experiences are created and orchestrators have the ability to switch between them (using something similar to a parameter). For example, maybe you create one version of the experience with a subtle prompt, one with a prominent prompt, and one that automates an action. Then the orchestrator can switch between these to achieve objectives over time.
- **Experience Language**: Where orchestrators can author and deploy experiences out of band of code changes, similar to separating intelligence from implementation. This might be accomplished by specifying experiences in scripts that clients download and execute. Or it might be more curtailed, so that non-engineers can safely make changes; for example an experience mark-up language with a restricted runtime.

Criteria for investing in balancing the experience:

- If your orchestration team does not have engineering resources, then you probably want to create some controls for the experience that are not engineering-based.
- If it takes a long time to deploy new client code to your customers, you might want to invest in ways to control experience through parameters and data-file updates.
- If you think you'll be changing your experience many times during the life-cycle of your Intelligent System, you might choose to invest in making it easy.
- If none of those are true—you do have engineering resources, you can easily deploy code, and you don't expect too many changes—it will almost certainly be cheaper to take an ad hoc approach to experience updates.

Override Intelligence

Overriding intelligence involves identifying a few contexts and hand-coding the answer the intelligence should provide for those contexts. You might want to do this to correct costly (or embarrassing) mistakes. You might want to do this to optimize a few common contexts. You might want to do this to support some business goal (like promote a piece of content in your system). You probably don't want to override intelligence too much—but when you need to, you are going to really need to.

Levels of investment for overriding intelligence include:

- **Ad Hoc**: Where you rely on intelligence creators to hack the learning process to try to achieve specific outcomes (which is very hard) or on engineers to hard-code the desired overrides into code and deploy them to customers.
- **As an Intelligence Feed**: Where overrides are treated as a very simple intelligence source that is deployed with the system's other intelligence and has the highest priority in the runtime. Perhaps represented using a data file in some simple format that maps contexts to outcomes.

- **Tool-based**: Where you treat overrides as an intelligence feed but create tooling to support orchestrators. Tools should include functions like these:
    - Ensuring the contexts to override are specified correctly.
    - Giving feedback on the prevalence of the specified contexts and the current intelligence responses for those contexts.
    - Giving feedback on existing overrides, including how many there are and how many users they are impacting.
    - Tracking who is doing overrides and how good/bad they turn out to be.
    - Managing expiration of overrides.
- **Browser-Based**: Where the tools are connected into a support suite, including ways to find interactions, view them, and override them all in one place.

Criteria for investing in overriding intelligence:

- If your system can make very costly mistakes you will almost certainly want to invest in overriding intelligence, possibly adding tracking and work-flow management around the tools discussed here. We'll discuss this in more detail in the next chapter, on dealing with mistakes.
- Overriding intelligence can also be a very helpful buffer for intelligence creators. The last thing you want is for lots of participants in the Intelligent System to use the product, find zillions of little problems, and file them as bugs against the intelligence creation process. Intelligence is going to make mistakes. Beating up the creators for every flaw will not help. Having the ability to override a few of the more important problems can make everyone feel better.

Create Intelligence

An important part of orchestration is controlling and creating the intelligence that drives your Intelligent System. This section recaps and summarizes some of the investments that can help intelligence grow and create impact.

Investments in creating intelligence might include:

- **Intelligent Management**: This will include the following:
  - Controlling when and how intelligence is updated and deployed.
  - Controlling what intelligence lives where.
  - Adding new sources of intelligence to the system.
  - Controlling how intelligence is combined.
- **Automated Machine-Learning Implementations**: These produce new intelligence on a regular basis with little or no oversight. These systems may provide controls to change the way intelligence is produced, including the following:
  - How much data to use.
  - Which data to use.
  - How often to run.
  - How much time to spend searching for good models.
  - Any other parameters of the modeling process that are tunable.
- **Intelligence Creation Environments**: Where the telemetry that captures context and outcome are gathered and made available for machine learning, including trying new machine-learning algorithms, new feature engineering, and incorporating new insights. These can produce ad hoc new intelligence, or find ways to improve existing automated learning systems.
- **Support for Feature Engineering**: Where intelligence creators are able to try new features very easily, and if they find ones that work are able to deploy them to customers quickly and safely.
- **Telemetry Systems**: Where the orchestrator can specify what data to collect and how much of it to collect.

Criteria for investing in creating intelligence:

Intelligence creation is kind of important for Intelligent Systems (it says so right in the name). You will probably end up investing in multiple systems to support this process.

The most effective tools tend to:

- Help prevent errors.
- Automate mundane tasks.
- Simplify multi-step processes.
- Leave some simple audit trail.

When building tools to support intelligence creation, avoid the temptation to mess with the core intelligence creation tools (like machine-learning systems or runtimes). These are generally standard and innovating with them is unlikely to be core to your value proposition.

Summary

This book has introduced many tools that are important for running an Intelligent System. This chapter summarized some of the key ones, including some criteria and approaches for investing in them. These include:

- Monitoring the success criteria
- Inspecting interactions
- Balancing the experience
- Overriding intelligence
- Creating intelligence

You can build a successful Intelligent System without investing much in these areas during implementation. But you'll probably need to do all of these activities during the system's life-cycle. You can decide how labor intensive you want to be, and how much you want to invest in tools.

And keep in mind that having good tools allows you to change your staffing over time from the system's creators (who often like to work on new exciting things) to skilled orchestrators (who need to be generalists with drive for sustained excellence). These are very different skill sets.

Often the orchestration environment will evolve as the Intelligent System does, with small investments being made where they add the most value over an extended period until no further investments make sense.

For Thought…

After reading this chapter, you should:

- Know the common tools needed to keep an Intelligent System healthy.
- Understand ways to start with low investment and scale the investment over time.
- Have some understanding of when the various tools will help, and which are most important for your Intelligent System.

You should be able to answer questions like these:

Imagine an important Intelligent System that exists today.

- Which area do you think it spends the most orchestration time: monitoring success, inspecting interactions, balancing the experience, overriding intelligence, or creating intelligence?
- What would you propose to reduce the cost in that area?

CHAPTER 24

Dealing with Mistakes

There will be mistakes. Humans make them. Artificial intelligences make them, too—and how. Mistakes can be irritating or they can be disastrous.

Every Intelligent System should have a strategy for identifying mistakes; for example, by monitoring critical metrics and giving users easy ways to report problems.

Every Intelligent System should also have a strategy for dealing with mistakes. Perhaps it's done by updating intelligence; perhaps by having humans override certain behaviors by hand; perhaps by offering a workaround or refund to affected users.

Some mistakes will be very hard to find. Some will be very hard to fix (without introducing new mistakes). And some will take a very long time to fix (hours to deploy a new model or months to come up with a new intelligence strategy).

This chapter will discuss ways of dealing with mistakes, including these topics:

- The types of mistakes the system might make (especially the bad ones).
- Reasons intelligences might make mistakes.
- Ways to mitigate mistakes.

Every orchestrator of an Intelligent System should embrace the reality of mistakes.

The Worst Thing That Could Happen

Ask yourself: What is the worst thing my Intelligent System could do?

- Maybe your Intelligent System will make minor mistakes, like flashing a light the user doesn't care about or playing a song they don't love.
- Maybe it could waste time and effort, automating something that a user has to undo, or causing your user to take their attention off the thing they actually care about and look at the thing the intelligence is making a mistake about.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_24

- Maybe it could cost your business money by deciding to spend a lot of CPU or bandwidth, or by accidentally hiding your best (and most profitable) content.
- Maybe it could put you at legal risk by taking an action that is against the law somewhere, or by shutting down a customer or a competitor's ability to do business, causing them damages you might end up being liable for.
- Maybe it could do irreparable harm by deleting things that are important, melting a furnace, or sending an offensive communication from one user to another.
- Maybe it could hurt someone—even get someone killed.

Most of the time, when you think about your system you are going to think about how amazing it will be, all the good it will do, all the people who will love it. You'll want to dismiss its problems; you'll even try to ignore them.

Don't.

Find the worst thing your system can do.

Then find the second worst.

Then the third worst.

Then get five other people to do the same thing. Embrace their ideas and accept them.

And then when you have fifteen really bad things your Intelligent System might do, ask yourself: is that OK?

Because these types of mistakes are going to happen, and they will be hard to find, and they will be hard to correct.

If the worst thing your system might do is too bad to contemplate, you might want to design a different system—one that couldn't do that bad thing, ever, no matter what the intelligence says. Maybe you make sure a human is part of the decision process. Maybe you use a less forceful experience. Maybe you find something completely different to do with your life…

Because the intelligence will make mistakes and, eventually, the worst thing will happen.

Ways Intelligence Can Break

An Intelligent System will make mistakes for many different reasons. Some of them are implementation or management problems; some of them are intelligence problems. This section discusses these potential problem sources, including the following:

- System outages
- Model outages
- Intelligence errors
- Intelligence degradation

The first step to fixing a mistake is understanding what is causing it.

System Outage

Sometimes computers crash. Sometimes the Internet is slow. Sometimes network cables get cut. Sometimes a system has subtle bugs in the way its systems interact. These are problems with the implementation or the operation of your Intelligent System, but they might show up the same way intelligence mistakes do, in user reports, escalations, and degrading metrics.

Isolating these types of problems can be difficult in large systems, particularly when intelligence is spread between clients (which are in different states of upgrade) and multiple servers (which can live in various data centers).

Catastrophic outages are usually easy to find—because everything tanks. But partial outages can be more subtle. For example, suppose 1% of your traffic is going to a particular server and the server bombs out in a crazy way. One percent of your users are getting a bad experience, and maybe they are reporting it, over and over… But that's just 1% of your user base. 99% of your users aren't bothered. Would you ever notice?

System outages should be rare, and they should be fixed immediately. If they become prevalent they will paralyze intelligence work—and they will be just plain bad for morale.

Model Outage

Related to system outages, a model outage is more an implementation problem than an intelligence problem—but it will have similar symptoms.

Model outages can occur when:

- A model file is corrupted in deployment.
- A model file goes out of sync with the code that turns contexts into features.
- The intelligence creation environment goes out of sync with the intelligence runtime environment.
- An intelligence goes out of sync with the experience.

These problems can be very hard to find—imagine if some feature code gets updated in the intelligence creation environment, but not in the intelligence runtime. Then when a new model (using the updated feature code) is pushed to the runtime (using the out-of-date feature code) it will be confused. It will get feature values it doesn't expect. It will make mistakes. Because of this, maybe the accuracy is 5% worse in the runtime than it is in the lab. All the testing in the lab shows that the intelligence is working fine, but users are getting a slightly worse experience.

Because these problems are so hard to find, every intelligence implementation should have checks and double-checks to make sure the intelligence-creation environment is in sync with the runtime environment, that everything is deployed correctly, and that all components are in sync.

Intelligence Errors

When the models that make up intelligence don't match the world perfectly (and they don't), there will be mistakes. Recall that creating intelligence is a balancing act between learning a very complex model that can represent the problem and learning a model that can generalize well to new contexts. There will always be gaps—places where the model isn't quite right.

And these gaps cause mistakes, mistakes that are hard to correct through intelligence creation. You can try another type of model, but that will make its own (new) types of mistakes. You can get more data, but that has diminishing returns. You can try more feature engineering—and it usually helps. But these types of mistakes will always exist.

They will appear a bit random. They will change over time (as the training data changes). They aren't easy to correct—it will require sustained effort, and it will get harder the further you go.

One additional challenge for intelligence errors is figuring out which part of the intelligence is responsible. When models are loosely coupled (for example, when they have an order of execution and the first model to make a statement about a context wins), it can be easy to determine exactly which model gave the incorrect answer. But when models are tightly coupled (for example, when the output of several models is combined using a complex heuristic or a meta-model), a mistake will be harder to track. If six models are each partially responsible for a mistake, where do you begin?

Intelligence Degradation

When an open-ended, time-changing problem changes, the intelligence you had yesterday will not be as good today. Changing problems compound generic intelligence errors because new mistakes will occur even when you don't change anything. Further, training data for the "new" problem will take time to accumulate, meaning you may need to wait to respond and you may never be able to get enough training data to learn any particular version of the problem well (by the time you learn it, it isn't relevant any more).

There are two main categories of change:

1. Where new contexts appear over time (or old ones disappear), in which case you will need to create new intelligence to work on the new contexts, but existing training data can still be used on old contexts.

2. Where the meaning of contexts changes over time, in which case existing training data can be misleading, and you'll need to focus on new telemetry to create effective intelligence.

One way to understand the degradation in your Intelligent System is to preserve old versions of your intelligence and run a spectrum of previous intelligences on current data—the intelligence from yesterday, from five days ago, from ten days ago, and so on. By looking at how mistakes change, you can gain intuition about the way your problem is evolving and use that intuition when choosing how to adapt.

Mitigating Mistakes

Random, low-cost mistakes are to be expected. But when mistakes spike, when they become systematic, or when they become risky or expensive, you might consider mitigations.

This section discusses the following approaches to mitigating errors:

- Investing in intelligence
- Balancing the experience
- Adjust intelligence management parameters
- Implementing guardrails
- Overriding errors

Invest in Intelligence

In a healthy Intelligent System, the intelligence will be constantly improving. One way to deal with mistakes is to wait for the intelligence to catch up.

In fact, almost every other approach to mitigating mistakes degrades the value of the Intelligent System for users who aren't having problems (by watering down the experience); or it adds complexity and maintenance cost in the long run (by adding manual tweaks that must be maintained). Because of this, improving the intelligence is a great way to deal with mistakes—when it is possible. The best ways to invest in improving intelligence with respect to mistakes are these:

1. Get more relevant telemetry or training data that contains the contexts where mistakes are occurring. This might allow the intelligence creation to start solving the problem with very little work.
2. Help intelligence creators prioritize the parts of the system they spend time on by categorizing mistakes into categories (by locals, age, user properties, and so on) and prioritizing the categories. Intelligence creators can then work on features and modeling that helps in those specific areas, maybe via partitioning and focused modeling.
3. Provide more resources to intelligence creation in terms of people and tools.

These investments will improve the overall quality of the intelligence and, over time, a rising tide will raise all ships.

And perhaps the worst way to invest in intelligence is to track intelligence errors as if they were software defects and hold intelligence creators accountable to fix them in order, one after the next, until there aren't any more. That's not the way it works. If there are errors that you absolutely must fix, then you should consider one of the other mitigation approaches discussed in this section.

Balance the Experience

If the errors are low-grade, random intelligence errors or are caused by intelligence degradation, they will be hard to solve. In these cases, you might want to rebalance the experience, making it less forceful and making the errors less costly.

If the problems are bad enough, you could consider essentially turning off an intelligent experience until you can get on top of the problem.

There are many chapters that discuss ways to balance intelligence and experience, so I won't cover them again here.

Adjust Intelligence Management Parameters

If the errors are because of degradation—that is, because new contexts are showing up quickly or old contexts are changing meaning—you might be able to address them by training and deploying new models faster.

You might also change what training data you use, for example by phasing out old training data more quickly (when contexts are changing meaning) or up-sampling new contexts in telemetry or training (which helps with changing meanings and when new contexts are appearing quickly).

These approaches are similar to investing in intelligence, but they can be more reactive. For example, when a holiday comes around, a bad weather event occurs, or a new product is launched your problem might change more-quickly than usual. An orchestrator might know this and tweak some knobs rather than waiting for intelligence creators to learn how to predict these types of events.

Implement Guardrails

Sometimes you encounter categories of mistakes that are just silly. Any human would look at the mistake and know that it couldn't possibly be right. For example:

- When the pellet griller is at 800 degrees you never want to add more fuel to the fire.
- When the user is 10 years old you never want to show them a horror movie.

In these cases, you could try to trick the intelligence creation algorithms to learn these things. You could gather focused training data. You could hound the intelligence creators. You could invest months of work…

Or you could implement a simple heuristic to override the intelligence when it is about to do something that is obviously crazy—a guardrail.

When using guardrails, make sure to:

1. Be conservative—only override obvious problems and don't get drawn into creating sophisticated intelligence by hand.
2. Revisit your decisions—by tracking the performance (and cost) of guardrails and removing or relaxing ones that become less important as intelligence improves or the problem changes.

Override Errors

Sometimes there is no way around it; your system will make expensive mistakes that can't be mitigated any other way, and you'll have to override these mistakes by hand. For example:

- You run a search engine and the top response for the query "games" is not about games.
- You run an anti-spam service and it is deleting all the mail from a legitimate business.
- You run an e-commerce site and it removed a product for "violating policy," but the product wasn't violating policy.
- You run a funny-webpage finder and it is marking the Internet's most popular joke site as not-funny.

When these mistakes are important enough, you might want to have special user experience to allow users to report problems. And you might want to have some processes around responding to these reports. For example, you might create a support group with the right tools and work-flows to examine every reported mistake within an hour, 24 hours a day, 7 days a week.

As with guardrails, make sure to use overriding sparingly and to track the quality and cost of overrides over time.

Summary

Mistakes are part of Intelligent Systems, and you should have a plan to measure mistakes and deal with them. Part of this plan is to understand what types of bad things your system can do. Be honest with yourself. Be creative in imagining problems.

In order to fix a problem, it's helpful to know what is causing the problem. Mistakes can occur when:

- A part of your Intelligent System has an outage.
- Your model is created, deployed, or interpreted incorrectly.
- Your intelligence isn't a perfect match for the problem (and it isn't).
- The problem or user-base changes.

Once you've found a problem you can mitigate it in a number of ways:

- By investing more in intelligence.
- By rebalancing the experience.
- By changing intelligent management parameters.
- By implementing guardrails.
- By overriding errors.

An active mistake mitigation plan can allow the rest of your Intelligent System to be more aggressive—and achieve more impact. Embracing mistakes, and being wise and efficient at mitigating them, is an important part of orchestrating an Intelligent System.

For Thought…

After reading this chapter, you should:

- Understand when and how mistakes put an Intelligent System at risk.
- Understand how to know if an Intelligent System is working, and to identify the common ways it might fail.
- Be able to mitigate mistakes using a collection of common approaches, and know when to use the various approaches.

You should be able to answer questions like these:

- What is the most widespread Intelligent System mistake you are aware of?
- What is the most expensive one?
- Design a system to address one of these two mistakes (the widespread one or the expensive one).
- Would it work for the other mistake? Why or why not?

CHAPTER 25

Adversaries and Abuse

Whenever you create something valuable, someone is going to try to make a buck off of it. Intelligent Systems are no different. If you spend energy, money, and time attract users, someone is going to try to make money off of those users. If you build a business that is putting pressure on a competitor, someone is going to try to make it harder for you to run that business.

These are some common ways that abuse can affect an Intelligent System:

- Abusers try to monetize your users, for example by spamming them.
- Abusers try to steal information about your system or users, to copy or sell.
- Abusers try to use your platform to host attacks on other systems.
- Abusers try to poison your system so it doesn't perform the way you want it to.

Some of these activities are illegal, but some of them aren't. And even when the activities are illegal, the global nature of the Internet makes it very hard to find the attackers and even harder to get them prosecuted.

Because of this, all successful Intelligent Systems need to be prepared to defend themselves from abuse.

This chapter explains the basics of abuse so you can understand the challenge, be ready to identify abuse when it happens to you, and have some tools to help make your Intelligent System harder to abuse.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_25

Abuse Is a Business

The first thing to know about abuse is that it is a business—a big business. The vast majority of people carrying out abuse are doing it to make money (although a bit of abuse is carried out for fun, for social justice, or to support espionage). Some of the ways abusers can make money include these:

- **Driving traffic**: By tricking your Intelligent System to show users things that the abuser wants them to see. This is essentially advertising. The abuser gets a deal with a web site, and gets paid for every user they direct from your site, to the target website. This is often called "spamming."
- **Compromising personal information**: Including social security numbers, contact information, passwords, banking information, and so on. Abusers can use this information directly or resell it for a quick profit to other abusers.
- **Compromising computers**: By tricking users to install bad things on their computers. Once they have bad things on a user's computer they can steal personal information or use the user's computer to launch further attacks. When abusers can use your Intelligent System to communicate with users, they can trick them to do all sorts of crazy things.
- **Boosting content**: By tricking your Intelligent System to behave in ways that they want. For example, an abuser might put fake reviews on a product in an e-commerce site to make the product more prominent and sell more.
- **Suppressing content**: By trying to hurt your Intelligent System or by trying to harm other users of your system. For example, an abuser might report a competitor's content as offensive.
- **Direct theft**: Of content, perhaps for resale, such as stealing digital content in an online game.

Abusers have created markets for all of these things (and more), so it is very easy for any abuser who finds a way to do things like these on your Intelligent System to turn that activity into money.

Abuse Scales

Imagine finding an activity you could do to earn a tenth of a penny. Click a button, select an option, click another button—Bam! A tenth of a penny shows up in your bank account.

It sounds pointless. You'd have to do that activity a thousand times just to make a dollar, a hundred thousand times to make a hundred dollars. What a waste of time!

But now imagine you can program a computer to do it for you, and the computer can do it a million times an hour, every hour, for the rest of eternity. This is Internet abuse. Generally Internet abuse involves an activity that is very unlikely to succeed (like tricking someone to give up a password), or is worth very little every time it does succeed (like sending traffic to a web site)—but the abuser does these-low value activities over and over and over and over. And they make good money doing it.

What this means is that you may not have an abuse problem one day. You think you're fine, but an abuser might be experimenting, trying different activities, measuring how often users fall for their scams or how much traffic they can produce off of your users, doing the math—and when they find math that is in their favor they can scale up, quickly.

It's easy for abuse to go from zero to disaster overnight.

Estimating Your Risk

Your Intelligent System will be interesting to abusers if any of the following are true:

- **It has a lot of users**: As your Intelligent System gets more popular it will have more users. This means abusers can scale their attacks further and make more money per attack. It's worth their time to experiment against your Intelligent System because if they find a way to make a tenth of a penny, they can make a lot of them.
- **Abusers can use your system to communicate with users**: Particularly if they can put a URL in their communication. The communication can be a message, an email, a web site, a picture, a review, a comment—anything. Abusers will find ways to make money off of communicating with users by tricking them and spamming them.

- **It interacts with user generated content**: And the intelligence plays any role in deciding which content to show, how to annotate or display the content, or how to order the content it does show. Influencing how users see content will influence which content they engage with—and how any associated money will flow. Abusers will try to get inside that kind of loop.
- **The mistakes it makes cost someone money**: Particularly if the costs can be directed toward specific parties. For example, when the smart toaster burns a particular brand of freezer-tart; or when your Intelligent System is putting the non-intelligent competitor out of business.
- **It does any other thing a dedicated mind can make money off of**: Think of abusers as smart hackers with dubious morals and a lot of time on their hands. Be prepared to be amazed by their ingenuity.

What an Abuse Problem Looks Like

When abusers scale an attack against your Intelligent System they will create odd usage patterns. You can often spot them in telemetry by looking for:

- Large groups of users who use your Intelligent System in degenerate ways (very focused on the parts that make them money).
- Contexts that see a spike in activity compared to normal usage.
- Contexts where the distribution of outcomes changes drastically (because abuser are sending you incorrect information).
- Patterns in user complaints and problem reports.

Abusers may try to blend in, but they will find it hard to match your legitimate user's activities, so you can usually spot their attacks if you spend the time looking for them. You can also usually spot their attacks retroactively by setting alerts for drastic changes. Traffic to a part of your system goes up by a huge amount? Someone should know. Complaints double? Someone should take a look.

Ways to Combat Abuse

If abuse does become a problem, some approaches include these:

- Add costs to your product.
- Becoming less interesting to abusers.
- Machine learning with an adversary.
- Get the adversary out of the loop.

Add Costs

You can stop abuse and make more money! Woo-hoo!

Well, charging more might scare away your legitimate customers too, so it may not be an option. But keep in mind that abuse is a business and your best way to stop abuse is to make it harder for abusers to profit.

For example, profiting from abuse will get harder if an abuser needs to:

- Pay 10 cents per account on your Intelligent System.
- Type in some squiggly characters for every review they want to leave.
- Buy a smart toaster for every attack they want to launch.

These costs are most effective when they impact abusers more than they impact legitimate users. For example, if each good user has to type in squiggly characters once, but the abuser needs to type them for every activity they do. Done right, adding cost might put abusers out of business, without legitimate users even knowing it is happening.

Becoming Less Interesting to Abusers

You could change your product to do less of the things abusers find interesting, for example:

- Removing or restricting communication channels.
- Storing less personal user information on the site and being careful about where you display it.
- Reducing the effect of user feedback on how content is presented.

These may also make your product less useful for real users, but sometimes a small tweak or two will make all the difference in breaking the ability of abusers to profit.

Machine Learning with an Adversary

You could use machine learning to identify abusive interactions and then delete them. I mean, by this point you're probably thinking: "This is a whole book about machine learning, so machine learning must be the right way to stop abuse, right?"

Unfortunately, not exactly. Machine learning is a fine tool, but abusers are very good at changing their attacks in patterns that fool machine learning. And it usually costs abusers much less to change an attack than it will cost you to chase them and change your machine learning.

You can do machine learning to combat abuse, and it will probably help, but I recommend you consider other options first and make sure you understand how machine learning can actually impact an abuser's business model before investing too much.

Get the Abuser out of the Loop

Whenever you identify abuse, you should block everything the abuser used to launch the attack, including the account, the toaster, the pellet griller, the funny web site, the sprinkler system—all of it. This will ensure abusers have to pay the most cost possible as they scale their attacks. The infrastructure they used to attack you yesterday is burned, so they need to go out and rebuild the infrastructure today.

Another option is to focus on creating your intelligence only from trusted users. Imagine you have 100,000 users who've been with you for years, using your intelligent service, producing telemetry and contexts with outcomes for training. These users are pretty safe—they aren't accounts created by abusers to launch some new attack. They are your customers. By restricting intelligence creation to "known good" users, you can often avoid abuse completely.

Summary

Whenever you create something valuable, abusers will come and try to benefit from your hard work, putting your users and your Intelligent System at risk.

The vast majority of abuse is done to make money. By understanding how abusers make money, you can control how interesting your Intelligent System is to them. Often the cheapest way to fight abuse is to make a few small tweaks in the way your system works so that abusers can't figure out how to make a reliable profit.

Abuse usually targets low-value activities that can be scaled dramatically—a tenth of a penny, one million times a day. You can often see abuse in telemetry as spikes of activity that doesn't match your regular usage patterns. You may not be able to stop it in real time doing this, but you can usually know if you are under attack.

Some practices can discourage abuse:

- Increase the cost of doing abuse.
- Change your Intelligent System to be less valuable to abusers.
- Use some machine learning (but be careful—abusers have the upper hand here).
- Trust your established users more than you trust new users, and delete all users who are involved in confirmed abuse.

For Thought…

After reading this chapter, you should:

- Know what abusers are and what they do.
- Be able to identify easy changes that will make your Intelligent System much less interesting to abusers.

You should be able to answer questions like these:

Consider your favorite Intelligent System.

- What is one simple change that would make it much more interesting to abusers?
- What is one simple change to make it less interesting to abusers?

CHAPTER 26

Approaching Your Own Intelligent System

Thank you for reading this book. I'm glad you got this far. You should now have the foundation to execute on your own Intelligent System project, knowing:

- **How to approach an Intelligent System**: What it is good for, when you need one, and how to set objectives for one.
- **How to design an intelligent experience**: One that achieves your objectives and produces data to help grow the intelligence.
- **What it takes to implement an Intelligent System**: How to execute, manage, and measure intelligence.
- **How to create intelligence**: Including many approaches, but particularly using machine learning.
- **How to orchestrate an Intelligent System**: To bring these parts together throughout its life-cycle and achieve the impact you want.

This chapter will review some key concepts and provide a checklist for approaching your own Intelligent System project.

An Intelligent System Checklist

Intelligent Systems are changing the world by closing the loop between users and intelligence. Intelligent Systems solve important problems, delight users, and help organizations achieve success.

To create an Intelligent System there are many things you'll need to do, and many decisions you'll need to make. This section collects the key ones of these into one place, with references to the chapters where you can find more detail to help.

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_26

When approaching an Intelligent System project, I recommend that you consider the following steps.

Approach the Intelligent System Project

1. Begin by making sure an Intelligent System is right for you.

Intelligent Systems are useful when you expect to change the intelligence of your product many times through its life-cycle. This is commonly needed when your problem is:

- Large.
- Open-ended.
- Time-changing.
- Intrinsically hard.

In addition, an Intelligent System works best when:

- A partial solution is viable and interesting.
- You can use data from users interacting with the system to improve it.
- You can influence a properly scoped and meaningful objective.
- The problem justifies the effort of building the Intelligent System.

Chapter 2 discusses these topics in more details.

2. Define what success looks like for your Intelligent System.

Create consensus about what your Intelligent System should do in a way that *communicates the desired outcome*, is *achievable*, and is *measurable*.

Decide what *organizational objectives* (like profit or units sold) your Intelligent System will contribute to, and what *leading indicators* you can use to get quicker feedback that you're making progress toward them. Then decide what your Intelligent System should optimize for on a day-to-day basis by identifying the *user outcomes* you want to create and the *model properties* that will be critical for success.

Come up with a plan for measuring that your Intelligent System is achieving its objectives, which might include: *telemetry*, *waiting for outcomes to become clear*, *using human judgement*, and *asking users*.

And be prepared to invest over time to keep your goals healthy. Chapter 4 contains more detail on setting objectives for Intelligent Systems.

Plan for the Intelligent Experience

3. Decide how to present your system's intelligence to users to achieve your goals.

The goal of an intelligent experience is to:

- Present intelligence to the user.
- Achieve the system's objectives.
- Minimize intelligence flaws.
- Create data to help improve the intelligence.

To achieve these, you need to *balance the experience*, by trading off between:

- The *forcefulness* of the experience.
- The *frequency* of interactions.
- The *value of success*.
- The *cost of mistakes* (both discovering and recovering).
- The *quality of the intelligence*.

Balancing an intelligent experience is a process, in which an experience changes as intelligence changes. You'll probably want to leave yourself some room to iterate. Chapter 7 talks about balancing intelligent experiences.

And Chapter 8 talks about specific modes of interaction between users and intelligence that you can use as part of this balancing, including:

- *Automation*
- *Prompting*
- *Organizing*
- *Annotating*

You'll almost certainly use some hybrid of these in your Intelligent System, so that your experience is more forceful when the intelligence is certain, and less forceful when the intelligence is unsure.

4. Plan for getting data from your experience.

An ideal intelligent experience will produce data that lets you improve the Intelligent System (and specifically the intelligence itself) over time.

To be useful for improving the Intelligent System, data should:

- Include the *context*, *user action*, and *outcomes* of the interaction.
- Provide good *coverage* over the parts of your Intelligent System and problem space.
- Represent *real interactions* with your users.
- Be *unbiased*.
- Avoid *feedback loops*.
- Have sufficient *scale* to be useful for creating intelligence.

It is best—by far—when this data is created implicitly, as a natural byproduct of users interacting with your Intelligent System. But sometimes you need to ask for help understanding how your users perceive the outcomes they receive. Ways to get explicit user feedback include these:

- Letting users provide *ratings* in the experience (such as thumbs up, thumbs down, or 1-5 stars).
- Accepting *user reports* of bad outcomes in the experience (such as a "report as spam" button).
- Creating a support tier so users can *escalate* problems to get help (call a number for help).
- Prompting users to *classify* some of their outcomes you select.

But keep in mind that users won't want to spend a lot of time and attention helping you build your product. Focus on implicit data and use explicit data sparingly. See Chapter 9 for more detailed discussion.

5. Plan to verify your intelligent experience.

You'll need to plan how to:

- Know that the experience is functioning correctly (despite all the mistakes intelligence is making).
- Know that the experience is doing its part in helping your Intelligent System achieve its objectives.

To help with this, you might want to plan for tools to inspect the experience that will be generated for any particular context, and tools to help create and capture contexts where problems are occurring.

You might also want to plan for using different versions of the system's intelligence in these tools, including the live intelligence; a snapshot of intelligence and some simple or stub intelligence.

Chapter 10 covers verifying intelligent experiences.

Plan the Intelligent System Implementation

Implementing an Intelligent System requires all the work of implementing a traditional system, plus work to enable the system's intelligence. These additional components include:

- The *intelligence runtime*.
- *Intelligence management*.
- *Telemetry* to help grow the intelligence.
- An *intelligence creation environment*.
- Intelligence *orchestration* tools.

These components provide important capabilities for improving an Intelligent System over its lifetime. Building the right implementation can make it much easier to create intelligence and to orchestrate an Intelligent System.

Chapter 11 is an overview of the components of an Intelligent System implementation.

6. Decide where your intelligence will live.

Intelligence can be any of the following:

- *Static intelligence* in the product.
- *Client-side intelligence*.
- *Server-centric intelligence*.
- *Back-end (cached) intelligence*.

Or probably some hybrid of these. You should select where your intelligence will live based on the effect it will have on:

- The *latency in updating* your intelligence.
- The *latency in executing* your intelligence.
- The *cost of operating* your Intelligent System.
- Any requirement for *offline operation*.

Chapter 13 explores issues around where intelligence can live; it will have the detail you need to make good decisions for your application.

7. Design your intelligence runtime.

The intelligence runtime is where the intelligence is executed and experience is updated. An intelligence runtime must do all of the following:

- Gather the *context* you need so your intelligence can make good decisions.
- *Extract features* from context in a way that is safe, but that supports innovation in intelligence creation.
- Deal with *model* files and updating them.
- *Execute* models on contexts (and their features).
- Take the *results* and *predictions* produced by the execution and light up the intelligent experience.

Chapter 12 discusses intelligent runtimes in detail.

8. Plan for intelligence management.

Intelligence management is the system that lets you add new intelligence to the system and turn it on for users in a safe way. This includes:

- *Ingesting intelligence* into the Intelligent System.
- *Sanity-checking intelligence* to make sure it is not obviously harmful or corrupted.
- *Combining intelligence* when there are multiple intelligence sources, including making sure they are all in sync. Chapter 21 catalogs ways of combining intelligence.

- *Lighting up intelligence* for users in a controlled fashion. That can include silent intelligence, controlled rollout, flighting (A/B testing), and support for reversions.

Chapter 14 discusses methods for intelligence management.

9. Plan for telemetry.

Telemetry is a critical link to connect usage to intelligence, so an Intelligent System can improve over time. Telemetry is used to make sure the Intelligent System is working, understand the outcomes users are getting, and gather data to grow the intelligence.

There are more things in a system than you could possibly measure, so you need to make some decisions about how to control the scale of what you collect in telemetry. But you also need to provide tools to allow intelligence creators and orchestrators to adapt as the problem and Intelligent System change over time. These tools include:

- *Sampling* what is collected in telemetry, and turning up and down the sampling rate over time.
- *Summarizing* interactions in ways that allow the right information to be collected efficiently.
- Supporting *flexible targeting* of what contexts and users to sample telemetry from.

You can learn all about telemetry in Chapter 15.

Get Ready to Create Intelligence

Intelligence is the part of the Intelligent System that makes the hard decisions, mapping from contexts to predictions, powering the intelligent experience, creating value for users and for your organization.

10. Prepare to evaluate intelligence.

With intelligence, evaluation is creation. You should be able to easily measure any intelligence to know:

- How well it *generalizes*.
- The *types of mistakes* it makes.
- Its *mistake distribution*.

To do this you will need evaluation data, hopefully from telemetry. This should be sufficient data to get statistically meaningful evaluations. It should also be *independent* from training data (which is sometimes tricky) and it should allow you to understand how your intelligence performs on important *sub-populations*.

Chapter 19 (the longest chapter in the book) discusses evaluating intelligence—it's that important.

11. Decide how you will organize your intelligence.

Organizing intelligence creation is important when working with large Intelligent Systems where you need to:

- Have multiple people *collaboratively creating intelligence*.
- *Clean up mistakes* that intelligence is making.
- *Solve the easy part the easy way* and focus more complex techniques on the hard parts of the problem.
- *Incorporate legacy intelligence* or intelligence acquired from external sources.

Here are some common ways to organize intelligence creation:

- *Decouple feature engineering*.
- Perform *multiple model searches*.
- *Chase mistakes* (which is probably the worst).
- Use *meta-models*.
- Perform *model sequencing*.
- *Partition contexts*.
- Use *overrides*.

Chapter 21 discusses all of these ways to organize intelligence creation.

12. Set up your intelligence creation process.

Set it up to use all the tools and data available to create and improve intelligence. This involves:

- Choosing the *representation* for your intelligence (as described in Chapter 17).

- Creating *simple heuristics* as a baseline.
- Using *machine learning* to produce more sophisticated intelligence when needed (as described in Chapter 20).
- Helping *understand the tradeoffs* of various implementation options.
- And *assessing and iterating*, on and on, as long as the intelligence needs to improve.

Chapter 18 gives an example of the intelligence-creation process.

Orchestrate Your Intelligent System

You need to take control of all the tools discussed in this book and "drive the race car" to achieve success day in and day out for as long as the Intelligent System is relevant. A well-orchestrated Intelligent System will:

- *Achieve its objectives* reliably over time.
- Have experience, intelligence, and objective in *balance.*
- Have *mistakes mitigated* effectively.
- *Scale* effectively over time.
- *Degrade* slowly.

Orchestration is important for Intelligent Systems when:

- *Objectives change.*
- *Users change.*
- The *problem changes.*
- *Intelligence changes.*
- The system needs to be more *efficient.*
- *Abuse* happens.

Chapter 22 introduces orchestration in detail.

13. Plan for orchestration throughout your Intelligent System's lifecycle.

This planning includes how to get started and how to invest over time. In order to orchestrate an Intelligent System, you'll need to

- *Monitor the success criteria.*
- *Inspect interactions.*
- *Balance the experience.*
- *Override intelligence.*
- *Create intelligence.*

And more. Chapter 23 describes the activities of orchestration and how to decide when and where to invest.

14. Prepare to identify and deal with mistakes.

Because with intelligence, mistakes will come.
You will need to identify why your Intelligent System is getting mistakes, including:

- *System outages.*
- *Model outages.*
- *Intelligence errors.*
- *Intelligence degradation.*

And when you do identify a source of mistakes you must decide how to react:

- *Invest in intelligence.*
- *Balance the experience.*
- *Retrain intelligence* differently.
- Implement *guardrails.*
- *Override errors.*

By embracing errors, finding them proactively, and mitigating the damage they cause, you can free up lots of space for intelligence to shine. Chapter 24 explores mistakes.

15. Get ready for abuse.

If someone can figure out how to make a buck by abusing your users or your Intelligent System, they will.

You should be prepared to identify abuse quickly if it starts to occur, by looking for:

- Groups of users with *degenerate usage*.
- Contexts that see *spikes in activity*.
- Contexts that see drastic *changes in outcomes*.
- Patterns in complaints and *problem reports*.

If abuse does happen, keep in mind that it is a business. You don't need to block abuse. You just need to make the abuser think they can't make money and they'll stop. Here are some ways you can do this:

- Add costs to users of your Intelligent System.
- Become less interesting to abusers.
- Model with an adversary.
- Remove the adversary from the loop.

Chapter 25 is about adversaries and abuse.

16. Enjoy the ride.

And that's not the end, but the beginning. Intelligent Systems can live for many years. Attract users. Achieve objectives. Create intelligence. Orchestrate the thing—and win!

Summary

Thank you for reading this book. You should now have the knowledge to approach an Intelligent System project, with confidence.

I'm excited to see what you'll accomplish!

For Thought…

After reading this chapter, you should:

- Feel great about the work you put into reading this book.
- Be prepared to participate in Intelligent System projects, confident you have the knowledge to contribute.

You should be able to answer questions like these:

- What type of Intelligent System would you like to build?
- How will it change the world?
- What's stopping you? Get going!

Index

© Geoff Hulten 2018
G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7

E

F

P, Q

R

U

V, W, X, Y, Z