Causal AI

Robert Osazuwa Ness

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Causal AI

Version 4

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *Causal AI*. You'll want to have established skills in scripting data science analyses to get the most benefit from this book. You should also know basic probability concepts such as conditional/joint probability, expectation, conditional independence, and Bayes rule. The book provides a review of these concepts.

My path to causal inference was through computational Bayesian statistics and probabilistic machine learning, techniques that, at their core, are about *simulation*. I found it easier to understand hard ideas when I could simulate them with code. If you are the same, this book is written for you.

When I entered the tech industry, I was surprised to find that causal inference practitioners weren't linking probabilistic machine learning and graphical causality. Instead, for historical reasons (tech companies had hired economists to tackle causal inference problems), causal inference was presented as a laundry list of applied statistical and econometric methods. So I started learning these techniques, and it immediately felt like a Sisyphean task. That's saying something, considering I already had a Ph.D. in statistics and had already spent years working on causal inference. If you've tried to learn causal inference and it also felt like drinking from a statistical firehose, this book is written for you.

At the same time, an interesting thing was happening in the world of machine learning; software libraries for probabilistic machine learning, especially for deep probabilistic machine learning, were gaining widespread adoption. Rather than explain the underlying statistical nuts and bolts of machine learning, these opens source tools iteratively abstracted them away under an accessible API.

That phenomenon of *commodified inference* motivated this book. The book's core thesis is that when we view a causal model as a special flavor of probability model, then we can implement the causal models using probabilistic machine learning tools. We can then use these tools' inference APIs to handle the nuts-and-bolts of the statistics, freeing us to have a code-first focus on the high-level ideas that thread through that entire laundry list of statistical methods. If you are interested in a two-birds-one-stone approach to learning causal inference *and* probabilistic machine learning, this book is written for you.

That said, this book will work through some of the elements of that laundry list, namely the statistical methods for causal effect inference. We'll focus on the high-level intuition and the API for these methods in the DoWhy library. This book is an excellent complement to books that do a more traditional statistical deep dive into those methods. This high-level view will also free us

to explore causality in the context of cutting-edge machine learning. If you're interested in the causal foundations of **representation learning**, **algorithmic fairness**, **reinforcement learning**, and **attribution**, this book is written for you.

I hope you find the book as fun and interesting to read as it was to write. Please check out www.altdeep.ai/p/causalAIbook for notebooks of the code tutorials and other book notes. Also, please post any questions, comments, or suggestions about the book in the liveBook discussion forum. Your feedback is essential in developing the best book possible.

—Robert Osazuwa Ness

brief contents

1 Introduction to causal AI

This chatper covers

- **Defining Causal AI and causal data science**
- **Describing how Causal AI is robust, explainable, and increases value**
- **Making machine learning fairer with causal analysis**
- **Extending probabilistic ML workflows and programming toolsets to causal generative models**
- **Exploring how the *commodification of inference* trend in ML empowers causal modeling**

This chapter will introduce some key motivations for learning causal AI. I'll start with what causal AI means and its importance to current data science best practices and the current state-of-the-art of machine learning. I'll also give some intuition for how algorithmic causality could unlock the next wave of AI.

Next, I'll present a concrete example of the causal modeling workflow on the MNIST image dataset, which is essentially the "hello world" of machine learning. This machine learning example gives great insight while building intuition with clear extensions to the more classical statistical data examples of most causal inference texts (we'll have several of those types of examples in this book as well).

Finally, I talk about a trend I call the *commodification of inference*. This trend motivates this book's approach of using cutting-edge machine learning frameworks like PyTorch to implement causal models and handle the statistical heavy lifting of causal inference.

1.1 What is causal AI?

Causal reasoning is a crucial element to how humans understand, explain, and make decisions about the world. Causal AI means automating causal reasoning with machine learning. Today's learning machines have superhuman prediction ability but aren't

particularly good at causal reasoning, even when we train them on obscenely large amounts of data. In this book, you will learn how to write algorithms that capture causal reasoning in the context of machine learning and automated data science.

Though humans rely heavily on causal reasoning to navigate the world, our cognitive biases make our causal inferences highly error-prone. We developed empiricism, the scientific method, and experimental statistics to address our tendencies to make errors in causal inference tasks such as finding and validating causal relations, distinguishing causality from mere correlation, and predicting the consequences of actions, decisions, and policies. Yet even empiricism still requires humans to interpret and explain *observational data* (data we observe in passing). The way we interpret causality from those different types of data is also error-prone. Causal AI attempts to use statistics, probability, and computer science to help us surpass these errors in our reasoning.

The difficulty of answering causal questions has motivated the work of millennia of philosophers, centuries of scientists, and decades of statisticians. But now, a convergence of statistical and computational advances has shifted the focus from discourse to algorithms that we can train on data and deploy to software. It is now a fascinating time to learn how to build causal AI.

1.2 Why should I or my team care about causal data science and AI?

I want to present some high-level reasons motivating the study of causal modeling. These reasons apply to researchers, independent contributors, and managers working on data science, machine learning, and other domains of data-driven automated decision-making in general.

1.2.1 Better data science

Organizations in big tech and tech-powered retail have realized the importance of causal inference and are paying big salaries to people with a causal inference skill set. The main reason is that the goal of data science - extracting actionable insights from data – is a causal task. Causal modeling helps the data scientist achieve that goal in several ways.

SIMULATED EXPERIMENTS AND CAUSAL EFFECT INFERENCE

Causal effect inference – quantifying how much a cause (e.g., a promotion) affects an effect (e.g., sales) is the most common goal of applied data science. The gold standard for causal effect inference is randomized experiments, such as an A/B test. The concepts of causal inference explain why randomized experiments work so well; randomization eliminates non-causal sources of statistical correlation.

More importantly, causal inference enables data scientists to simulate experiments and estimate causal effects from *observational data*. Most data in the world is observational data because most data is not the result of an experiment. When people say "big data," they almost always mean observational data. When a tech company boasts of training a natural language model on petabytes of Internet data, it is observational data. When we can't run a randomized experiment because of infeasibility, cost, or ethics, causal inference enables data scientists to turn to observational data to estimate causal effects.

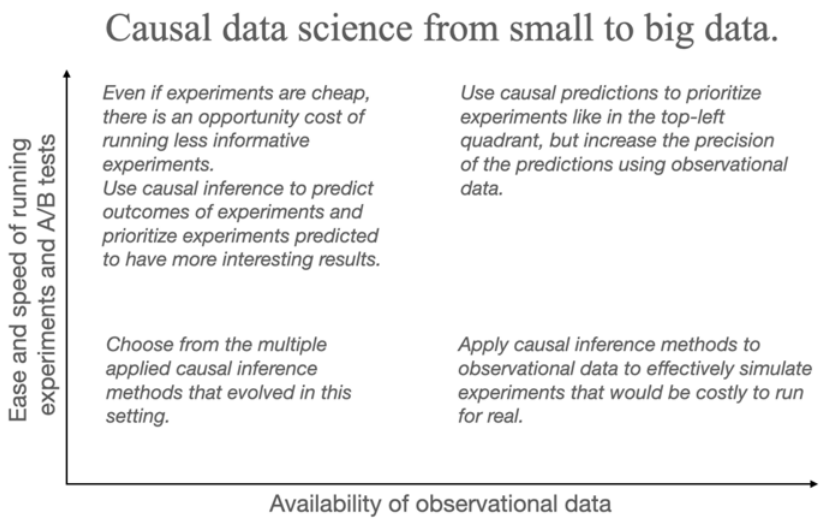


Figure 1.1 Causal data science is a valuable tool no matter how small or big your data or how easy it is to run experiments.

A common belief is that in the era of big data, it is easy to run a virtually unlimited amount of experiments. If you can run unlimited experiments, who needs causal inference? But even when you can run experiments at little *actual* cost, there are often *opportunity* costs to running experiments. For example, suppose a data scientist at an e-commerce company has a choice of one thousand experiments to run and running each one would take time and potentially sacrifice some sales or clicks in the process. Causal modeling allows that data scientist to use results from a past experiment that answers one question to simulate the results of a new experiment that would answer a slightly different question. That means she could use past data to simulate the results for each of these one thousand experiments and then prioritize the experiments by those predicted to have the most impactful results in simulation. She avoids wasting clicks and sales on less insightful experiments.

COUNTERFACTUAL DATA SCIENCE

Counterfactual questions have the form, "given what happened, what would have happened if things had been different?" Causal modeling provides a logical way to predict counterfactual outcomes. Data science that can infer counterfactuals can answer critical business questions more directly.

For example, the TV show *The Office* was the most popular series on Netflix. That posed a problem for Netflix because Netflix licenses *The Office* from Comcast, which competes with Netflix through its own streaming service. If Comcast decided to deny Netflix access to *The Office*, Netflix would be in danger of losing subscribers who don't watch much else. Thus,

Netflix had a strong incentive to find ways to encourage these subscribers to engage more with other Netflix content.

Suppose you are a data scientist at Netflix. The company introduces the show Space Force, which like *The Office,* casts actor Steve Carrell and has a similar flavor of dry humor. The hope is that *Space Force* would act as a "gateway drug" to greater engagement in other Netflix content.

The classical data science analysis divides heavy viewers of *The Office* into those who watched Space Force and who didn't, controls for other variables, and looks for a statistically significant difference in their hours of engagement in other content. This analysis would answer the question, "How much does Space Force drive engagement among *The Office* watchers?" That question is indeed relevant to the business problem of getting *The Office* watchers to engage in other content.

However, consider the counterfactual questions you could answer with data and a causal model. For example, consider those *The Office* watchers who watched *Space Force.* What is the probability that had *Space Force* not been introduced, they would have spent that time watching *The Office*? Alternatively, what is the probability that had *Space Force* not been introduced and if *The Office* were no longer available, they would have spent that time watching content from a Netflix competitor? These questions get more at the heart of the business problem.

BETTER ATTRIBUTION, CREDIT ASSIGNMENT, AND ROOT CAUSE ANALYSIS

The "attribution problem" in marketing is perhaps best articulated by a quote attributed to entrepreneur and advertising pioneer John Wanamaker:

> ***Half the money I spend on advertising is wasted; the trouble is I don't know which half.***

In other words, it is difficult to know what advertisement, promotion, or other action caused a specific customer behavior, sales number, or another key business outcome. Even in online marketing, where the data has gotten much richer and more granular than in Wanamaker's time, attribution remains a challenge. For example, a user may have clicked after seeing an ad, but was it that single ad view that led to the click, or was there a cumulative effect of all the nudges to click they received over multiple channels? Causal modeling addresses the attribution problem by using formal causal logic to answer "why" questions, such as "why did this user click?"

Attribution goes by other names in other domains, such as "credit assignment" and "root cause analysis." The core meaning is the same; we want to understand why a particular event outcome happened.

1.2.2 Causal machine learning is more robust, explainable, and valuable to the organization

There are several ways causal machine learning provides benefits to an organization engaged in engineering software reliant on machine learning. In particular, I argue that causality makes machine learning more robust, explainable, and valuable.

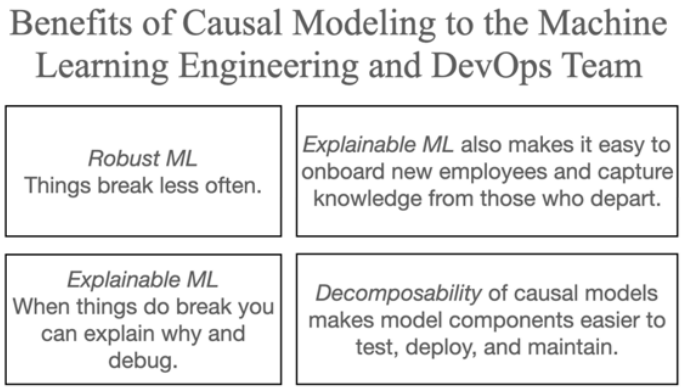


Figure 1.2 The properties of causal models offer machine learning engineering and DevOps teams several benefits relative to large machine learning models.

MORE ROBUST MACHINE LEARNING

Machine learning models lack robustness when differences between the environment where the model was trained, and the environment where the model is deployed, cause the model to break down. Examples of a lack of robustness include:

- Overfitting: The learning algorithm places too much weight on spurious statistical patterns in the training data (and "held-out" data used for parameter tuning). "Spurious" means these patterns appeared by random chance and won't likely appear in the environment where you hope to deploy the model. Techniques such as cross-validation attempt to ameliorate overfitting.
- "Underspecification"[1]: Many equivalent configurations of a model perform equivalently on test data but perform differently in the deployment environment. One sign of "underspecificaton" is sensitivity to arbitrary elements of the model's configuration, such as a random seed.
- Data drift: As time passes, the characteristics of the data in the environment where you deploy the model differ or "drift" from the characteristics of the training data.

Some claim the way to address these issues is *more* data -- that given enough data, deep learning architectures can learn anything. Only the machine learning engineers at companies like Microsoft, Google, or Meta have the luxury of virtually unlimited data and the millions of dollars of computational budget to train models on all that data. And even for them, the adage "correlation does not imply causation" remains true, even if it's a deep learning architecture learning that correlation from billions of points of data.

But these robustness problems do not condemn modern machine learning. Rather, they show we have work to do in discovering how to use causal methods to enhance the state-of-the-art. That is why Microsoft, Google, Meta, and other tech companies deploy causal

[1] D'Amour, A., Heller, K., Moldovan, D., Adlam, B., Alipanahi, B., Beutel, A., Chen, C., Deaton, J., Eisenstein, J., Hoffman, M.D. and Hormozdiari, F., 2020. Underspecification presents challenges for credibility in modern machine learning. *arXiv preprint arXiv:2011.03395*.

machine learning techniques to make their machine learning services more robust. It is also why notable deep learning researchers are pursuing research combining deep learning with causal reasoning.

Causal modeling enhances robustness by representing *invariant* causal relationships between predictors and the predicted outcome. As a simple example, if I collect data on temperature and air pressure in the tropical coastal city of Honolulu, it will look much different than the temperature and air pressure data collected in the high elevation city of Katmandu. But the physics-based causal mechanisms connecting temperature and air pressure are the same (invariant) no matter where I am on Earth.

Capturing that causal invariance helps avoid overfitting. From a causal perspective, a "spurious statistical pattern" is a pattern that isn't driven by some underlying causal relationship. For example, we might call a correlation between Nicolas Cage movie releases and drownings "spurious" because it's a random alignment of truly unrelated events. However, a causal modeler might consider a correlation between ice cream sales and robberies as originating from a shared cause of Summer heat; and thus, not spurious at all.

I had thought both were considered spurious correlations despite the former having a common cause of 'summer time warm' and the latter being truly random. In this book, we learn how causal models naturally avoid overfitting by connecting statistical patterns in the data to causal structure in the world.

Modeling causal invariance also helps avoid *underspecification*. One core contribution of formal causal inference is algorithms that tell you when a causal prediction is "identified," i.e., not "underspecified," meaning a unique answer exists given the assumptions and the data.

Finally, causal invariance helps with data drift. With the pressure/temperature example, if I train a model on the Honolulu data, I'll achieve better performance on the Katmandu data if I can manage to encode the causal invariance of that physical mechanism in the model's architecture.

MORE EXPLAINABLE MACHINE LEARNING

The behavior of modern machine learning behavior can be hard to explain. Explicability is particularly important in the context of business and engineering. If your team deploys a predictive algorithm and it behaves in a way that hurts your business, you don't want to be stuck spouting machine learning technobabble and handwaving when your boss asks you what happened. You want a concise explanation that hopefully suggests ways to avoid the problem in the future. As an engineer, you want that explanation distilled down to a concise bug report that shows in simple terms the nature of the error, what the correct output should have been, what inputs will reproduce the error, and where the code logic starts to go awry given those inputs. Armed with that explanation of the issue, you can efficiently fix the problem.

Explicability also matters to third-party users of a machine learning-based service. For example, suppose a product feature presents a user with a recommendation. That user could need to know why the feature made them a particular recommendation. An explanation is an essential element in providing recourse so the user can get better results in the future. For example, video streaming services often explain recommended content with "Because you watched X," where X is viewed content similar to the recommended content.

Instead, imagine richer content based on favored genres, actors, and themes. Instead of promoting rabbit holes of similar content, such explanations might suggest how you might explore unfamiliar content that could expand your tastes and generate more valuable recommendations in the future.

There are multiple approaches to explanation, such as analyzing node activation in neural networks. But causal models are eminently explainable because they directly encode easy-to-understand causal relationships in the modeling domain. Indeed, causality is the core of explanation; to explain an event means to provide the cause of the event[2]. Causal models provide explanations in the language of the domain you are modeling (semantic explanations) rather than in terms of the model's architecture ("nodes" and "activations" - syntactic explanations).

MORE VALUABLE MACHINE LEARNING

When a machine learning engineer trains and validates a machine learning algorithm, she deploys it to a production environment, like any other set of code. Once she does, it becomes an artifact that has value to the organization.

All else equal, a model artifact is more valuable if it has causal elements than if it does not. We've already seen how robustness and explicability contribute to value by reducing the cost of maintenance. If it is robust, it breaks less often, and if it is explainable, you can figure out how to fix it when it does.

In addition, causal invariance allows the modeler to decompose some causal models into smaller composable modules. These modules can be individually and independently tested and validated, aligning with software engineering best practices. Computer operations on these artifacts can execute separately, enabling more efficient use of modern cloud computing infrastructure. For large machine learning model artifacts, if we get additional training data or discover an issue with the initial training data, one typically has to retrain the model from scratch, which is often expensive. In contrast, we would only need to retrain the modules of the causal model that are relevant to the new data. Finally, your team can reuse components from old problems in models attacking new problems if those problems overlap.

[2] "Explain." *Merriam-Webster.com Dictionary*, Merriam-Webster, https://www.merriam-webster.com/dictionary/explain. Accessed 7 Mar. 2022.

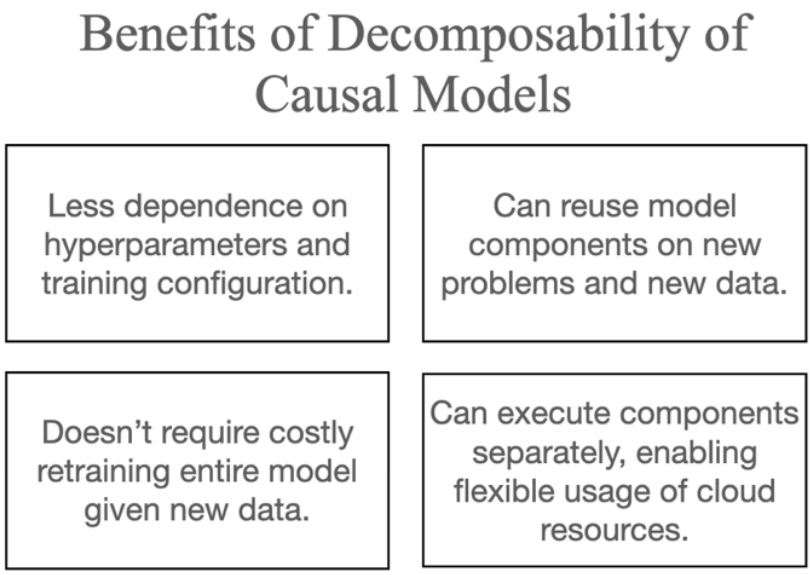


Figure 1.3 Causal models can be decomposed into components. This ability has benefits when contrasted with large machine learning artifacts.

Training large machine learning artifacts is often more art than science, relying on the intuition and experience of individual machine learning engineers. When they leave the organization to move on to their next opportunity, they take all of that intuition and experience with them. In contrast, by virtue of being decomposable and explainable, it is easier for your team to understand individual components of the model. Further, by forcing the engineer to encode their causal knowledge about a problem domain into the structure of the artifact, more of the fruits of their knowledge and experience stays with the organization as valuable intellectual property after that employee moves on.

These questions about how a modeling approach can translate into value for the organization get less attention from the statistical and machine learning research communities, who tend not to publish in business journals. However, understanding this relationship is vital to managers responsible for building business value on top of machine learning artifacts. It is also essential for independent contributors (engineers and data scientists) who want to be strategic about their careers.

1.2.3 Algorithmic Fairness

Suppose Bob applies for a business loan. A machine learning algorithm predicts that Bob would be a bad loan candidate, so Bob is rejected. Bob is a man. He got ahold of the bank's loan data, and it shows that women and gender non-binary individuals were more likely to have their loan applications approved. So was this an "unfair" outcome?

We might say the outcome is "unfair" if, for example, the algorithm made that prediction because Bob is a man. We might otherwise say if the prediction is "fair" if it was due to factors relevant to Bob's ability to pay back the loan, such as his credit history, his line of business, or his available collateral. Bob's dilemma is another example of why we'd like machine learning to be explainable, so we could analyze what factors in Bob's application lead to the algorithm's decision.

Suppose the training data came from a history of decisions from loan officers, some of whom harbored a gender prejudice that hurt men. For example, they might have read the studies that show men are more likely to default in times of financial difficulty, while women are more likely to rely on their families and social relationships for help making debt payments. Based on those studies, they decide to deduct points from their rating if the applicant is a man.

Furthermore, when that data was collected, the bank advertised the loan program on social media. When we look at the campaign results, we notice that men who responded to the ad were, on average, less qualified than the women who clicked on the ad. This discrepancy might have been because the campaign was better targeted towards women, or because the average bid price in online ad auctions was lower when the ad audience was a less qualified man.

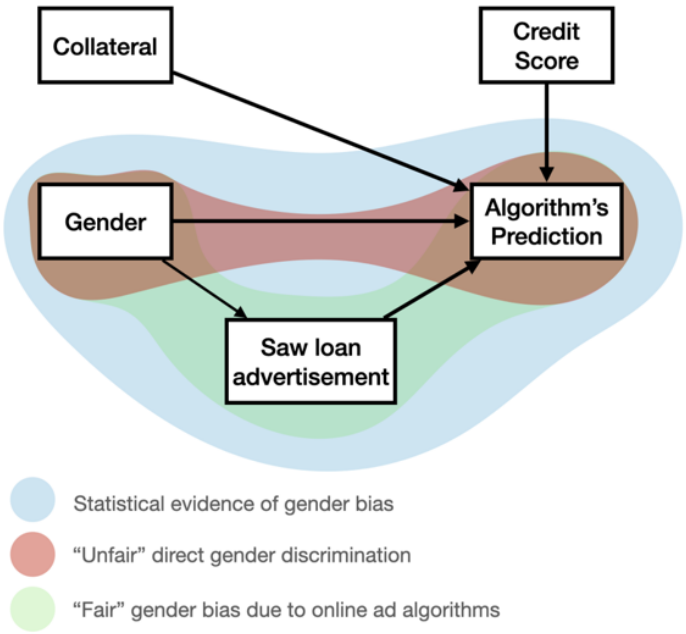


Figure 1.4 Statistical bias against gender could come from an algorithm directly penalizing gender (unfair) and indirectly through gender discrepancies in applicants attracted by advertising (fair). Causal inference can parse bias into fair and unfair sources.

Thus, we have two possible sources of statistical bias against men in the data. One source of bias is from the online ad that attracted men who were, on average, less qualified, leading to a higher rejection rate for men. The other source of statistic bias comes from the prejudice of loan officers. One of these sources of bias is arguably "fair" – it's hard to blame the bank for online ad auction pricing dynamics, and one of the sources is "unfair" – we can blame the bank for sexist loan policies. But when we only look at the training data without this causal context, all we'll see is statistical bias against men. The learning algorithm reproduced this bias when it made its decision about Bob.

One naïve solution is simply removing gender labels from the training data. But it is reasonable to assume that even if those loan officers didn't see an explicit indication of the person's gender, they could infer it from elements of the application, such as the person's name. Those sexist loan officers encode their prejudicial views in the form of a statistical correlation between those proxy variables for gender and loan outcome. The machine learning algorithm would discover this statistical pattern and use it to make predictions. As a result, you could have a situation where the algorithm would produce two different predictions for two individuals who had the same repayment risk but differed in gender, even if gender wasn't a direct input to the prediction. For these reasons, there is justified fear of how widespread deployment of machine learning algorithms could adversely impact our society by magnifying the unfair outcomes captured in our society's data.

However, causal analysis is instrumental in parsing algorithmic fairness issues. In this example, we could use causal analysis to parse that statistical bias into "unfair" bias due to sexism and bias due to "not unfair" factors like online ad market dynamics (not unfair). Ultimately, we could use causal modeling to build a model that only considered variables *causally relevant* to whether or not an individual can repay a loan.

It is important to note that causal analysis is insufficient to solve algorithmic fairness. Before applying causal analysis, all concerned parties need to agree on what is fair and what isn't. To illustrate, suppose that the social media ad campaign served the loan ad to less women than to men and gender non-binary individuals. Typically, there is more competition for the online attention of people who identify as women. Thus, an ad campaign will serve the ad less often to women because the highest bid on online ad auctions is higher when the audience is a woman, resulting in fewer women seeing the ad. The data will show a statistical bias against women due to the ad campaign, but is this result unfair? Concerned parties would need to decide how to balance the trade-off between gender fairness with the audience and pricing fairness to advertisers who "have a right" to serve their ad if they win the auction. Finding that consensus is not an easy task. But after we agree on what's fair and unfair, causal analysis can parse statistical bias into these two categories.

1.2.4 Causal reinforcement learning, representation learning, and the next AI wave

Incorporating causal logic into machine learning is already leading to new advances in machine learning. For example, much of state-of-the-art deep learning methods attempt to learn geometric representations of the objects being modeled. However, these methods struggle with learning causally meaningful representations. For example, a causally meaningful representation of an image of a cat on a table wouldn't have the cat floating in

midair if the table were removed. The challenge of building causally meaningful representations presents an opportunity for those who are among the first to forge new ground on this fascinating frontier.

Another example comes from a machine learning task called reinforcement learning. In canonical reinforcement learning, learning agents ingest large amounts of data and learn like Pavlov's dog; they learn actions that correlate positively with good outcomes and negatively with bad outcomes. Daniel Kahneman, in Nobel prize-winning work featured in his popular book *Thinking, Fast and Slow*, argues that causal *counterfactual* mental predictions define how humans make judgments and decisions. Humans take an outcome, mentally rewind to the point of decision, and imagine how things might have turned out had they made a different decision. If they imagine things might have turned out better, they experience regret and update their decision-making policies to avoid experiencing regret in the future. In other words, humans not only learn from what happened, they also learn from what *might have* happened.

Causal modeling approaches capable of learning complex causal representations and conducting counterfactual reasoning are a path to new algorithms that could automate these reasoning capabilities. Furthermore, they can do so with machine learning's uncanny ability to detect statistical nuance in data and without the baggage of human cognitive biases.

People already working with neural networks when the deep learning wave was gaining momentum did quite well career-wise. The next wave of AI is still taking shape, but it is clear it will fundamentally incorporate some representation of causality. The goal of this book is to help you ride that wave.

1.3 A machine learning-themed primer on causality

Let's motivate an AI view of causality with a popular benchmark dataset used in machine learning called MNIST. MNIST is a dataset of images of handwritten digits, each labeled with the actual digit represented in the image. Figure 1.5 illustrates multiple examples of the digits in MNIST.

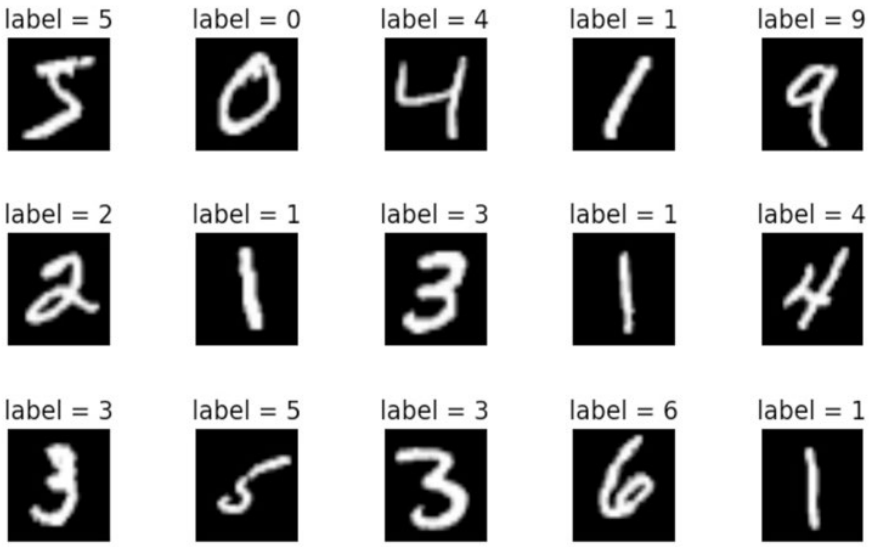


Figure 1.5 Examples of images from MNIST. Each image in the dataset is an image of a written digit, and each image is labeled with the digit it represents.

MNIST is essentially the "Hello World" of machine learning. It is primarily used to experiment with different machine learning algorithms and to compare their relative strengths. The basic prediction task is to take the matrix of pixels representing each image as input and return the correct image label as output.

1.3.1 Queries, probabilities, and statistics

Machine learning can use probability in analyses about quantities of interest. Probabilistic machine learning models learn a probabilistic representation of all the variables in that system. We can make predictions and decisions with probabilistic machine learning models using a three-step process.

1. **Pose the question**: *What is the question you want to answer?*
2. **Write down the math**: *What probability (or probability-related quantity) will answer the question given the evidence or data*?
3. **Do the statistical inference**: *What statistical analysis will give you (estimate) that quantity?*

There is more formal terminology for these steps (namely "query, estimand, and estimator") but we'll avoid the jargon for now. Instead, I'll start with a simple statistical example. Your step 1 might be "How tall are Bostonians?" For step 2, you decide that knowing the **mean** height (in probability terms, the "expected value") of everyone who lives in Boston as the quantity will answer your question. And step 3 might be to randomly select

one hundred Bostonians, and take their average height; statistical theorems guarantee that this sample average is a close estimate of the true population mean.

POSE THE QUESTION.

Now let's extend that workflow to modeling MNIST images. Suppose this we are looking at the

Figure 1.6 An ambiguous MNIST digit image.

MNIST image in Figure 1.6. In step 1, we articulate a question, such as "given this image, what is the digit represented in this image?"

WRITE DOWN THE MATH.

In step 2, we want to find some probabilistic quantity that answers the question given the evidence or data. In other words, we want to find something we can write down in probability math notation that can answer step 1. For our example with Figure 1.2, the "evidence" or "data" is the image. Is the image a 4 or a 9? Let the variable *I* represent the image and *D* represent the digit. In probability notation, we can write the probability the digit is a 4 given the image as $P(D=4|I=$  $)$, where $I=$ is shorthand I being equal to some vector representation of the image. We can compare this probability to $P(D=9|I=$ $)$, and choose the value of D that has the higher probability. Generalizing to all ten digits, the mathematical quantity we want in step 2 is:

$$\underset{d}{\operatorname{argmax}}\, P(D=d|I=$$

In plain English, this is "the value d that maximizes the probability that D equals d given the image" where d is one of the ten digits (0-9).

DO THE STATISTICAL INFERENCE.

Step 3 uses statistical analysis to assign a number to the quantity we identified in step 2. There are any number of ways we can do this. For example, we could train a deep neural network that takes in the image as an input and predicts the digit as an output; we can design the neural net to assign a probability to D=d for every value d.

1.3.2 Causality and MNIST

So how could causality feature in this analysis? Yann LeCun is a Turing Award winner (AI's Nobel prize) for his work on deep learning, and director of AI research at Meta. He is also one of the three researchers behind the creation of MNIST. He discusses the *causal* backstory of the MNIST data on his personal website yann.lecun.com:

> ***The MNIST database was constructed from NIST's Special Database 3 and Special Database 1 which contain binary images of handwritten digits. NIST originally designated SD-3 as their training set and SD-1 as their test set. However, SD-3 is much cleaner and easier to recognize than SD-1. The reason for this can be found on the fact that SD-3 was collected among Census Bureau employees, while SD-1 was collected among high-school students. Drawing sensible conclusions from learning experiments requires that the result be independent of the choice of training set and test among the complete set of samples. Therefore, it was necessary to build a new database by mixing NIST's datasets.***[3]

The authors mixed the two data sets because they argue that if one trained a machine learning model solely on digits drawn by high schoolers', it would underperform when applied to digits drawn by bureaucrats. However, in real-world settings, we want robust models that can learn in one scenario and predict in another, even when those scenarios differ. For example, we want a spam filter to keep working when the spammers switch from Nigerian princes to Bhutanese princesses. We want our self-driving cars to stop even when there is graffiti on the stop sign. Shuffling the data like a deck of cards is a luxury not easily afforded in practical real-world settings.

Causal modeling leverages knowledge about the causal mechanisms underlying how the digits are drawn that would help them generalize from high school students to bureaucrats in the training data to high schoolers in the test data. Figure 1.3 illustrates a causal graph, a popular and effective way to represent causal mechanism.

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE*, 86(11):2278-2324, November 1998.

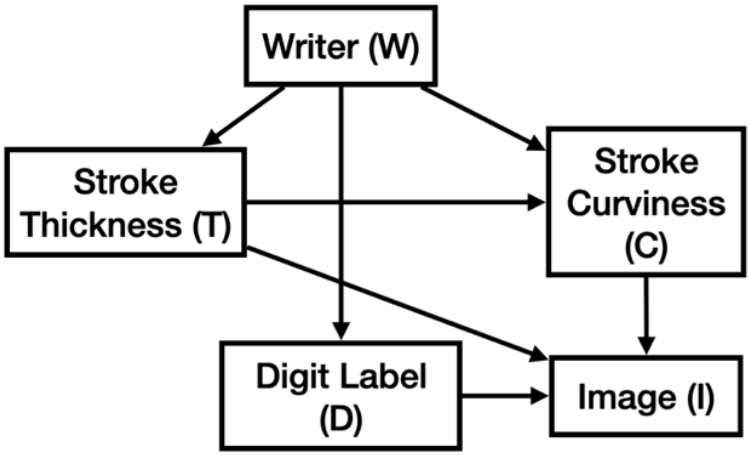


Figure 1.7 An example causal graph representing generation of MNIST images.

This particular graph imagines that the writer determines the thickness and curviness of the drawn digits, i.e., that high schoolers tend to have a different handwriting style than bureaucrats. The graph also assumes that the writer's identity is a cause of what digits they draw. Perhaps bureaucrats write more 1's, 0's, and 5's as these numbers occur more frequently in census work, while high schoolers draw other digits more often because they do more long division in math classes (this is a similar idea to how in topic models, "topics" *cause* the frequency of words in a document). A causal modeling approach would use this causal knowledge to train a predictive model that could extrapolate from the high school training data to the bureaucrat test data.

1.3.3 Causal queries, probabilities, and statistics

We answer questions like these with the same three-step analysis as above, with a bit of added causal nuance:

1. Pose the **causal** question: *What is the question you want to answer?*
2. Write down the **causal** math: *What probability (or expectation) will answer the causal question given the evidence or data*?
3. Do the statistical inference: *What statistical analysis will give you (or "estimate") that causal quantity?*

STEP 1: POSE THE CAUSAL QUESTION.

We already asked, "Given this high schooler data, what would a bureaucrat's number 4 look like?" Here are examples of some causal questions we could ask about our causal model.

- "Is gender also a cause of how the image looks?" This question type is called *causal discovery* (see chapter ?).

- Assuming that stroke thickness is a cause of the image, we might ask, "What would a 2 look like if it were as curvy as possible?" That is *intervention prediction* (chapter ?).
- "How much does the writer's identity (high schooler vs. bureaucrat) affect the look of the image?" *Causal effect estimation* (this chapter, and chapter ?).
- "Given an image, how would it have turned out differently if the stroke curviness were heavier?" *Counterfactual reasoning* (chapter ?).
- "What should the stroke curviness be to get an aesthetically ideal image?" *Causal decision-making and agent modeling* (chapter ?).
- "The image looks strange? Was it because strokes were too curvy?" *Actual causality*, *attribution*, and *explanation* (chapter ?).

STEP 2: WRITE DOWN THE CAUSAL MATH.

Causal inference theory largely focuses on step 2. In non-causal models, we rely purely on statistical correlation. But as the saying goes, correlation does not imply causation. To move from correlation to causation, we need causal assumptions. Causal graphs such as Figure 1.5 encode those assumptions in graphical form. Causal theory such as something called the *do-calculus* tells us how to turn those assumptions into the desired causal math. Much of that theory is readily implemented as graph-based algorithms in software packages.

STEP 3: DO THE STATISTICAL INFERENCE.

In the causal setting, the usual questions of statistical analysis apply, such as the trade-offs between bias and variance, scalability to large data, parallelizability, etc. This book takes the strong view that we should rely on statistical modeling and machine learning frameworks to handle step 3, and focus on honing our skills on steps 1 and 2.

1.4 Causality and the commoditization of inference

Note that steps 1 and 2 are up to us humans. We need to determine the questions we want to answer, and we need to determine what math is enough to answer our question. Indeed, in most cases, step 2 is automated by algorithms called "identification algorithms." So most of our focus goes to step 1. People who want to tackle Step 3 need varying degrees of knowledge of statistics and optimization, depending on the problem domain. However, "the commodification of inference" trend means that we can rely on modern machine learning frameworks to handle the heavy lifting of step 3.

The term "inference" is overloaded. Here, I use it as a catch-all term for the act of quantifying something from data. Examples of inference include:

- Calculating the average of or correlation between a set of measurements
- Fitting the weight parameters of a linear regression model or deep neural network.
- Calculating a p-value or a confidence interval.
- Running a Bayesian sampling algorithm.
- Generating a prediction or a forecast.

"The commodification of inference" refers to the fact that modeling frameworks increasingly automate away the statistical heavy lifting of inference tasks. For example, much of modern statistics and machine learning, including deep learning, rely on gradient-based optimization to fit model parameters or "weights." In deep learning, gradient-based

optimization at one time required modelers to specify complex derivatives of functions in the model manually. Then, we saw the emergence of differential programming frameworks such as TensorFlow and PyTorch that calculate gradients automatically. Even then, model-builders had to specify a complex set of "hyperparameters" that configure the learning algorithm. The ability to optimize hyperparameters is an essential, albeit tedious, skill in deep learning job candidates. But eventually, differential programming frameworks came to automate hyperparameter optimization – that job skill is becoming less essential as this task becomes commoditized.

We've seen similar trends in the world of probabilistic machine learning – a broad bag that includes statistical models and machine learning methods that use algorithms to derive probability distributions from data. Software for probabilistic graphical models and Bayesian models provided inference algorithms that worked automatically after one specified the model. At one time, if one wanted to use an inference algorithm Markov Chain Monte Carlo to estimate a model parameter from data, they needed a solid grasp of probability theory and be adept at writing data structures and algorithms in a low-level programming language like C. Now frameworks like Stan, PyMC, and Pyro provide an implementation of MCMC that doesn't require the user to understand anything beyond how to interpret the algorithm's results. Finally, those differential programming frameworks also allow one to use cutting-edge deep learning abstractions to do approximate probabilistic inference.

A key element of this trend is that the frameworks that enable this commoditization are universal modeling frameworks. For example, it is not as though PyTorch is specific to natural language models and TensorFlow is only used for computer vision, nor is Stan for biology while PyMC is for finance. Instead, these frameworks provide modeling abstractions that are domain-independent and provide a large amount of flexibility in how the statistical model is specified.

Most texts on causal inference focus on various statistical methods for estimating causal effects. Indeed, when I've taught workshops on causal modeling, a common reason for joining is a professional need to learn specific causal effect methods like *difference-in-differences* or *propensity score matching*. Understanding what statistical methods analysts in your problem domain use and why they are popular is indeed important, particularly if one is to maintain data pipelines built around these tools and debug them when they fail.

However, in contrast to those texts, we make a strong bet that as inference continues to get commodified by universal frameworks, those universal frameworks will handle the statistical nuts and bolts of these various causal inference methods. For example, chapter ??? illustrates the use of Microsoft Research library *DoWhy* to estimate a causal effect using several different causal effect estimation methods; *DoWhy* allows you to switch between these methods simply by changing an argument to a function or a method, much as you can toggle between models in *scikit learn*.

The assumption frees us up to focus on a high-level of causal modeling that unifies the many specific statistical methods for causal inference. Secondly, we can work with programming frameworks that allow us to specify models at that high level and rely on the commoditization of inference to do the heavy lifting for us. In this book, we specifically rely on Python-based graphical modeling framework *pgmpy*, and PyTorch-based generative

machine learning framework *Pyro*[4], which works well with causal models. Rather than slogging through each individual statistical estimation procedure without knowing how they tie together, we can pursue a global approach, using cutting-edge machine learning technology, compare procedures based on their compute time and outputs, and be in an excellent position to deep dive into individual statistical procedures if and when we need to.

1.5 Summary

- Causal AI seeks to augment statistical learning and probabilistic reasoning with causal logic.
- Causal analysis helps data scientists extract more causal insights from observational data (the vast majority of data in the world) and experimental data.
- When data scientists can't run experiments, causal analysis can simulate experiments from observational data.
- When data scientists can run experiments, data scientists can still simulate experiments and prioritizing actual experiments by interesting simulated outcomes.
- Causal analysis also helps data scientists have more business impact through algorithmic counterfactual reasoning and attribution.
- Causal analysis also makes machine learning more *robust*, *explainable*, and *valuable* to the organization. Causal models are easier for your employees and colleagues to understand and maintain, and capture employee knowledge and effort as intellectual property.
- Causal analysis is useful for formally analyzing *fairness* in predictive algorithms and for building fairer algorithms by parsing ordinary statistical bias into its causal sources.
- Types of causal inference tasks include *causal discovery*, *intervention prediction*, *causal effect estimation*, *counterfactual reasoning*, *explanation*, and *attribution*.
- The way we build and work with probabilistic machine learning models can be extended to causal generative models implemented in probabilistic machine learning tools such as PyTorch.
- The *commodification of inference* is a trend in machine learning refers to how universal modeling frameworks like PyTorch continuously automate the nuts and bolts of statistical learning and probabilistic inference.
- The *commodification of inference* reduces the need for the modeler to be an expert at the formal and statistical details of causal inference and focus on turning domain expertise into better causal models of their problem domain.

[4] At the time of writing, pgmpy and Pyro tend to be light on documentation. For pgmpy, the Python is pretty basic, so I suggest general pedagogy on probabilistic graphical models and Bayesian networks. For Pyro, most of the bugs you encounter will be Pytorch bugs, so Pytorch tutorials will help. Beyond that, look to books and tutorials on probabilistic machine learning.

2

Primer on probability modeling

This chapter covers

- **An introduction to the *pgmpy* and *pyro* libraries**
- **Probability theory essentials for causal modeling**
- **Computational probability essentials for causal modeling**
- **Statistics essentials for causal modeling**

Chapter 1 motivated learning how to code models for causal inference. It also motivated why would want to tackle causal modeling with probabilistic machine learning, which roughly refers to machine learning models that use probability to simulate data and model uncertainty. The probabilistic machine learning approach to causality makes causal modeling more intuitive and utilizes cutting-edge tools like Pytorch. This chapter introduces the concepts from probability, statistics, modeling, inference, and even philosophy that we will need to implement key ideas from causal inference and implement the probabilistic machine learning approach.

This is not a mathematically exhaustive introduction to these ideas. I focus on what is needed for the rest of this book and omit the rest. This chapter also introduces *pgmpy* and *pyro* as modeling tools.

Any data scientist seeking causal inference expertise should not neglect the practical nuances of probability, statistics, machine learning, and computer science. See the book notes at www.altdeep.ai/p/causalAIbook for recommendations for resources where you can get deeper introductions or review materials.

2.1 Primer on probability

In this section, I review the probability theory you need to work with this book. These are just few basic mathematical axioms and their logical extensions without yet adding any real-world interpretation. Let's start with the concrete idea of a simple three-sided die (these exist).

2.1.1 Random variables and probability

A ***random variable*** is a variable whose possible values are numerical outcomes of a random phenomenon. These values can be discrete or continuous. For example, the values of a discrete random variable representing a three-sided dice roll could be {1, 2, 3}. Alternatively, in 0-indexed programming language like Python, it might be better to use {0, 1, 2}. Similarly, a discrete random variable representing a coin flip could have outcomes {0, 1}. Alternatively, they could be {True, False} assuming our programming languages let us treat Booleans (True/False) as integers (1/0) when we need to do random variable math.

The standard notation is to write random variables with capitals like X, Y, and Z. For example, suppose X represents a dice roll with outcomes {1, 2, 3} and the outcome represents the number on the side of the dice. X=1 and X=2 represents the events of rolling a 1 and 2 respectively. If we want to abstract away the specific outcome with a variable, we typically use small case. For example, I would use "X=x" to represent the event "I rolled an 'x'!" where x can be any value in {1, 2, 3}.

Each outcome of a random variable has a probability value. The probability value is often called a probability mass for discrete variables and probability density for continuous variables. For discrete variables, probability values are between zero and one and summing up the probability values for each possible outcome yields 1. For continuous variables, probability densities are greater than zero, and integrating the probabilities densities over each possible outcome yields 1.

Given a random variable with outcomes {0, 1} representing a coin flip, what is the probability value assigned to 0? What about 1? At this point, we just know the two values are between zero and one and sum to one. To go beyond that, we have to talk about how to *interpret* probability. Before we do that, let's hash out a few more concepts.

2.1.2 Probability distributions and distribution functions

A probability distribution function is a function that maps the random variable outcomes to a probability value. For example, if the outcome of a coin flip is 1 (heads) and the probability value is 0.51. The distribution function maps 1 to 0.51. I stick to the standard notation P(X=x), as in P(X=1) = 0.51. For longer expressions, when the random variable is obvious, I drop the capital letter and keep the outcome, so P(X=x) becomes P(x), and P(X=1) becomes P(1).

A probability distribution is some representation the probability distributions mapping of the outcomes to the probability values. If the random variable has a finite set of discrete outcomes, it looks like a table. For example, a random variable representing outcomes {1, 2, 3} might look like Figure 2.1.

X	1	2	3
P(X)	0.45	0.30	0.25

Figure 2.1 A simple tabular representation of a discrete distribution.

In this book I adopt the common notation of P(X). In programming terms, think of this as a distribution function that hasn't yet been called on a specific outcome; the uncalled function P(X) represents all the possible arguments (outcomes) and corresponding return values (probability values).

To implement this object in *pgmpy*, we'll use the DiscreteFactor class.

Listing 2.1 Implementing a discrete distribution table in pgmpy

```
from pgmpy.factors.discrete import DiscreteFactor
dist = DiscreteFactor(
    variables=["X"],    #A
    cardinality=[3],    #B
    values=[.45, .30, .25],    #C
    state_names= {'X': ['1', '2', '3']}    #D
)
```

#A A list of the names of the variables in the factor
#B The cardinality (number of possible outcomes of each variable in the factor
#C The values each variable in the factor can take
#D Dictionary where the key is the variable name and the value is a list of the names of that variable's outcomes

```
print(dist)

+------+----------+
| X    |   phi(X) |
+======+==========+
| X(1) |   0.4500 |
+------+----------+
| X(2) |   0.3000 |
+------+----------+
| X(3) |   0.2500 |
+------+----------+
```

"phi(X)" is the probability value assigned to each outcome of X. One thing to note is that these phi values don't need to sum to one. For example, I can multiple each probability value by one hundred as follows.

```
dist = DiscreteFactor(
    variables=["X"],
    cardinality=[3],
    values=[45, 30, 25],
    state_names= {'X': ['1', '2', '3']}
)

print(dist)

+------+----------+
| X    |   phi(X) |
+======+==========+
| X(1) |  45.0000 |
+------+----------+
| X(2) |  30.0000 |
+---3--+----------+
| X(2) |  25.0000 |
+------+----------+
```

pgmpy relaxes the constraint to sum to one is because the relaxation is useful in some algorithms. We can always *normalize* to obtain proper probability values.

```
dist.normalize()    #A

print(dist)

+------+----------+
| X    |   phi(X) |
+======+==========+
| X(1) |   0.4500 |
+------+----------+
| X(2) |   0.3000 |
+------+----------+
| X(3) |   0.2500 |
+------+----------+
```

#A Normalize takes phi values for each outcome and divides them by their sum to get a probability.

2.1.3 Joint probability and conditional probability

Often, we are interested in reasoning about more than one random variable. Suppose in addition to the random variable X in Figure 2.1, there was an additional random variable Y with two outcomes {0, 1}. Then there is a joint probability distribution function that maps each combination of X and Y to a probability value.

As a table, it could look like Figure 2.2.

Y \ X	1	2	3
0	0.25	0.20	0.15
1	0.20	0.10	0.10

Figure 2.2 A simple representation of a tabular joint probability distribution.

The `DiscreteFactor` object in *s* will represent joint distributions as well.

Listing 2.2 Modeling a joint distribution in pgmpy

```
joint = DiscreteFactor(
    variables=['X', 'Y'],   #A
    cardinality=[3, 2],    #B
    values=[.25, .20, .20, .10, .15, .10],   #C
    state_names= {
        'X': ['1', '2', '3'], #D
        'Y': ['0', '1']  #D
    }
)

print(joint)

+------+------+------------+
| X    | Y    |   phi(X,Y) |
+======+======+============+
| X(1) | Y(0) |     0.2500 |
+------+------+------------+
| X(1) | Y(1) |     0.2000 |
+------+------+------------+
| X(2) | Y(0) |     0.2000 |
+------+------+------------+
| X(2) | Y(1) |     0.1000 |
+------+------+------------+
| X(3) | Y(0) |     0.1500 |
+------+------+------------+
| X(3) | Y(1) |     0.1000 |
+------+------+------------+
```

#A Now we have two variables instead of one.
#B X has 3 outcomes, Y has 2.
#C You can look at the printed output to see how the values are ordered of values.
#D Now there are two variables, so we name the outcomes for both variables.

Note that the sum of all the sum to 1. Further, when we marginalize (i.e. "sum over" or "integrate over") Y across the rows, we recover the original distribution P(X), (AKA the marginal distribution of X).

Y \ X	1	2	3
0	0.25	0.20	0.15
1	0.20	0.10	0.10
P(X)	0.45	0.30	0.25

Figure 2.3 Marginalizing over Y yields the marginal distribution of X.

The marginalize method will accomplish sum over the specified variables for us.

```
print(joint.marginalize(variables=['Y'], inplace=False))

+------+----------+
| X    |   phi(X) |
+======+==========+
| X(1) |   0.4500 |
+------+----------+
| X(2) |   0.3000 |
+------+----------+
| X(3) |   0.2500 |
+------+----------+
```

Setting the in-place argument to False gives us a new marginalized table rather than modifying the original joint distribution table.

Similarly, when we marginalize X over the columns, we get P(Y).

Y \ X	1	2	3	P(Y)
0	0.25	0.20	0.15	0.60
1	0.20	0.10	0.10	0.40

Figure 2.4 Marginalizing over X yields the marginal distribution of Y.

```
print(joint.marginalize(variables=['X'], inplace=False))

+------+----------+
| Y    |   phi(Y) |
+======+==========+
| Y(0) |   0.6000 |
+------+----------+
| Y(1) |   0.4000 |
+------+----------+
```

I'll use the notation P(X, Y) to represent joint distribution and P(X=x, Y=y) to represent an outcome probability. For shorthand I write P(x, y). For example, in Figure 2.2, P(X=1, Y=0) = P(1, 0) = 0.25. We can define a joint distribution on any number of variables; if there where three variables {X, Y, Z}, I'd write the joint distribution as P(X, Y, Z).

In this tabular representation of the joint probability distribution, the number of cells increases exponentially with each additional variable. There are some "canonical" joint probability distributions (such as the multivariate Normal distribution, I'll show more examples in section 2), though not many. For that reason, in multivariate settings, we tend to work with conditional probability distributions.

The conditional probability of Y given X is $P(Y = y|X = x) = \frac{P(X=x,Y=y)}{P(X=x)}$. Intuitively, P(Y|X=1) refers to the probability distribution for Y conditional on X being 1. For the tabular representation of the joint distribution, this is just dividing the cells in the joint probability table with marginal probability values, as in Figure 2.5.

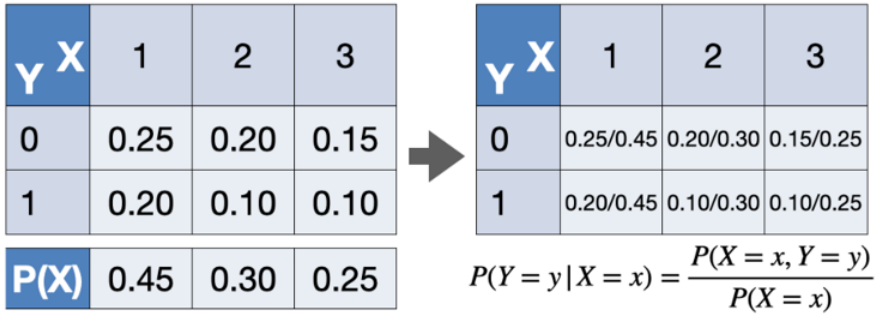


Figure 2.5 Derive the conditional probability distribution by dividing the joint distribution by a marginal distribution.

Note that now the columns on the conditional probability table in Figure 2.5 sum to 1.

The pgmpy library allows us to do this division using the "/" operator.

```
print(joint / dist)
```

```
+------+------+------------+
| X    | Y    |   phi(X,Y) |
+======+======+============+
| X(1) | Y(0) |     0.5556 |
+------+------+------------+
| X(1) | Y(1) |     0.4444 |
+------+------+------------+
| X(2) | Y(0) |     0.6667 |
+------+------+------------+
| X(2) | Y(1) |     0.3333 |
+------+------+------------+
| X(3) | Y(0) |     0.6000 |
+------+------+------------+
| X(3) | Y(1) |     0.4000 |
+------+------+------------+
```

However, the more direct way in pgmpy to specify a conditional probability distribution table is with the TabularCPD class:

```
from pgmpy.factors.discrete.CPD import TabularCPD
PYgivenX = TabularCPD(
    variable='Y',    #A
    variable_card=2,    #B
    values=[
        [.25/.45, .20/.30, .15/.25],    #C
        [.20/.45, .10/.30, .10/.25],    #C
    ],
    evidence=['X'],
    evidence_card=[3],
    state_names = {
        'X': ['1', '2', '3'],
        'Y': ['0', '1']
    })
```

```
print(PYgivenX)
```

```
+------+--------------------+--------------------+------+
| X    | X(1)               | X(2)               | X(3) |
+------+--------------------+--------------------+------+
| Y(0) | 0.5555555555555556 | 0.6666666666666667 | 0.6  |
+------+--------------------+--------------------+------+
| Y(1) | 0.4444444444444445 | 0.3333333333333333 | 0.4  |
+------+--------------------+--------------------+------+
```

#A Conditional distribution has one variable instead of DiscreteFactor's list of variables.
#B variable_card is the cardinality of Y
#C Elements of the list correspond to outcomes for Y. Elements of each list correspond to elements of X.

The argument `variable_card` is the cardinality of Y, and `evidence_card` is the cardinality of X.

Conditioning as an operation

In the phrase "conditional probability", "conditional" is an adjective. It is useful to think of "condition" as a verb. You condition a random variable like Y on another random variable X. For example, in Figure 2.5, I can condition Y on X=1, and get essential a new random variable with the same outcomes values as Y but with a probability distribution equivalent to P(Y|X=1).

For those with more programming experience, think of conditioning on X = 1 as filtering on the event X == 1, as in "what is the probability of distribution of Y when X == 1?"

Thinking of conditioning as a verb helps us understand how to implement conditional probability an operation performed on objects representing random variables. As we'll see, it also contrasts nicely with the core causal modeling concept of "intervention" where we "intervene" on a random variable.

Pyro implements conditioning as an operation with the `pyro.condition` function. We'll explore this in chapter 3.

2.1.4 The Chain Rule, the law of total probability, and Bayes Rule

From the basic axioms of probability, we can derive the chain rule of probability, the law of total probability, and Bayes rule. These laws of probability that are especially important in the context of probabilistic modeling and causal modeling, so we highlight briefly.

The chain rule of probability states that we can factorize a joint probability into the product of conditional probabilities. For example P(X, Y, Z) can be factorized as follows:

$$\mathrm{P}(\mathrm{x},\mathrm{y},\mathrm{z}) = \mathrm{P}(\mathrm{x})\mathrm{P}(\mathrm{y}|\mathrm{x})\mathrm{P}(\mathrm{z}|\mathrm{x},\mathrm{y})$$

We can factorize in any order we like. Above the ordering was X, then Y, then Z. However, Y-then-Z-then-X or Z-then-X-then-Y and other orderings are just as valid.

$$\begin{aligned} P(x,y,z) &= P(y)P(z|y)P(x|y,z) \\ &= P(y)P(z|y)P(x|z,y) \\ &= P(z)P(x|z)P(y|z,x) \end{aligned}$$

The chain rule is important from a modeling and a computational perspective. The challenge of implementing a single object that represents P(X, Y, Z) is that it would need to map each combination of possible outcomes for X, Y, and Z to a probability value. The chain rule lets us break this into three separate tasks for each factor in a factorization of P(X, Y, Z). The **law of total probability** allows you to relate marginal probability distributions (distributions of individual variables) to joint distributions: $P(y) = \sum_x P(x,y)$.

In the case where X is a continuous random variable, we integrate over X rather than summing over X. For example, suppose we have a distribution on X and Y denoted P(X, Y) and want to derive the marginal distribution of X, P(X). The law of total probability tells us to the distribution over Y. We see this in Figure 2.3 where we summed over Y in the rows to get P(X).

Finally, Bayes rule is a simple equation that follows from the chain rule and the law of total probability: $P(x|y) = \frac{P(y|x)P(x)}{P(y)}$. By itself, the rule is not particularly interesting. Bayesianism becomes interesting when it comes to its philosophy around modeling, inference, and reasoning about the world.

2.1.5 Parameters and parameter-based complexity

Suppose I wanted to implement in code an abstract representation of a probability distribution, like the tabular distribution in Figure 2.1, that I could use for different finite discrete outcomes. To start, if I were to model another three-sided die, it might have different probability values. So, what I want to keep is the basic structure as in Figure 2.6.

X	1	2	3
P(X)			

Figure 2.6 The scaffolding for a tabular probability distribution data structure.

In code, I could represent this as some object type that has a constructor that takes two arguments ρ_1 and ρ_2 ("ρ" is the Greek letter "rho").

X	1	2	3
P(X)	ρ_1	ρ_2	$1-\rho_1-\rho_2$

Figure 2.7 Adding parameters to the data structure.

The reason the third probability value is a function of the other two (instead of ρ_3) is because the probability values must sum to one. These set of two values $\{\rho_1, \rho_2\}$ are the parameters of the distribution. In programming terms, I could create some data type that represents of a table with six values. Then when I want a new distribution, I construct a new instance of this type with these five parameters as arguments.

Finally, perhaps I want my data structure to handle a prespecified number of outcomes. In that case, I'd need a parameter for the number of outcomes. Let's denote that with the Greek letter kappa κ. So, my parameterization in $\{\kappa, \rho_1, \rho_2, \ldots \rho_{\kappa-1}\}$.

In the *pgmpy* classes `DiscreteFactor` and `TabularCPD`, the ρ's where the list of values passed to the "values" argument, and the κ corresponds to the values passed to the cardinality/variable_card/evidence_card arguments.

Or suppose I wanted to write code that operated on different shapes. I could write a representation for a circle shape, and then instantiate the circle by specifying a radius parameter. I could write a representation for a rectangle shape, then instantiate the rectangle with height and width parameters. Similarly, we can instantiate probability distributions with a set of parameters.

Greeks vs. Romans

In this book, use Roman letters (A, B, and C) to refer to random variables representing objects in the modeling domain, such as "dice roll" or "GDP" and I use Greek letters for so-called "parameters." "Parameters" in this context refers to values that characterize the probability distributions of the Roman-lettered variables. This distinction between "Greeks" and "Romans" is not as important in statistics, for example a Bayesian statistician treats both Romans and Greeks as random variables. However, in causal modeling the difference matters because Romans can be causes and effects, while Greeks serve to characterize the statistical relationship between causes and effects.

2.1.6 Canonical classes of probability distribution

There are several common classes of distribution functions. For example, the tabular examples we just looked are examples from the class of Categorical distributions. The Bernoulli distribution class is a special case of the Categorical class when there are only two possible outcomes. There are other canonical distribution classes appropriate for continuous, bounded, or unbounded sets of variables. For example, the Normal (Gaussian) distribution class illustrates the famous "bell curve."

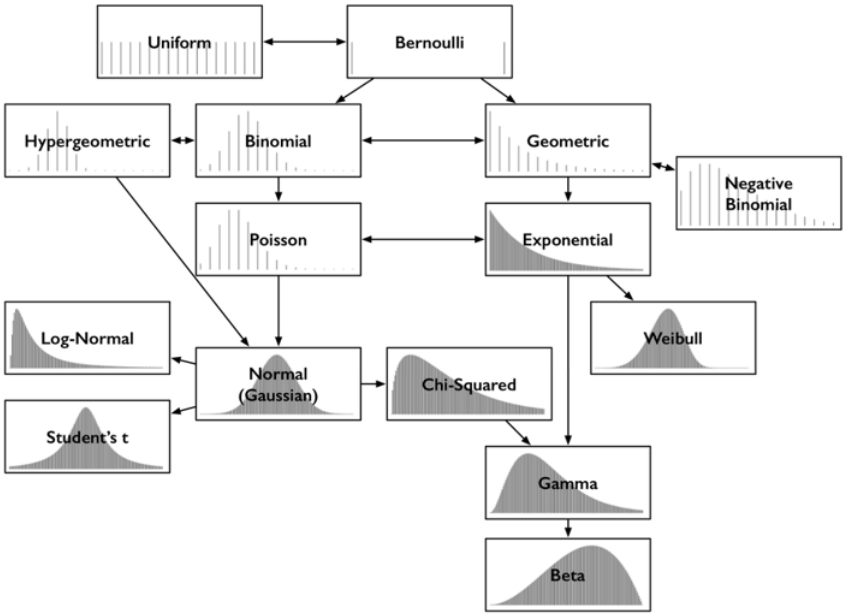


Figure 2.8 A common set of canonical probability distributions. Connected distributions are mathematically related. Light-colored distributions are discrete distributions, dark-colored distributions are continuous.

I use the term "class" (or, perhaps more ideally, "type") in the computer science sense because the distribution isn't realized until we assign our Greek-lettered parameters. For example, for a Normal (Gaussian) distribution class the probability distribution function is:

$$\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$$

Here, μ and σ are the parameters. Figure 2.8 illustrates several popular canonical distributions. The arrows between the distributions highlight adjustments to the mathematical specification that convert one distribution into another.

TYPES OF PARAMETERS

In probabilistic modeling settings it is useful to have an intuition for how to interpret canonical parameters. To that end, think of the probability in a distribution as a scarce resource that must be shared across all the possible outcomes; some outcomes may get more than others but at the end of the day, it all must sum/integrate to 1. Parameters characterize how the finite probability is distributed to the outcomes.

As an analogy, we'll use a city with a fixed population. The parameters of the city determine where that population is situated. Location parameters, such as the Normal distribution's "μ" (μ is the mean of the Normal, but not all location parameters are means) is

like pin that drops down when you search the city's name in Google Maps. The pin characterizes a precise point we might call the "city center." In some cities most of the people leave near the city center, and it gets less populous the further away from the center you go. But in other cities, other non-central parts of the city are densely population. Scale parameters, like the Normal's "σ," (σ is the standard deviation of a Normal, but not all scale parameters are standard deviation parameters). determine the spread of the population; Los Angeles has a high scale parameter. A shape parameter (and it's inverse the "rate parameter") affects shape of a distribution in a manner that is not simply shifting it (as a location parameter does) or stretching/shrinking it (as a scale parameter does). As an example, think of the skewed shape of Hong Kong, which has a densely packed collection of skyscrapers in the downtown area, while the more residential Kowloon has shorter buildings spread over a wider space.

The pyro library provides canonical distributions as modeling primitives. The pyro analog to a discrete categorical distribution table is a Categorical object:

Listing 2.3 Canonical parameters in Pyro

```
import torch
from pyro.distributions import Bernoulli, Categorical, Gamma, Normal    #A

print(Categorical(probs=torch.tensor([.45, .30, .25])))    #B
print(Normal(loc=0.0, scale=1.0))
print(Bernoulli(probs=0.4))
print(Gamma(concentration=1.0, rate=2.0))

Categorical(probs: torch.Size([3]))    #C
Normal(loc: 0.0, scale: 1.0)    #C
Bernoulli(probs: 0.4000)    #C
Gamma(concentration: 1.0, rate: 2.0)    #C
```

#A Pyro includes the commonly used canonical distributions.
#B The categorical distribution takes a list of probability values, each value corresponding to an outcome.
#C Printed representation of distribution objects

Rather than providing a probability value, the `log_prob` method will provide the natural log of the probability value because log probabilities have computational advantages over regular probabilities. Exponentiating (taking e^l where l is the log probability) converts back to the probability scale. For example, we can create a Bernoulli distribution object with a parameter value of .4.

```
bern = dist.Bernoulli(0.4)
```

That distribution assigns a .4 probability to the value 1.0. However, we can only get the log probability. But we can use the exp function in the math library to convert back to the probability scale.

```
lprob = bern.log_prob(torch.tensor(1.0))

import math
math.exp(lprob)

0.3999999887335489
```

It is close but not the same as .4 due to numerical error.

CONDITIONAL PROBABILITY WITH CANONICAL DISTRIBUTIONS

There are few canonical distributions commonly used to characterize sets of individual random variables, such as random vectors or matrices. However, we can use the chain rule and conditional independence to factor and simplify a joint probability distribution into conditional distributions we can represent with canonical distributions. For example, we could represent P(y|x, z) with the following Normal distribution:

$$\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(y-\mu(x,z))^2}$$

Where the location parameter μ(x,z) is a function of x and z. An example is the following linear function:

$$\mu(x,z) = \beta_0 + \beta_x x + \beta_z z$$

Other functions, such as neural networks, are possible as well. These β parameters are typically called weight parameters in machine learning.

2.1.7 Visualizing distributions

In probabilistic modeling and Bayesian inference settings we commonly conceptualize distributions in terms of visuals. In the discrete case, a common visualization is the bar plot. For example, we can visualize the probabilities in Figure 2.1 as the bar-plot in Figure 2.9.

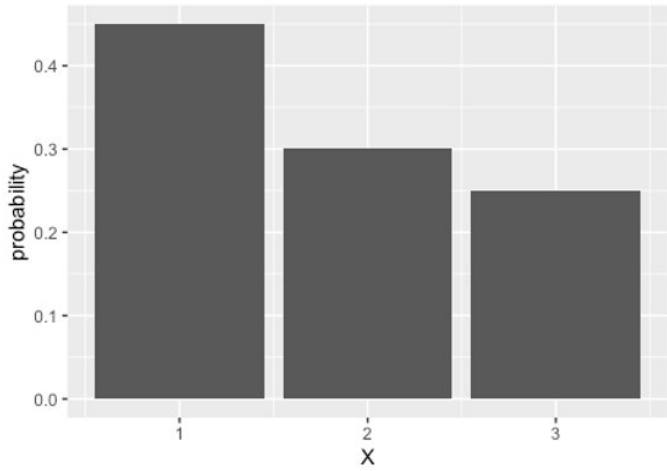


Figure 2.9 Visualization of a discrete probability distribution.

Note that is not a histogram, I'll highlight the distinction in section 2.3.

We still use visualizations when the distribution has a non-finite set of outcomes. For example, Figure 2.10 overlays two distributions functions; a discrete Poisson distribution and a continuous Normal (Gaussian) distribution (I specified the two distributions in such a way that they overlapped). The discrete Poisson has no upper bound on outcomes (it's lower bound is 0), but the probability tapers off for higher numbers, resulting in smaller and smaller bars until the bar becomes too infinitesimally small to draw. We visualize the Normal by simply drawing the probability distribution function as a curve in the figure. The Normal has no lower or upper bound, but the further way to get from the center, the smaller the probability values get. Each block in Figure 2.10 illustrates a possible visualization for the popular named distributions (though the shapes can of course vary). The continuous distributions have dark grey visualizations and the discrete distributions have light grey visualizations.

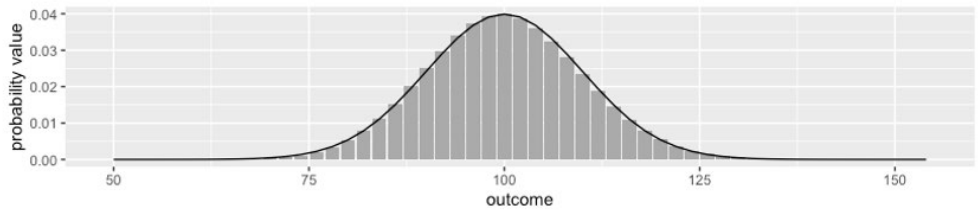


Figure 2.10 A continuous Normal distribution (solid line) approximates a discrete Poisson distribution.

Visualizing conditional probability distributions involves mapping each conditioning variable to some element in the image. For example, in Figure 2.11, X is discrete, and Y conditioned on X has a Normal distribution where the location parameter is a function of X.

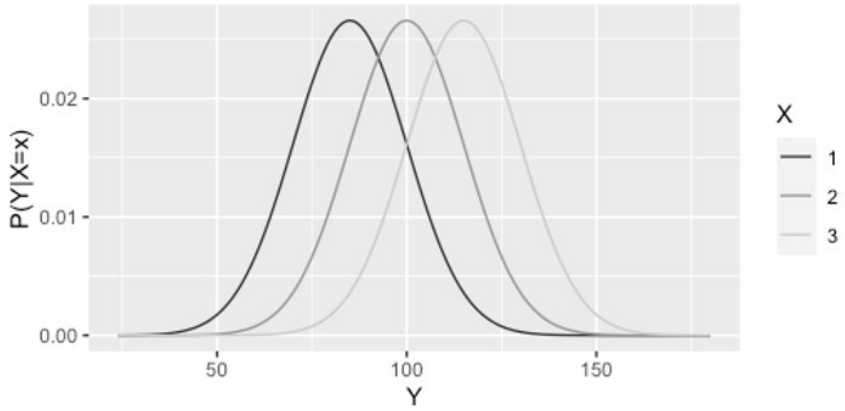


Figure 2.11 A visualization of the conditional probability distribution of continuous Y given discrete X.

Since X is discrete, it is simplest to map X to color and overlay the curves for P(Y|X=1), P(Y|X=2), and P(Y|X=3). However, if we wanted to visualize P(Y|X, Z), we'd need to map Z to an aesthetic element other than color, such as a third axis in a pseudo-3D image or rows in a grid of images. But there is only so much information we can add to with a 2D colored image. Fortunately, conditional independence helps us reduce the number of conditioning variables.

2.1.8 Independence and conditional independence

Two random variables are independent if, informally speaking, observing an outcome of one random variable does not affect the probability of outcomes for the other variable, i.e. $P(y|x) = P(y)$. We denote this as X $\perp$ Y. If two variables are not independent, then they are *dependent*, which we write as $X \not\perp Y$.

Two dependent variables can become conditionally independent given other variables. For example, X $\perp$ Y | Z means that X and Y may be dependent ($X \not\perp Y$) but they are conditionally independent given Z. In other words, if $X \not\perp Y$ and X $\perp$ Y | Z, then it is not true that P(y|x) ≠ P(y) but it is true that P(y|x, z) = P(y|x).

INDEPENDENCE IS POWERFUL TOOL FOR SIMPLIFICATION

Independence is a powerful tool for simplifying representations of probability distributions. Consider a joint probability distribution P(W, X, Y, Z). Suppose we represented it as a table. The number of cells in the table is the product of the number of possible outcomes each for W, X, Y, and Z. We could use the chain rule to do break the problem up into factors {P(W), P(X|W), P(Y|X, W), P(Z|Y, X, W)}, but the total number of

parameters across these factors doesn't change and thus the aggregate complexity is the same.

However, what if X ⊥ W? Then P(X|W) reduces to P(X). What if Z ⊥ Y|X? Then P(Z|Y, X, W) reduces to P(Z| X, W). Every time we can impose a pairwise conditional independence condition as a constraint on the joint probability distribution, we can reduce the complexity of the distribution by a large amount. Indeed, much of model building and evaluation in statistical modeling, regularization in machine learning, and deep learning techniques such as "drop-out" are either direct or implicit attempts to impose conditional independence on the joint probability distribution underlying the data.

CONDITIONAL INDEPENDENCE AND CAUSALITY

Conditional independence is fundamental to the causal modeling. Causality relationships lead to conditional independence between correlated variables. That fact allows us to learn and validate causal models against evidence of conditional independence. In chapter 4, we'll explore the relationship between conditional independence and causality in formal terms.

2.1.9 Expected value

The expected value of a random variable with a finite number of outcomes is the weighted average of all possible outcomes, where the weight is probability of that outcome. The expected value is synonymous with the mean of the random variable's distribution.

$$E(X) = \sum_{X=x} xP(x)$$

$$E(X|Y=y) = \sum_{X=x} xP(x|y)$$

In the case of a continuum of possible outcomes, the expectation is defined by integration.

$$E(X) = \int_x xP(x)dx$$

$$E(X|Y=y) = \int_x xP(x|y)dx$$

Some of causal quantities we'll be interested in calculating will be defined in terms of expectation. Those quantities only reason about the expectation, not how the expectation is calculated. It is easier to get an intuition for a problem when working with the basic arithmetic of discrete expectation rather than integral calculus of the continuous case. So, in this book, when there is a choice, I use examples with discrete random variables and discrete expectation. The causal logic in those examples all generalize to the continuous case.

In probabilistic modeling, we are frequently interested in the expectation of a function of a random variable. The expectation of a function f(X) is:

$$E\big(f(X)\big) = \sum_{X=x} f(x)P(x)$$

An example is variance, which is the expectation of the function $f(X) = (X-E(X))_2$.

There are many interesting mathematical properties of expectation. In this book, we care about the fact that conditional expectations simplify under conditional independence:

If $X \perp Y$, then $E(X|Y) = E(X)$. If $X \perp Y|Z$, then $E(X|Y,Z) = E(X|Z)$

Other than this, the most important property is the linearity of the expectation; meaning that the expectation passes through linear functions. Here are some useful reference examples of the linearity of expectation:

- For random variables X and Y: $E(X + Y) = E(X) + E(Y)$ and $E(\sum X) = \sum E(X)$
- For constants a and b: $E(aX + b) = aE(X) + b$
- If X only has outcomes 0 and 1, and $E(Y|X) = aX + b$, then $E(Y|X=1) - E(Y|X=0) = a$. (Spoiler alert: this one is important for linear regression-based causal effect inference techniques).

In several canonical distributions, the expectation maps to a function of the parameters. In some cases, such as in the Normal distribution, the location parameter is the expectation (i.e., the distribution mean, and the scale parameter is the standard deviation, the square root of the variance). But the location parameter and the expectation are not always the same. For example, the Cauchy distribution has a location parameter, but its expected value/mean is undefined.

In the next section, we introduce how to represent distributions and calculate expectations using computational methods.

2.2 Computational Probability

We need to *code* these ideas from probability to use them in our models. In the previous section, we saw how to code up a probability distribution for a three-sided die. But how do we code up *rolling* a three-sided die? How do we write code representing two dice rolls that are conditionally independent? While we're at it, how do we get a computer to do the math that calculates an expectation? How do we get a computer, where everything is deterministic, to roll dice so that the outcome is unknown beforehand?

2.2.1 The physical interpretation of probability

So, I have a three-sided die. I have some probability values assigned to each outcome on the die. What do those probability values mean? How do we interpret them?

Suppose I repeatedly rolled the die and kept a running tally of how many times I saw each outcome. Firstly, the roll is random, meaning that though I roll it the same way each time, I get varying results. That said, the physical shape of the die affects those tallies; if one face of the die is larger than the other two, that size difference will affect the count. As I repeat the roll many times, the proportion of total instances I saw a given outcome

converges to a number. Suppose I use that number for my probability value. Further, suppose I interpret that number as the "chance" of seeing that outcome each time I roll.

This idea is called physical or frequentist probability. Physical probability means imagining some repeatable physical random process that results in one outcome in a set of possible outcomes. We assign a probability value using the convergent proportion of times the outcome appears when we repeat the random process ad infinitum. We then interpret that probability as the propensity for that physical process to produce that outcome.

2.2.2 Random generation

Given this definition, we can define random generation. In random generation, an algorithm randomly chooses an outcome from a given distribution. The algorithm's choice is based on physical probability; the way it selects an outcome is such that if we ran the algorithm ad infinitum, the proportion of times it chose that outcome would equal the distribution's probability value for that outcome.

Computers are deterministic machines. If we repeatedly run a computer procedure on the same input, it will always return the same output; it cannot produce anything genuinely random. So, computers have to use deterministic algorithms to emulate random generation. These algorithms are called pseudo-random number generators. These pseudo-random number generation algorithms take some starting number called a random seed and return a deterministic series of numbers. Those algorithms mathematically guarantee that series of numbers is statistically indistinguishable from the ideal of random generation.

In notation, I write random generation as follows:

$$x \sim P(X)$$

This is reads as "x is generated from the probability distribution of X". In the case of a joint distribution:

In random generation, synonyms to "generate", include "simulate", and "sample." For example, in pgmpy the sample method in DiscreteFactor does random generation. It returns a pandas data frame.

$$x, y \sim P(X, Y)$$

Listing 2.4 Simulating from DiscreteFactor in pgmpy and Pyro

```
from pgmpy.factors.discrete import DiscreteFactor
dist = DiscreteFactor(
    variables=["X"],
    cardinality=[3],
    values=[.45, .30, .25],
    state_names= {'X': ['1', '2', '3']}
)

dist.sample(n=1)    #A
```

#A n is the number of instances you wish to generate

	X
0	1

Figure 2.12 Generating one instance from P(X) creates a Pandas DataFrame object with one row.

The n argument represents the number of samples. Note that, since this is random generation, you might not get the same output the first time you run this code.

```
joint = DiscreteFactor(
    variables=['X', 'Y'],
    cardinality=[3, 2],
    values=[.25, .20, .20, .10, .15, .10],
    state_names= {
        'X': ['1', '2', '3'],
        'Y': ['0', '1']
    }
)

joint.sample(n=1)
```

	X	Y
0	2	0

Figure 2.13 Generating one instance from P(X, Y) creates a Pandas DataFrame object with one row.

Canonical distributions in pyro also use a method with sample method.

```
import torch
from pyro.distributions import Categorical
Categorical(probs=torch.tensor([.45, .30, .25])).sample()

tensor(1.)
```

2.2.3 Coding random processes

We can write our own random processes out as code when we want to generate in a particular way. Other names for random processes in code form include stochastic processes, stochastic functions, probabilistic subroutines, and probabilistic programs.

For example, consider the joint probability distribution P(X, Y, Z). How can we randomly generate from this joint distribution? Unfortunately, software libraries don't usually provide pseudo-random generation for arbitrary joint distributions.

We get around this by applying the chain rule and, if it exists, conditional independence. So, for example, we could factorize as follows:

$$P(x, y, z) = P(z)P(x|z)P(y|x, z)$$

Suppose then that Y is conditionally independent of Z given X, then:

$$P(x, y, z) = P(z)P(x|z)P(y|x)$$

Finally, suppose we can sample from P(Z), P(X|Z), and P(Y|X) given the basic random generation functions in our software library. Then he can then use this factorization to compose an algorithm for sampling.

$$z \sim P(Z)$$
$$x \sim P(X|Z = z)$$
$$y \sim P(Y|X = x)$$

This is a random process that I can execute in code. Random processes in code form as First, I generate a Z-outcome z from P(Z). I then condition X on that z, and generate an X-outcome x. I do the same to generate a Y-outcome y. Finally, this procedure is equivalent to generate a tuple {x, y, z} from the joint distribution P(X, Y, Z).

In pgmpy, I create a random process using the class called `BayesianNetwork`.

Listing 2.5 Creating a random process in pgmpy and pyro.

```
from pgmpy.factors.discrete.CPD import TabularCPD
from pgmpy.models import BayesianNetwork
from pgmpy.sampling import BayesianModelSampling

PZ = TabularCPD(    #A
    variable='Z',    #A
    variable_card=2,    #A
    values=[[.65], [.35]],    #A
    state_names = {    #A
        'Z': ['0', '1']    #A
    })    #A

PXgivenZ = TabularCPD(    #B
    variable='X',    #B
    variable_card=2,    #B
    values=[    #B
        [.8, .6],    #B
        [.2, .4],    #B
    ],    #B
    evidence=['Z'],    #B
    evidence_card=[2],    #B
    state_names = {    #B
        'X': ['0', '1'],    #B
        'Z': ['0', '1']    #B
    })    #B

PYgivenX = TabularCPD(    #C
    variable='Y',    #C
    variable_card=3,    #C
    values=[    #C
        [.1, .8],    #C
        [.2, .1],    #C
        [.7, .1],    #C
    ],    #C
    evidence=['X'],    #C
    evidence_card=[2],    #C
    state_names = {    #C
        'Y': ['1', '2', '3'],    #C
        'X': ['0', '1']    #C
    })    #C

model = BayesianNetwork([('Z', 'X'), ('X', 'Y')])    #D
model.add_cpds(PZ, PXgivenZ, PYgivenX)    #E

generator = BayesianModelSampling(model)    #F
generator.forward_sample(size=1)    #G
```

#A P(Z)
#B P(X|Z=z)
#C P(Y|X=x)
#D Create a BayesianNetwork object. The arguments are edges of a directed graph, which we'll cover in chapter 3.
#E Add the conditional probability distributions to the model.
#F Create a `BayesianModelSampling` object from the `BayesianNetwork` object.
#G Sample from the resulting object.

This produces one row in a Pandas DataFrame.

	Z	X	Y
0	0	1	1

Figure 2.14 The `forward_sample` method simulates one instance of X, Y, and Z as a row in a Pandas DataFrame.

Implementing random processes for random generation is powerful because it allows generating from joint distributions that we can't represent in clear mathematical terms or as a single canonical distribution.

For example, while pgmpy works well with categorical distributions, pyro gives us the flexibility of working with combinations of canonical distributions. Here for example is another version of the above random process; same dependence between Z, X, and Y, but different canonical distributions:

Listing 2.6 Working with combinations of canonical distributions in Pyro

```
import torch
from pyro.distributions import Bernoulli, Poisson, Gamma

z = Gamma(7.5, 1.0).sample()    #A
x = Poisson(z).sample()    #B
y = Bernoulli(x / (5+x)).sample()    #C
```

#A Represent P(Z) with a Gamma distribution, and sample z.
#B Represent P(X|Z=z) with a Poisson distribution with location parameter z, and sample x.
#C Represent P(Y|X=x) with a Bernoulli distribution. The probability parameter is a function of y.

```
print(z, x, y)

tensor(7.1545) tensor(5.) tensor(1.)
```

Z comes from a gamma distribution, X from a Poisson with mean parameter set to z, and Y from a Bernoulli with its parameter set to a function of x.

We can add even more nuanced conditional control flow to the code:

$$z \sim P(Z)$$
$$x \sim P(X|Z = z)$$

"for i in range(x):"

$$y_i \sim P(Y_i|X = x)$$
$$y = \sum_{i}^{x} y_i$$

Here, y is still dependent on x. However, it is defined as the sum of x individual random components. In pyro, I might implement this as follows.

Listing 2.7 Random processes with nuanced control flow in Pyro

```
import torch
from pyro.distributions import Bernoulli, Poisson, Gamma
z = Gamma(7.5, 1.0).sample()
x = Poisson(z).sample()
y = torch.tensor(0.0)    #A
for i in range(int(x)):    #A
y += Bernoulli(.5).sample()    #A
```

#A y is calculated as a sum of random coin flips. y is generated from P(Y|X=x) because the number of flips depends on x.

In Pyro, best practice is to implement random processes as functions. Further, use the function `pyro.sample` to generate, rather than using the `sample` method on distribution objects. I rewrite the above `random_process` code as follows.

Listing 2.8 Using functions for random processes and pyro.sample

```
import torch
import pyro
def random_process():
    z = pyro.sample("z", Gamma(7.5, 1.0))
    x = pyro.sample("x", Poisson(z))
    y = torch.tensor(0.0)
    for i in range(int(x)):
        y += pyro.sample(f"y{i}", Bernoulli(.5))    #A
    return y
```

#A f"y(i)" creates the names "y1", "y2", ...

The first argument in `pyro.sample` is a string that assigns a name to the variable you are sampling. The reason will become apparent when we start running inference algorithms in Pyro in chapter 3.

2.2.4 Monte Carlo Simulation and Expectation

Monte Carlo algorithms use random generation to estimate expectations from a distribution of interest. The idea is simple. If you want E(X), generate multiple x's, and take the average of those x's. If you want E(f(X)), generate multiple x's apply the function f(.) to each of those x's, and take the average. Monte Carlo works even in cases when X is continuous.

In *pgmpy*, you use sample or `forward_sample` methods to generate a pandas data frame and using the mean method.

```
generated_samples = generator.forward_sample(size=100)
generated_samples['Y'].apply(int).mean()
```

In pyro, we call the random process function repeatedly. I'll do this for the above Pyro generator with a for loop that generates 100 samples:

```
generated_samples = torch.stack([random_process() for _ in range(100)])
```

This code repeatedly calls `random_process` in a Python list comprehension. Recall that pyro extends PyTorch, the value of y it returns is a tensor. So I use torch.stack to turn this list of tensors into a single tensor. Finally, I call the mean method on the tensor, to obtain the Monte Carlo estimate of E(Y).

```
generated_samples.mean()
```

```
tensor(3.7800)
```

Most things you'd want to know about a distribution can be framed in terms of some function f(X). So, for example, if you wanted to know the probability of X being greater than 10, you simply generate a bunch of x's and convert each x to 1 if it is greater than 10 and 0 otherwise. Then you take the average of the 1's and 0's, and the resulting value estimates the desired probability.

To illustrate, the following code block extends the previous block to calculate $E(Y_2)$.

```
torch.square(generated_samples).mean()
```

```
tensor(18.8000)
```

When calculating E(f(X)) for a random variable X, remember to get the Monte Carlo estimate by applying the function to the samples first, then take the average. If you apply the function to the sample average, you instead get an estimate of f(E(X)), which is almost always different (to see why, look up "Jensen's inequality").

2.2.5 Programming probabilistic inference

Suppose we implement in code a random process that generates an outcome {x, y, z} from P(X, Y, Z) as follows:

$$z \sim P(Z)$$
$$x \sim P(X|Z = z)$$
$$y \sim P(Y|X = x)$$

Further, suppose we are interested in generating from P(Z|Y=3). How might we do this? Our process can sample from P(Z), P(X|Z), and P(Y|Z); it is not clear how we go from these to P(Z|Y).

Probabilistic inference algorithms generally take an outcome-generating random process and some target distribution as inputs. Then, they return a means of generation from that target distribution. This class of algorithms is often called Bayesian inference algorithms

because the algorithms often use Bayes rule to go from P(Y|Z) to P(Z|Y). However, the connection to Bayes rule is not always explicit, so I prefer "probabilistic inference."

For example, a simple class of probabilistic inference algorithms is called accept/reject algorithms. Applying a simple accept/reject technique to generating from P(Z|Y=3) works as follows:

1. Repeatedly generate {x, y, z} using our generator for P(X, Y, Z)
2. Throw away any generated outcome where $y \neq 3$.
3. The resulting set of outcomes for Z will have distribution P(Z|Y=3)

Illustrating with pyro, I'll rewrite the above `random_process` function to return z and y. Then I'll obtain a Monte Carlo estimate of E(Z|Y=3).

Listing 2.9 Monte Carlo estimation in Pyro

```
import torch
import pyro
from pyro.distributions import Bernoulli, Gamma, Poisson
def random_process():
    z = pyro.sample("z", Gamma(7.5, 1.0))
    x = pyro.sample("x", Poisson(z))
    y = torch.tensor(0.0)
    for i in range(int(x)):
        y += pyro.sample(f"{i}", Bernoulli(.5))
    return z, y    #A

generated_samples = [random_process() for _ in range(1000)]    #B
z_mean = torch.stack([z for z, _ in generated_samples]).mean()    #C
```

#A This new version of `random_process` now returns both z and y.
#B Generate 1000 intances of z and y using a list comprehension.
#C Turn the individual z tensors into a single tensor, then calculate the Monte Carlo estimate via the `mean` method.

```
print(z_mean)
tensor(7.5032)
```

This code estimates E(Z). Since Z has a Gamma distribution, the true mean E(Z) is the shape parameter 7.5 divided by the rate parameter 1.0, which is 7.5. To estimate E(Z|Y=3), I'll filter the samples and keep only the samples where Y is 3.

```
z_given_y = torch.stack([z for z, y in generated_samples if y == 3])

print(z_given_y.mean())

tensor(6.9088)
```

That probabilistic inference algorithm works well if the outcome Y = 3 occurs frequently. If that outcome were rare, the algorithm would be inefficient: I'd have to generate many samples to get samples that meet the condition, and I'd be throwing away many samples.

Fortunately, there are various other algorithms for probabilistic inference. Unfortunately, the topic is too rich and tangential to causal modeling to explore in-depth. Nevertheless, here are a few worth mentioning for what we cover in this book.

PROBABILITY WEIGHTING METHODS

These methods generate outcomes from a joint probability distribution and then weight them according to their probability in the target distribution. We then use the weights to do weighted averaging via Monte Carlo estimation. Popular variants of this kind of inference include importance sampling and inverse probability reweighting, the latter of which is popular in causal inference and is covered in Chapter 5.

INFERENCE WITH PROBABILISTIC GRAPHICAL MODELS

Probabilistic graphical models use graphs to represent conditional independence in a joint probability distribution. The presence of a graph enables graph-based algorithms to power inference. Two well-known approaches include variable elimination and belief propagation. In Figure 2.3 and Figure 2.4, I showed you could "eliminate" a variable by summing over its columns or rows in the probability table. Variable elimination uses the graph structure to find an optimal order of variable and then eliminates them one by one until the resulting table represents the target distribution. In contrast, belief propagation is a message-passing system; the graph is used to form different "cliques" of neighboring variables. For example, If P(Z|Y=1) is the target distribution, Y=1 is a message iteratively passed back and forth between cliques. Each time a message is received, parameters in the clique are updated, and the message is passed on. Eventually, the algorithm converges, and we can derive a new distribution for Z from those updated parameters.

One of the attractive features of graph-based probabilistic inference is that users typically doesn't implement them themselves; software like *pgmpy* just does it for you. There are theoretical caveats, but they usually don't matter in practice. This feature is an example of the "commodification of inference" trend I highlighted in chapter 1. In this book, we work with causal graphical models, a special type of probabilistic graphical model that works as a causal model. That gives us the option of applying graph-based inference for causal problems.

VARIATIONAL INFERENCE

In variational inference, you write in code a new stochastic process that generates samples from an "approximating distribution" that resembles the target distribution. That stochastic process has parameters that you optimize using gradient-based techniques now common in deep learning software. The objective function of the optimization tries to minimize the difference between the approximating distribution and the target distribution.

Pyro is a probabilistic modeling language that treats variational inference as a principal inference technique. It calls the stochastic process that generates from the approximating distribution a "guide function," and a savvy Pyro programmer gets good at writing guide functions. However, it also provides a suite of tools for "automatic guide generation," another example of the commodification of inference.

HONORARY MENTIONS; MARKOV CHAIN MONTE CARLO (MCMC) AND ADVERSARIAL INFERENCE

MCMC and adversarial inference are not used in this text, but I wish to mention them briefly to give the reader a bit of context, as these are popular inference algorithms. MCMC is an inference algorithm popular amongst computational Bayesians. These are accept/reject algorithms where each newly generated outcome depends on the previous (non-rejected) generated outcome. This produces a chain of outcomes, and the distribution of outcomes in the chain eventually converges to the target distribution. Hamiltonian Monte Carlo is a popular version that doesn't require users to implement the generator, an instance of the "commodification of inference" trend from chapter 1.

Adversarial inference algorithms typically have two components, one that generates outcomes and one that rejects those outcomes if they are not good enough, according to some standard. Generative adversarial networks made adversarial inference *de rigueur* with their ability to generate uncannily realistic images. The component in generative adversarial networks that rejects is called a "discriminator" and rejects generated outcomes if it can "discriminate" a generated outcome from a *real* outcome.

2.3 Data, populations, statistics, and models

So far, we have talked about random variables and distributions. Now we move on to data and statistics. Let's start with defining some terms. You doubtless have an idea of what *data* is but let's define it in terms we've already defined in this chapter. *Data* is a set of recorded outcomes of a random variable or set of random variables. A *statistic* is anything you calculate from data. For example, when you train a neural network on training data, the learned weight parameter values are statistics, and so are model's predictions (since they depend on the training data via the weights).

The real-world causal process that generates a particular stream of data is call the *data generating process* or (DGP). A *model* is a simplified mathematical description of that process. A *statistical model* is a model with parameters tuned such that the model aligns with statistical patterns in the data.

This primer presents some of the core concepts related to data and statistics needed to make sense of this book.

2.3.1 Probability distributions as models for populations

In applied statistics, we take statistical insights from data and generalize them to a population. Consider, for example, with the MNIST digit classification problem described in chapter 1. Suppose the goal of training a classification model on MNIST data was to deploy the model in software that digitizes written text documents. In this case, the population is all the digits on all the texts the software would see in the future.

Populations are heterogeneous. Heterogeneity means, for example, that while on average a feature on a website might drive engagement among the population of uses, the feature might make some subpopulation of users less engaged. So, you would want to target the feature to the right subpopulations. Marketers call this "segmentation".

In another example, a medicine might not be much help on average for a broad population of patients, but there might be some subpopulation that experiences benefits. Targeting those subpopulations is the goal of the field of precision medicine.

In probabilistic models, we use probability distributions to model populations. The ability to model populations with probability distributions is particularly useful because we can target subpopulations by with conditional probability. For example, suppose P(E|F=True) represents the distribution of engagement numbers among all users exposed to the website feature, then P(E|F=True, G="millennial") represents the subpopulation of users exposed to the feature who are also millennials.

CANONICAL DISTRIBUTIONS AND STOCHASTIC PROCESSES AS MODELS OF POPULATIONS

If we use probability distributions to model populations, then what canonical distributions should we use for a given population. Figure 2.15 includes common distributions and the phenomena they typically model.

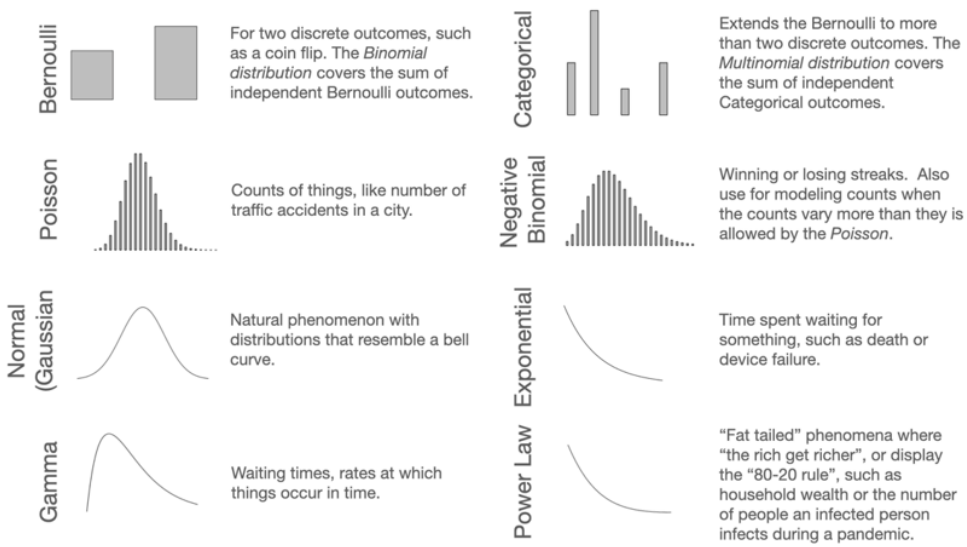


Figure 2.15 Examples of common canonical distributions and the types of phenomena and data they typically model.

These choices don't come from nowhere. The canonical distributions are themselves derived from stochastic functions. For example, the Binomial distribution is the result of a process where you do a series of coin flips. When something is the result of adding together a bunch of independent (or weakly dependent) small changes, you get a Normal distribution. The exponential distribution is the result of a waiting process where the amount of time already elapsed has no bearing on how much time you still have to weight. If you restarted the clock each time the waited-for event happened, then the number of events after a certain amount of time follows the Poisson distribution.

A useful trick in probabilistic modeling is to think of the stochastic process that created your target population. Then either choose the appropriate canonical distribution or

implement the stochastic process in code using various canonical distributions as primitives in the code logic. In this book, we'll see that this line of reasoning aligns well with causal modeling.

SAMPLING, IID, AND GENERATION

Usually, our data is not the whole population, but a small subset from the population. The act of randomly choosing an individual is called *sampling*. When the data is created by repeatedly sampling from the population, the resulting dataset is called a *random sample*. If we can view data as a *random sample*, we call that data *independent and identically distributed (IID*. That means that the selection of each individual data point is *identical* in how it was sampled, and each sampling occurred *independently* of the others, and they all were sampled from the same population distribution.

The idea of sampling and IID data illustrates the second benefit of using probability distributions to model populations. If we use a probability distribution to model a population, then we can use generation from that distribution to model sampling from a population. You can implement a stochastic process that represents the DGP by writing a stochastic process that represents the population and composing it with a process that generates data from the population process, emulating IID sampling.

In *pgmpy*, this is just as simple as generating more than one sample.

```
generator.forward_sample(size=10)
```

	Z	X	Y
0	0	1	3
1	0	0	3
2	0	0	2
3	0	0	3
4	0	0	3
5	0	0	3
6	1	0	3
7	1	0	3
8	1	0	2
9	0	0	3

Figure 2.16 A pandas DataFrame created by generating ten data points from a model in pgmpy.

The pyro approach for IID sampling is *pyro.plate*:

Listing 2.10 Generating IID samples in Pyro

```
import pyro
from pyro.distributions import Bernoulli, Poisson, Gamma

def model():
    z = pyro.sample("z", Gamma(7.5, 1.0))
    x = pyro.sample("x", Poisson(z))
    with pyro.plate('IID', 10):    #A
        y = pyro.sample("y", Bernoulli(x / (5+x)))    #B
    return y

model()
```

#A pyro.plate is a context manager for generating conditional independent samples. This instance of pyro.plate will generate 10 IID samples.
#B Calling pyro.sample to generates a single outcome y, where y is a tensor of 10 IID samples.

Using generation to model sampling is particularly useful in machine learning, because often the data is not IID. Recall the MNIST example in chapter 1, the original NIST data was not IID, one block of data came from high school students, the other from government officers. You could capture the identity of the digit writer as a variable in your stochastic process. Then the data is IID *conditional* on that variable.

DON'T MISTAKE THE MAP FOR THE TERRAIN

Consider again the MNIST data. The population for that data is quite nebulous and abstract. If that digit classification software were licensed to multiple clients, the population is a practically unending stream of digits. Generalizing to abstract populations is the common scenario in machine learning. It is for statistics as well; when R.A. Fisher, the founding father of modern statistics, was designing experiments for testing soil types on crop growth at Rothamsted Research, he was trying to figure out how to generalize to the population of future crops (with as small amount of samples as possible).

The problem with working with nebulously large populations is that it can lead to the mistake of mentally conflating populations with the probability distributions. Do not do this. Do not mistake the map for the terrain.

To illustrate, consider the following example. While writing much of this book, I was living in Silves, a town in the Portuguese Algarve with a big castle, deep history, and great hiking. Suppose I were interested in modeling the heights of Silves residents.

Officially the population of Silves is 11,000. Let's take that number as ground truth. That means there are 11,000 different height values in Silves. Suppose I physically went down to the national health center in Silves and got an actual spreadsheet of every resident's height. Then the data I have is not a random sample. It is the full population itself.

I then compute the histogram on that population, as seen in Figure 2.17.

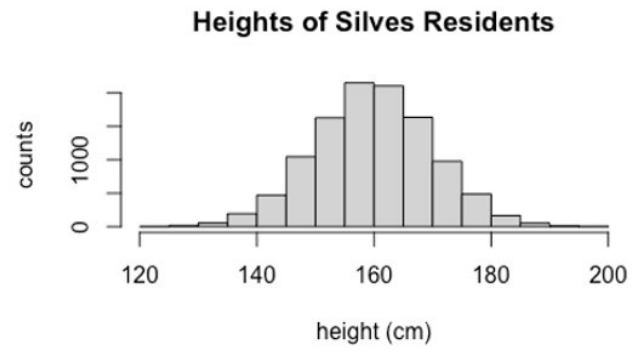


Figure 2.17 A histogram illustrating the distribution of all Silves residents.

This illustration represents the full population distribution. I can make it look more like a probability distribution by dividing by the counts by the number of people, as in Figure 2.18. 14

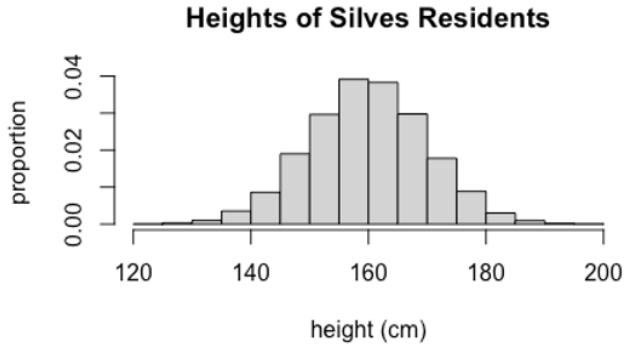


Figure 2.18 Histogram of proportions of Silves residents with given height.

Now, one might say this distribution follows the Normal (Gaussian) probability distribution, because the Normal represents evolutionary bell-shaped phenomena such as height. But that statement is not precisely true. To see this, note that all Normal distributions are defined for negative numbers (though those numbers might have an infinitesimal amount of probability density); heights can't be negative. So, what we are really doing is using the Normal distribution as a *model* of this population distribution.

In another example, Figure 2.19 shows the true distribution of the parts-of-speech in Jane Austen's novels. Note that this is not based on a sample of pages from her novels, I created this visualization from the parts-of-speech distribution of the 725 thousand words in literally *all* her six completed novels.

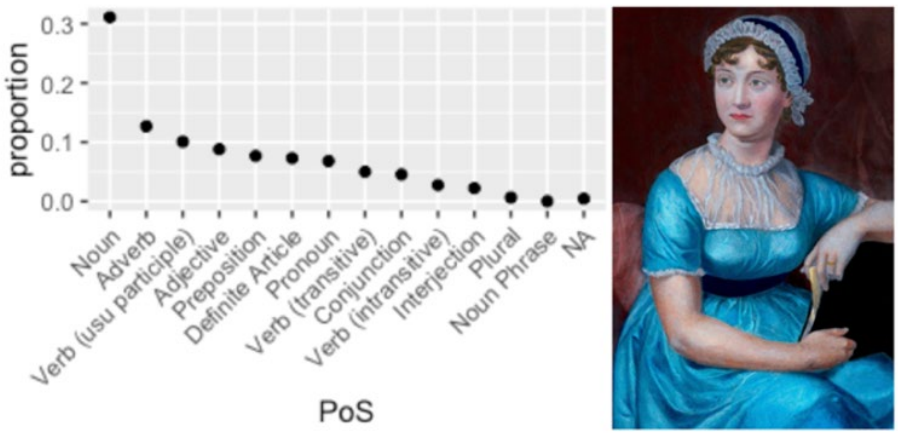


Figure 2.19 Actual distribution of word types in all of Jane Austen's novels.

It is important not to conflate the probability distribution model (the map) with the population distribution it models (the terrain). As statistician George Box famously said, "All models are wrong, some are useful." This point may seem like trivial semantics. One reason it isn't is that in the era of big data, we often can reason about the entire population instead of just samples. For example, popular online social networks have hundreds of millions and sometimes billions of users. That's a huge size, yet the entire population is just one database query away.

In causal modeling, having a bit of precision about how we think about modeling data and populations is extremely useful. Particularly in terms of modeling the data generating process.

2.3.2 From the observed data to the data generating process

In causal modeling it is important to understand how the observed data maps back to joint probability distribution, and how that joint probability distribution maps back to the data generating process. Most modelers have some level of intuition of the relationships between these entities, but in causal modeling we must be explicit. This explicit understanding is. important because while in ordinary statistical modeling you model the joint distribution (or elements of it), in causal modeling you need to model the data generating process.

FROM THE OBSERVED DATA TO THE EMPIRICAL JOINT DISTRIBUTION

Suppose we had the following dataset of five data points.

Table 2.1 A simple data set with five examples.

	jenny_throws_rock	brian_throws_rock	window_breaks
1	False	True	False
2	True	False	True
3	False	False	False
4	False	False	False
5	True	True	True

We can take counts of all the observed observable outcomes.

Table 2.2 Empirical counts of each possible outcome combination

	jenny_throws_rock	brian_throws_rock	window_breaks	counts
1	False	False	False	2
2	True	False	False	0
3	False	True	False	1
4	True	True	False	0
5	False	False	True	0
6	True	False	True	1
7	False	True	True	0
8	True	True	True	1

Dividing by the number of outcomes (5) gives us the empirical joint distribution.

Table 2.3 The empirical distribution of the data.

	jenny_throws_rock	brian_throws_rock	window_breaks	proportion
1	False	False	False	0.40
2	True	False	False	0.00
3	False	True	False	0.20
4	True	True	False	0.00
5	False	False	True	0.00
6	True	False	True	0.20
7	False	True	True	0.00
8	True	True	True	0.20

So, in the case of discrete outcomes, we go from the data to the empirical distribution using counts. In the continuous case, we could use a histogram or some other summary technique. Figure 2.20 illustrates this first step of going from the data to the joint probability distribution.

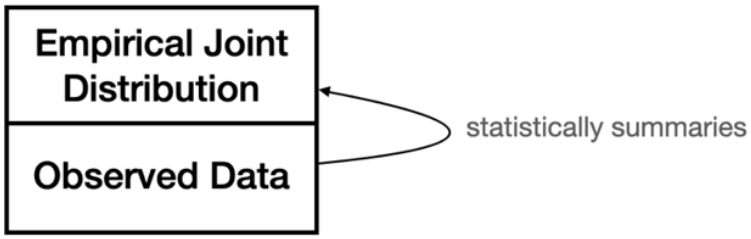


Figure 2.20 The observed data has an empirical distribution.

Importantly, the empirical joint distribution is not the actual joint distribution of the variables in the data. For example, we see that several outcomes in the empirical distribution never appeared in those five data points. Is the probability of their occurrence 0? More likely, the probabilities were greater than 0 but we didn't see those outcomes since only five points were sampled.

As an analogy, if a fair die has a 1/6 probability of rolling a 1. If you roll the die five times, you have a near $(1-1/6)_5$=40% probability of not seeing 1 in any of those rolls. If that happened to you, you wouldn't want to conclude the probability of seeing a 1 is 0. If, however, you kept rolling, the proportional of times we saw the 1 would converge to $1/6_1$.

[1] More precisely, our frequentist interpretation of probability tells us to interpret probability as the proportion of times we get a 1 when we roll ad infinitum. Despite the "ad infinitum", we don't have to roll many times before that proportion starts converging to a number (1/6).

FROM THE EMPIRICAL JOINT DISTRIBUTION TO THE OBSERVED JOINT DISTRIBUTION

Let's suppose the following is the true joint probability distribution of these observed variables.

Table 2.4 Assume this to be the true observational joint distribution.

	jenny_throws_rock	brian_throws_rock	window_breaks	probability
1	False	False	False	0.25
2	True	False	False	0.15
3	False	True	False	0.15
4	True	True	False	0.05
5	False	False	True	0.00
6	True	False	True	0.10
7	False	True	True	0.10
8	True	True	True	0.20

So, sampling from the joint observational distribution produces the empirical joint distribution.

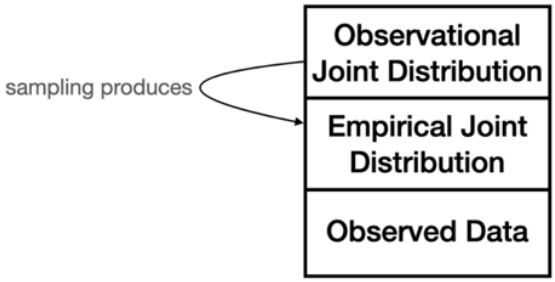


Figure 2.21 Sampling from the observational joint distribution produces the observed data and empirical distribution.

LATENT VARIABLES; FROM THE OBSERVED JOINT DISTRIBUTION TO THE FULL JOINT DISTRIBUTION

In statistical modeling, *latent variables* are variables that are not directly observed in the data but included in the statistical model. Going back to our data example, imagine there were a fourth latent variable "strength_of_impact"; it's latency is indicated by the grey shading in the following table.

Table 2.5 The values in the strength of impact column are unseen "latent" variables.

	jenny_throws_rock	brian_throws_rock	strength_of_impact	window_breaks
1	False	True	0.6	False
2	True	False	0.6	True
3	False	False	0.0	False
4	False	False	0.0	False
5	True	True	0.8	True

Latent variable models are common in disciplines ranging from machine learning, to econometrics, to bioinformatics. For example, in natural language processing, an example of a popular probabilistic latent variable model is topic models, where the observed variables represent the presence of "tokens" (e.g., words and phrases) in a document, and the latent variable represents the topic of the document (e.g., sports, politics, finance, etc.)

The latent variables are omitted from the observational joint probability distribution, because, as the name implies, the observational joint probability distribution is the joint distribution of the variables observed in the data. The joint probability distribution of both the observed and the latent variables is the full joint distribution. To go from the full joint distribution to the observational joint distribution, we marginalize over the latent variables.

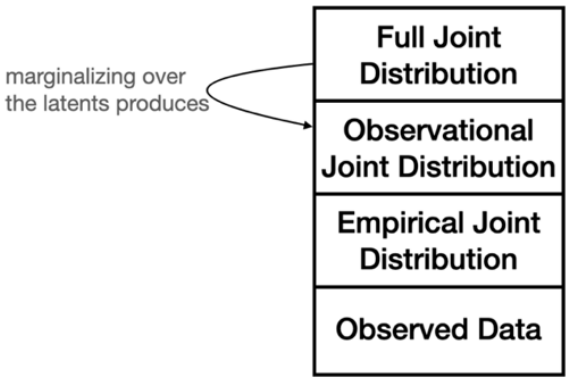


Figure 2.22 Marginalizing the full joint distribution over the latent variables produces the observational joint distribution.

FROM THE FULL JOINT DISTRIBUTION TO THE DATA GENERATING PROCESS

I wrote the actual DGP for the five data points use the following Python code[2]:

[2] Note in general the DGP is unknown, and our models are making guesses about it's structure. That said, many modern modelers are reasoning about data generated from actual code that can be viewed in a code repository. For example a data scientist working at a tech company might be analyzing data generated by software.

Listing 2.11 An example of a data generating process in code form

```
def true_dgp(jenny_inclination, brian_inclination, window_strength):    #A
    jenny_throws_rock = jenny_inclination > 0.5    #B
    brian_throws_rock = brian_inclination > 0.5    #B
    if jenny_throws_rock and brian_throws_rock:    #C
        strength_of_impact = 0.8    #C
    elif jenny_throws_rock or brian_throws_rock:    #D
        strength_of_impact = 0.6    #D
    else:    #E
        strength_of_impact = 0.0    #E
    window_breaks = window_strength < strength_of_impact    #F
    return jenny_throws_rock, brian_throws_rock, window_breaks
```

#A Input variables reflect Jenny and Brian's inclination to throw and the window strength.
#B Jenny and Brian throw the rock if so inclined.
#C If both Jenny and Brian throw the rock, the total strength of the impact is .8.
#D If either Jenny or Brian throws the rock, the total strength of the impact is .6.
#E Otherwise, no one throws and the strength of impact is 0.
#F If the strength of impact is greater than the strength of the window, the window breaks.

In this example, `jenny_inclination`, `brian_inclination`, and `window_strength` are latent variables between 0 and 1. `jenny_inclination` represents Jenny's initial inclination to throw, `brian_inclination` represents Brian's initial inclination to throw, and `window_strength` represents the strength of the window pane. These are the initial conditions that lead to one instantiation of the observed variables in the data: (`jenny_throws_ball`, `brian_throws_ball`, `window_breaks`).

I then called the true_dgp function on the following five sets of latent variables.

```
initials = [
    (0.6, 0.31, 0.83),
    (0.48, 0.53, 0.33),
    (0.66, 0.63, 0.75),
    (0.65, 0.66, 0.8),
    (0.48, 0.16, 0.27)
]
```

In other words, the following for-loop in Python is the literal sampling process producing the five data points.

```
data_points = []
for jenny_inclination, brian_inclination, window_strength in initials:
data_points.append(
    true_dgp(
        jenny_inclination, brian_inclination, window_strength
    )
)
```

So, the DGP is the causal process that generated the data. Note the narrative element that is utterly missing from the full joint probability distribution; Jenny and Brian throw a rock at a window if they are so inclined, and if they hit the window, the window will break depending on if one or both of them threw rocks and the strength of the window. The DGP

entails the full joint probability distribution. In other words, the joint probability distribution is a consequence of the DGP based on *how* it generates data.

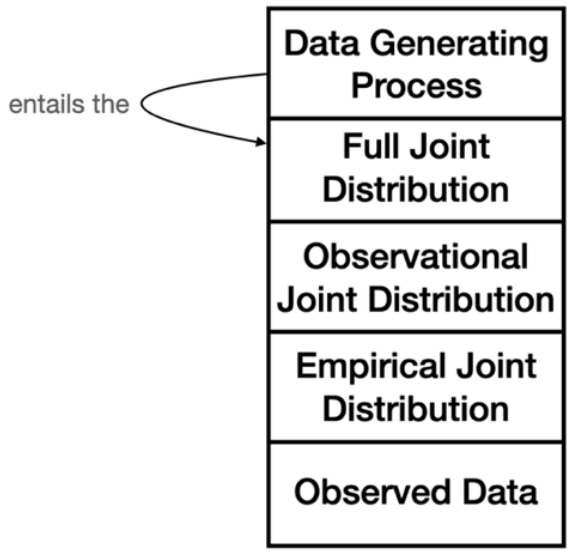


Figure 2.23 The data generating process entails the full joint distribution.

In summary, the data generating process entails the full joint distribution. Marginalizing over the full joint produces the observational joint distribution. Sampling from that distribution produces the observed data and the corresponding empirical joint distribution. There is a many-to-one relationship as we move down this hierarchy that has implications to causal modeling and inference.

MANY-TO-ONE RELATIONSHIPS DOWN THE HIERARCHY

As we move down from DGP to full joint to observational joint to empirical distribution to data, there is a many-to-one relationship from the preceding level to the subsequent level:

Figure 2.24 There is a many-to-one relationship as we move down the hierarchy. In summary, there are multiple data generating processes consistent with the observed data.

- **There could be multiple empirical joint distributions consistent with the observational joint distribution**. How we construct the empirical joint depends on statistical choices. How many bins should be in the histogram? Should we use raw proportions in the probability table? Or perhaps we should use additive smoothing to avoid zeroes when the sample size is small?
- **There could be multiple empirical joint distributions consistent with the observational joint distribution**. If we sample 5 points, then sample 5 more, we'll get different datasets and thus different empirical distributions.
- **There could be multiple full joint distributions consistent with one observational joint distribution**. The difference between the two distributions is the latent variables. But what if we have difference choices for the sets of latent variables? For example, if our observation distribution is P(X, Y), then the full joint would be P(X, Y, Z, W) if our set of latent variables is {Z, W}, or P(X, Y, Z, V) if our set of latent variables is {Z, V}.
- **There could be multiple DGP's consistent with one full joint probability distribution**: For a given joint probability distribution, there are possibly multiple DGPs that are consistent with that joint probability distribution. Suppose in our window-breaking example, Jenny had a friend Isabelle who sometimes egged Jenny on to throw the ball and sometimes did not, affecting Jenny's inclination to throw. This DGP is different from the original, but the relationship between the latent variable of Isabell's peer pressure and Jenny's inclination to throw could be such that this new DGP entailed exactly the same joint probability distribution. As a more trivial example, suppose we looked at the distribution of a single variable corresponding to the sum of the roll of three dice. The data generating process is rolling three dice then summing them together. Two DGPs could differ in terms of the order of summing the dice; e.g., (first + second) + third or (first + third) + second or (second + third) + first. These would all yield the same distribution.

Those last two many-to-one relationships is fundamental to the concept of *causal identifiability,* the core reason why causal inference is hard. This concept is the reason "correlation does not imply causation" as the phrase goes.

2.3.3 Statistical tests for independence

Causality imposes independence and conditional independence on variables. So, we rely on statistical tests for conditional independence to build and validate causal models.

Suppose X and Y are independent, or X and Y are conditionally independent given Z. If we have data observing X, Y and Z, then we can run a statistical test for independence. This is a statistical procedure that returns a statistic, namely a p-value, that quantifies the statistical evidence of dependence/independence.

Just as evidence of a murder is not the same as truth of the occurrence of a murder, statistical evidence of independence between two variables is not the same as the true or false fact of independence between variables. For example, given independence is true, the strength of the statistical evidence can vary on several factors, such as how much data there is. And it is always possible to make false conclusions from these tests.

Remember the if X and Y are independence, then P(Y|X) is equivalent to P(Y). In predictive terms, that means X has no predictive power on Y. So, if you can't use classical statistical tests (e.g., X and Y are vectors) then you can try training a predictive model and subjectively evaluate how well the model predicts.

2.3.4 Statistical estimation of parameters

Given observational data, we can *estimate* (in machine learning parlance, "train") the parameters of a model. In general, statistical modeling and machine learning, the goal of parameter estimation is modeling the observational or joint probability distribution. In causal modeling, the objective is modeling the DGP. The distinction is important for making good causal inferences. Moreover, in some cases we can use certain causality-related constraints to help us in parameter estimation.

ESTIMATING BY MAXIMIZING LIKELIHOOD

In informal terms and in the context of parameter estimation*, likelihood* is the probability of having observed the observed data given a value of the parameter vector. Maximizing likelihood means choosing a value for the parameter vector that maximizes likelihood. Usually, we work with maximizing the log of the likelihood instead of likelihood directly because its mathematically and computationally easier to do so; the value that maximizes likelihood is the same as the value that maximizes likelihood. In some models, such as linear regression, the maximum likelihood estimate has a mathematical solution. In general, we must find the solution using optimization techniques. In some models, such as neural networks, it is infeasible to find the value that maximizes likelihood, so you settle for a candidate that has high likelihood.

ESTIMATING BY MINIMIZING OTHER LOSS FUNCTIONS AND REGULARIZATION

In machine learning, there are a variety of loss functions for estimating parameters. Maximizing likelihood is a special case of minimizing a loss function, namely the negative log-likelihood loss function.

Regularization is the practice of adding additional elements to the loss function that steer the optimization towards better parameter values. For example, L2 regularization adds a value proportional to the sum of the square of the parameter values to the loss. Since a small increase in value leads to a larger increase in the square of the value, L2 regularization helps avoid exceedingly larger parameter estimates.

BAYESIAN ESTIMATION

Bayesian estimation treats parameters as random variables and tries to model the conditional distribution of the parameters (typically called the *posterior* distribution) given the observed variables in the data. It does so by putting a "prior probability distribution" on the parameters. The prior distribution has its own parameters called "hyperparameters" that the modeler has to specify. When there are latent variables in the model, Bayesian inference targets the joint distribution of the parameters and the latent variables conditional on the observed variables.

One of the main advantages of Bayesian estimation is that rather than getting a point value for the parameters, you get a representation of conditional probability distribution, such as samples. That probability distribution represents uncertainty about the parameter values, and you can incorporate that uncertainty into predictions or other inferences you make from the model.

According to Bayesian philosophy, the prior should capture the modeler's subjective beliefs or uncertainty about the true value of the parameters. I consider myself a Bayesian statistician, and I can attest that using priors to quantify uncertainty or encode subjective beliefs is no easy feat in general. Most of the time modelers choose priors that are commonly used by other modelers, or that have tractable mathematical or computational quantities.

STATISTICAL AND COMPUTATIONAL ATTRIBUTES OF AN ESTIMATOR

Given the many ways of estimating a parameter, we look for ways to compare the quality of estimation methods. Statisticians care about the bias and consistency of an estimator. An estimator is a random variable because it comes from data (and data has a distribution). So, an estimator has a distribution. An estimator is unbiased if the parameter it is estimating is the mean of that distribution. In practice, the consistency of the estimator is more important than whether it is unbiased. Consistency means that the more data you have, the closer the estimate is to the parameter.

Computer scientists know getting an estimator (or any algorithm) to work with "more data" is easier said than done. They care about the computational qualities of an estimator in relation to the size of the data. Does the estimator scale with the data? Is it parallelizable? A consistent estimator may converge, but when its running on my iPhone app will it converge in milliseconds and not eat up my battery's charge in the process?

This book decouples understanding causal logic from the statistical and computational properties of estimators of causal parameters. We will focus on the causal logic and rely on libraries like *DoWhy* to make the statistical and computational contrasts easy to do. There will be key exceptions when the causal logic and statistical logic are intertwined, such as with instrumental variables. I'll call this out when it occurs.

GOODNESS-OF-FIT VS CROSS VALIDATION

When you estimated the parameters, you can calculate various statistics to tell you how well you've done. One class of statistics are called goodness-of-fit statistics. Statisticians define goodness-of-fit as statistics that quantify how well the model fits the data used to train the model. Here's another definition, goodness-of-fit statistics tell you how well your model pretends to be the DGP for the data you used to train your model. But as we saw, there are multiple DGPs possible for a given data set.

Cross validation statistics generally see how well your model predicts data it was not trained on. It is possible to have a model with a decent goodness-of-fit relative to other models, but still predict poorly. Machine learning is usually concerned with the task of prediction, and so favors cross validation. However, note that a model can be a good predictor and provide completely bogus causal inferences.

2.4 Determinism and Subjective Probability

This section will venture into the philosophical underpinnings we'll need for probabilistic causal modeling. The first idea is to view the data generating process as deterministic. The second idea is to view the probability in our models of the data generating process as subjective.

2.4.1 Determinism

Note that the code for the rock-throwing data generating process is entirely deterministic; given the initial conditions, the output is certain. Consider our definition of physical probability again. If I throw a die, why is the outcome random?

If I had a superhuman level of dexterity, perception, and mental processing power, I could mentally calculate the roll's physics and know the outcome with certainty.

This philosophical idea of determinism essentially says that the data-generating process is deterministic. 18th-century French scholar Pierre-Simon Laplace explained determinism with a thought experiment called Laplace's demon. Laplace imagined some entity (the demon) that knew every atom's precise location and momentum in the universe. With that knowledge, that entity would know the future state of the universe with complete deterministic certainty because it could calculate them from the laws of (Newtonian) mechanics. In other words, given all the causes, the effect is 100% entirely determined and not at all random.

To be clear, this view of the world does not quite align with quantum mechanics. Indeed, it may make sense to model emergence and complex systems as fundamentally random. However, this philosophical view of modeling will apply to most things we'll care to model.

2.4.2 Subjective Probability

So, the physical probability I use when I roll the die represents my lack of the demon's superhuman knowledge of the location and momentum of all the die's particles. In other words, when I build probability models of the data generating process, the probability reflects my lack of knowledge. This philosophical idea is called subjective probability or Bayesian probability. The argument goes beyond Bayes rule and Bayesian

statistical estimation to say that probability in the model represents the modeler's lack of complete knowledge about the data generating process and does not represent inherent randomness in the data generating process.

Subjective probability expands our "random physical process" interpretation of probability. The physical interpretation of probability works well for simple "physical processes" like rolling a die, flipping a coin, or shuffling a deck of cards. But, of course, we will want to model many phenomena that are difficult to think of as repeatable physical processes. In these cases, we will still model these phenomena using random generation. The probabilities used in the random generation reflect that while we as modelers may know some detail about the data-generating process, we'll never have the superhuman deterministic level of detail.

2.5 Summary

- A random variable is a variable whose possible values are numerical outcomes of a random phenomenon.
- A probability distribution function is a function that maps the random variable outcomes to a probability value. A joint probability distribution function maps each combination of X and Y to a probability value.
- We derive the chain rule, the law of total probability, and Bayes Rule from the fundamental axioms of probability. These are useful rules in modeling.
- Canonical classes of distributions are mathematically well-described representations of distributions. They provide us with primitives that make probabilistic modeling flexible and relatively easy.
- Canonical distributions are instantiated with a set of parameters., such as location, scale, rate, and shape parameters.
- When we build models knowing what variables are independent or conditionally independent dramatically simplifies the model. In causal modeling, independence and conditional independence will be vital in separating correlation from causation.
- The expected value of a random variable with a finite number of outcomes is the weighted average of all possible outcomes, where the weight is the probability of that outcome.
- Probability is just a value. We need to give that value an interpretation. The physical process interpretation maps probability to the proportion of ties an outcome would occur if a physical process could be run repeatedly ad infinitum.
- In contrast, the Bayesian view of subjective probability interprets probability in terms of beliefs.
- When coding a random process, Pyro allows you to use canonical distributions as primitives in constructing nuanced random process models.
- Monte Carlo algorithms use random generation to estimate expectations from a distribution of interest.
- Popular inference algorithms include graphical model-based algorithms, probability weighting, MCMC, and variational inference.

- Canonical distributions and random processes can serve as proxies for populations for which we wish to model and make inferences. Conditional probability is an excellent way to model heterogeneous subpopulations.
- Difference canonical distributions are used to model different phenomena, such as counts, bell curves, and waiting times.
- Generating from random processes is a good model of real-life sampling independent and identically distributed data.
- Given a dataset, multiple data generating processes could have potentially generated that dataset. This fact connects to the challenge of parsing causality from correlation.
- Statistical independence tests validate independence and conditional independence claims about the underlying distribution.
- There are several methods to learn model parameters, including maximum likelihood estimation and Bayesian estimation.
- Determinism suggests that if we knew everything about a system, we could predict its outcome with zero error. Subjective probability is the idea that probability represents the modeler's lack of that complete knowledge about the system. Adopting these philosophical perspectives will serve us in understanding causal AI.
- A great way to build models is factorizing a joint distribution, simplifying the factors with conditional independence, and then implementing factors as random processes.
- A powerful modeling technique is to use probability distributions to model populations, particularly when you care about heterogeneity in those populations.
- When we use probability distributions to model populations, we can map generating from random processes to sampling from the population.
- While traditional statistical modeling models the observational joint distribution or the full joint distribution, causal modeling models the data generating process.

3

Building a causal graphical model

This chapter covers

- **Building a causal directed acyclic graph (DAG) to model a data generating process**
- **Using your causal graph as a communication, computation and reasoning tool**
- **Building a causal DAG in pgmpy and pyro (PyTorch)**
- **Training a probabilistic machine learning model using the causal DAG as a scaffold.**

In the previous chapter, I introduced the concept of the data generating process and how it relates to the joint probability distribution of the variables in your modeling domain and the data you use in your model. In this chapter, we'll build our first models of the data generating process using the **causal directed acyclic graph** (causal DAG) and causal graphical models built on top of that DAG.

Learning causal modeling requires a bit of a mental refactor. Causal modelers don't model the data; they model the data generating process (DGP). That model attempts to capture how variables relate causally instead of just statistically.

The model-the-data mindset works well for predictions. For example, suppose the DGP yields examples of some predictors and a prediction target. A decent predictive modeling approach will pick up on the the statistical patterns stemming from the probabilistic dependence between the predictors and prediction target. That model would probably produce decent predictions.

However, while data alone can support prediction, we need more than data to make causal inferences. Causal inferences will require us to make explicit modeling assumptions about the DGP. Even causal discovery algorithms that attempt to learn a causal DAG from data rely on strong assumptions about that DGP (e.g., we'll define these assumptions in chapter 9). So, we need to focus on the DGP over the data. That said, we'll see in later chapters how statistical analysis and modeling of data will *help* us determine if our model of the DGP is correct or not.

3.1 Introducing the causal DAG

So, a model of the data generating process attempts to represent how variables relate causally instead of just statistically. What does this representation look like?

One option is dynamic mathematical models such as ordinary differential equations and partial differential equations, as is common in physics and engineering. Another option is to use computational simulators, such as are used in meteorology and climate science.

This book will focus on the causal DAG as our representation of the DGP. We will also look at models that are built on top of causal DAGs. I do not claim that the causal DAG is the "right way" to build a causal model of the DGP relative to alternatives like dynamic models or simulators, or statistical models that encode causal assumptions in different ways. However, I will claim that causal DAGs have advantages over other alternatives. The most basic advantage is that graphs are easy for humans to think about; graphs are the go-to method for making sense of complicated domains.

Secondly, graphs are easy to compute over; they are a fundamental data structure in computer science. Computer scientists have built many efficient algorithms for solving problems on graphs that we can bring to bear on causal reasoning. Finally, graphs form the basis of much of the formal theory in causal inference. You can use that formal theory to get guarantees on your analysis that other methods don't yet provide because researchers have yet to work out those mathematical results for these other causal representations.

3.1.1 How does a directed acyclic graph work as a causal model?

Causal DAGs give us a simple and powerful way to represent causal structure in the DGP and provide structure for our causal model. The causal DAG approach assumes that we will build a model that represents components of the DGP as a discrete set of variables. Those variables may be discrete or continuous. Often the variables are univariate, but they can also be multivariate vectors or matrices. A combination of variable values represents a possible state of the overall data generating process. Importantly, we assume variables are causes/effects of other variables and that these cause-effect relationships reflect true causality in the DGP. Finally, we represent the variables as nodes in the graph, and directed edges correspond to the causal relationships.

For our first example, recall the rock-throwing DGP from Chapter 2. Recall in the example that we start with Jenny and Bryan having a certain amount of inclination to throw rocks at a windowpane that has a certain amount of strength. If either person's inclination to throw suppresses a threshold, they throw. The window breaks depending on if either or both of them throw and the strength of the window. We'll now create the causal DAG that will visualize this process. As a Python function, the DGP is as follows:

Listing 3.1 DAG Rock-throwing example

```
def true_dgp(jenny_inclination, brian_inclination, window_strength):    #A
    jenny_throws_rock = jenny_inclination > 0.5    #B
    brian_throws_rock = brian_inclination > 0.5    #B
    if jenny_throws_rock and brian_throws_rock:    #C
        strength_of_impact = 0.8    #C
    elif jenny_throws_rock or brian_throws_rock:    #D
        strength_of_impact = 0.6    #D
    else:    #E
        strength_of_impact = 0.0    #E
    window_breaks = window_strength < strength_of_impact    #F
return jenny_throws_rock, brian_throws_rock, window_breaks
```

#A Input variables are numbers between 0 and 1
#B Jenny and Brian throw the rock if so inclined
#C If both throw the rock the strength of impact is .8
#D If one of them throws the strength of impact is .6
#E If neither throws the strength of impact is 0
#F The window breaks if strength of imact is greater than window strength

Figure 3.1 illustrates the rock-throwing data generating process as a causal DAG.

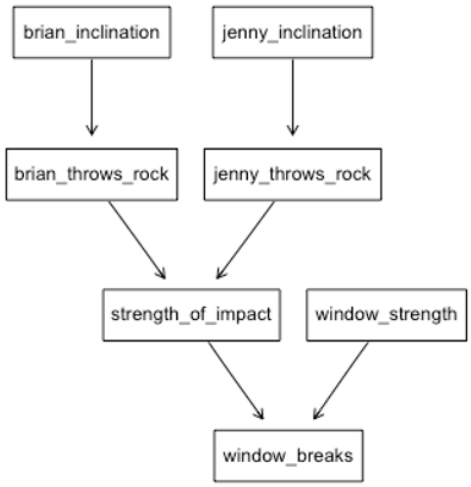


Figure 3.1 A causal DAG representing the rock-throwing data generating process. In this example each node corresponds to a random variable in the data generating process.

In Figure 3.1, each node corresponds to a random variable in the data generating process. The directed edge corresponds to a cause-effect relationship (source node is the cause, end node is the effect).

Causal abstraction and causal representation learning

In modeling, level of abstraction refers to the level of detail and granularity of the variables in the model. In Figure 3.1, there is a mapping between the variables in the DGP and the variables in the causal DAG because the level of abstraction in the data generated by the DGP and the level of abstraction of causal DAG are the same. But it is possible for variables in the data to be at a lower level of abstraction. This is particularly common in machine learning, where we often deal with low-level features, such as pixels.

When the level of abstraction in the data is lower than the level the modeler wants to work with, the modeler must use domain knowledge to derive the high-level abstractions that will appear as nodes in the DAG. For example, a doctor may be interested in a high-level binary variable node like "Tumor (present/absent)," while the data itself contains low-level variables such as a matrix of pixels from medical imaging technology.

That doctor must look at each image in the data set and manually label the high-level tumor variable. Alternatively, a modeler can use analytical means (e.g., math or logic) to map low-level abstractions to high-level ones, a task called *causal abstraction*. Alternatively, they could use machine learning to learn high-level abstractions from lower ones in data, a task called *causal representation learning*. We touch on the latter topic in chapters 5 and 9.

3.1.2 Case Study: A causal model for transportation

The following example features a model of people's choice of transportation on their daily commutes. Find links to accompanying code and video tutorials at https://altdeep.ai/p/causalaibook.

Suppose you were an urban planning consultant trying to model the relationships between people's demographic background, the size of the city where they live, their job status, and their decision on how to commute to work each day.

You break down the key variables in the system as follows:

- **Age (A):** The age of an individual.
- **Gender (S):** An individual's reported gender.
- **Education (E):** The highest level of education or training completed by an individual.
- **Occupation (O):** An individual's employment status.
- **Residence (R):** The size of the city the individual lives in.
- **Travel (T):** The means of transport favored by the individual.

You then think about the causal relationships between these variables using knowledge about the domain. Here is your narrative:

- Educational standards are different across generations. In previous generations, one could achieve a middle-class lifestyle (e.g., own a home, support a family) with a high school degree. In more recent generations, it is hard to get stable employment even with college degrees. Thus age (A) is a cause of education (E).
- Similarly, a person's gender is often a factor in their decision to pursue higher levels of education. So, gender (S) is a cause of education (E).

- Many white-collar jobs require higher education. Many credentialed professions (e.g., doctor, lawyer, or accountant) certainly require higher education. So, education (E) is a cause of occupation (O).
- White collar jobs that depend on higher levels of education tend to cluster in urban areas. Thus, education (E) is a cause of where people reside (R).
- People who are self-employed might work from home and therefore don't need to commute while people with employers do. Thus occupation (O) is a cause of transportation (T).
- People in big cities might find it more convenient to commute by walking or using public transportation, while people in small cities and towns rely on cars to get around. Thus residence (R) is a cause of transportation (T).
- Finally, you reduce this narrative to the following causal DAG shown in Figure 3.2.

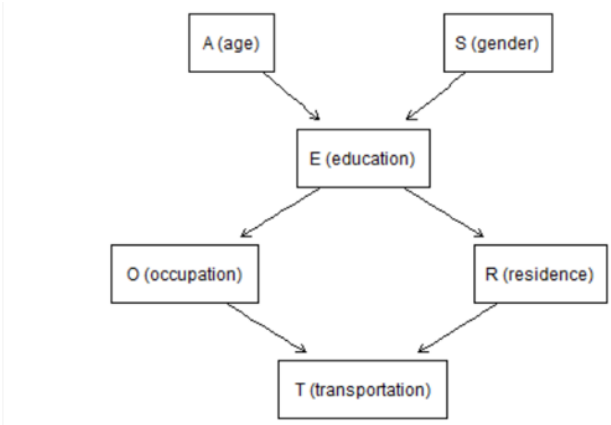


Figure 3.2 Causal DAG representing a model of the causal factors behind how people commute to work.

We can build this causal DAG using the following code.

Listing 3.2 Building the transportation DAG (Figure 3.2) in pgmpy

```
model = BayesianNetwork(    #A
       [
        ('A', 'E'),    #B
        ('S', 'E'),    #B
        ('E', 'O'),    #B
        ('E', 'R'),    #B
        ('O', 'T'),    #B
        ('R', 'T')     #B
      ]
)
```

#A pgmpy provides a *BayesianNetwork* class where we add the edges to the model.
#B Inut the DAG as a list of edges (tuples).

The `BayesianNetwork` object in *pgmpy* is built on the `DiGraph` class from *networkx*, the preeminent graph modeling library in Python.

3.1.3 Why is it a directed acyclic graph?

The causal DAG is acyclic, meaning it doesn't allow for any cycles. Intuitively, this captures the notion that causes precede effects in time; and so a cycle would imply things that happen later in time cause things that happen earlier.

In some causal systems, relaxing the acyclicity constraint makes sense, such as with systems that have feedback loops. Some formal causal models allow for acyclicity. However, many such systems can be abstracted to have acyclicity. Sticking to the acyclic allows us to use the benefits of the causal DAG formalism.

3.2 The benefits of the DAG as a causal representation

There are several benefits of using the causal DAG as a representation of the data generating process. These benefits, which I will discuss further in the following sections, break down as follows,

1. DAGs are useful in communicating and visualizing causal assumptions.
2. It is easy to compute over DAGs.
3. DAGs link causality to conditional independence.
4. DAGs can provide scaffolding for probabilistic ML models.

3.2.1 DAGs are useful in communicating and visualizing causal assumptions.

A Causal DAG Is a powerful communication device. Visual communication of information is about highlighting important information at the expense of other information. As an analogy, consider the two maps of the London Underground in Figure 3.3.

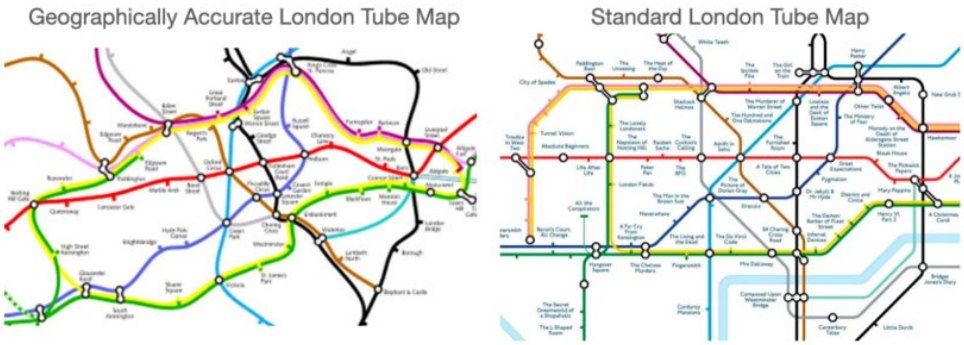


Figure 3.3 Visual communication is a powerful use case for a graphical representation. For instance, the map of the London Underground on the left is geographically accurate while the one on the right trades that accuracy for a clear representation of each station's position relative to the others. That latter is more useful for train riders than geographic detail. Similarly, a causal DAG abstracts away much detail of causal mechanism to a simple representation that is easy to reason about visually.

The map on the left is geographically accurate. The simpler map on the right ignores the geographic detail and focuses on the position of each station relative to other stations, which is, arguably, all one needs to find their way around London.

Similarly, a causal DAG highlights causal relationships while ignoring other things. For example, the rock-throwing DAG ignores the if-then conditional logic of how Jenny and Brian's throws combined to break the window. Similarly, the transportation DAG says nothing about the types of variables we are dealing with. Should we consider age (A) in terms of continuous-time, integer years, categories like young/middle-aged/elderly, or intervals like 18-29 / 30-44 / 45-64 / >65? What are the categories of the transportation variable (T)? Could the occupation variable (O) be a multi-dimensional tuple like {employed, engineer, works-from-home}? The DAG also fails to capture which of these variables are observed in the data, nor the number of data points in that data.

CAUSAL DAGS DON'T ILLUSTRATE MECHANISM

The graph also doesn't visualize interactions between causes. For example, in older generations, women were less likely to go to college than men. In younger generations, the reverse is true. While both age (A) and gender (S) are causes of education (E), you can't look at the DAG and see anything about how age and gender interact to affect education.

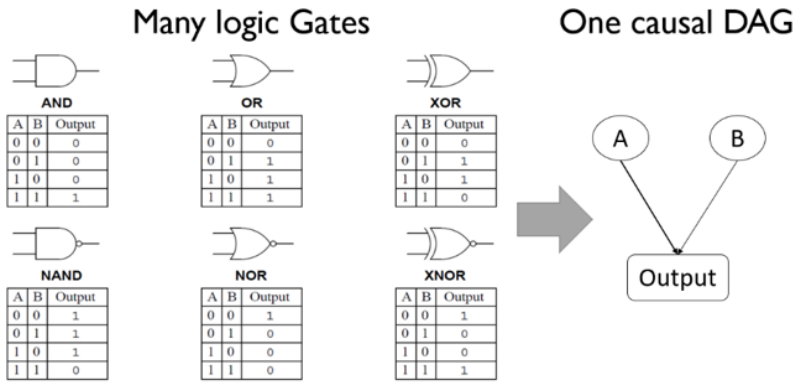


Figure 3.4 The various kinds of logic gates all have the same causal DAG.

More generally, DAGs don't convey any information about the causal mechanism or *how* the causes impact the effect. Consider for example the various logic gates in Figure 3.4 . The input binary values for A and B determine the output differently depending on the type of logic gate. But if we represent a logic gate as a causal DAG, then all the logic gates have the same causal DAG. In other words, the causal DAG doesn't visually represent any mechanism for how the causal parents interact to determine the value of the child. We can use the causal DAG as a scaffold for causal graphical models that capture this logic, but we can *see* the logic in the DAG.

This is a strength and a weakness. The causal DAG simplifies matters by communicating *what* causes what, but not *how*. But in some cases, visualizing the "how" would be desirable.

CAUSAL DAGS REPRESENT CAUSAL ASSUMPTIONS

The causal DAG represents the modeler's assumptions and beliefs about the data generating process because we don't have access to that process most of the time. Thus, the causal DAG allows us to visualize our assumptions and communicate them to others.

Beyond this visualization and communication, the benefits of the causal DAG are mathematical and computational (I explain these in the following subsections). Causal inference researchers vary in their opinons on the degree to which these mathematical and computational properties of the causal DAG are practically beneficial. However, most agree on this fundamental benefit of visualization and communication of causal assumptions.

The assumptions encoded in a causal DAG are strong. Consider, for example, the transportation DAG.

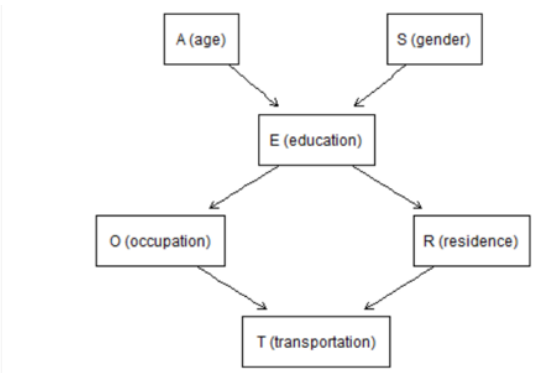


Figure 3.5 The causal DAG model of transportation choices.

Consider the alternatives to that DAG; how many possible DAGs could we draw on this simple six node system? The answer is 3,781,503. So, when I use a causal DAG to communicate my assumptions about this system, I'm communicating my top choice over 3,781,502 alternatives.

And how about some of those competing DAGs? Some of them seem plausible. Perhaps baby boomers prefer small-town life while millennials prefer city life, implying there should be an A → R edge? Perhaps gender determines preferences and opportunities in certain professions and industries, implying an G → O edge? The assumption that age and gender cause occupation and residence only indirectly through education is a powerful assumption that will provide useful inferences *if it is right*.

But what if our causal DAG is wrong? It seems it is likely to be wrong given its 3,781,502 competitors. In chapter 4, we'll learn to use data to show us when the causal assumptions in our chosen DAG fail to hold.

3.2.2 It is easy to compute over DAGs.

Directed graphs are well-studied objects in math and computer science. In computer science, they are a fundamental data structure. Computer scientists have solved many practical problems with graph algorithms with theoretical guarantees on how long they take to arrive at solutions. Most developers and data scientists have some exposure to graphical libraries in their scripting language of choice, such as *networkx* in Python and *igraph* in R.

We can bring that mathematical and computational theory and tooling to bear on a causal modeling problem when we represent the causal model in the form of a causal DAG. For example, in *pgmpy* we can train a causal DAG on data to get a directed causal graphical model. Given that model, we can apply algorithms for graph-based probabilistic inference, such as *belief propagation*, to estimate conditional probabilities defined on variables in the graph. The directed graph structure enables these algorithms to work in typical settings without our needing to configure them to a specific problem or task.

In the next chapter, we'll introduce the concept of d-separation, which is a graphical abstraction for conditional independence, and the fundamental idea behind the do-calculus theory for causal inference. D-separation is all about finding paths between nodes in the directed graph, something any worthwhile graph library makes easy by default. Indeed, conditional independence is the key idea behind the third benefit of the causal DAG.

Representing time in Causal DAGs

One of the benefits of visualizing a data generating process as a causal DAG is the DAG has an implicit representation of time. In more technical terms, the DAG provides a "partial ordering" that we can read as a partial temporal ordering because causes precede effects in time.

For example, consider the graph in Figure 3.5. This graph describes a data generating process where a change in cloud cover (Cloudy) causes both a change in the state of a weather-activated sprinkler (Sprinkler) and the state of rain (Rain), and these both cause a change in the state of the wetness of the grass (Wet Grass). So, we know that change in the state of a weather causes rain and sprinkler activation, and that these both cause a change in the state of the wetness of the grass. However, the graph doesn't tell us which happens first, the sprinkler activation or the rain.

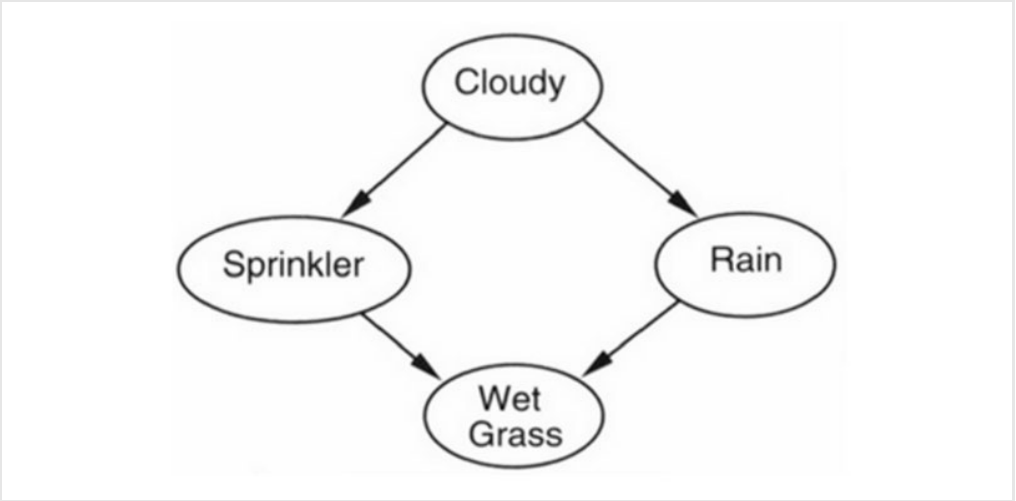


Figure 3.6 A causal DAG representing a sprinkler system that activates when the sky is not cloudy. Cloudy skies lead to rain. Both rain and sprinklers make the grass wet.

This partial ordering in Figure 3.6 may seem trivial but consider the DAG in Figure 3.7. Visualization libraries can use the partial ordering in the hairball-like DAG on the right into the much more readable form on the left.

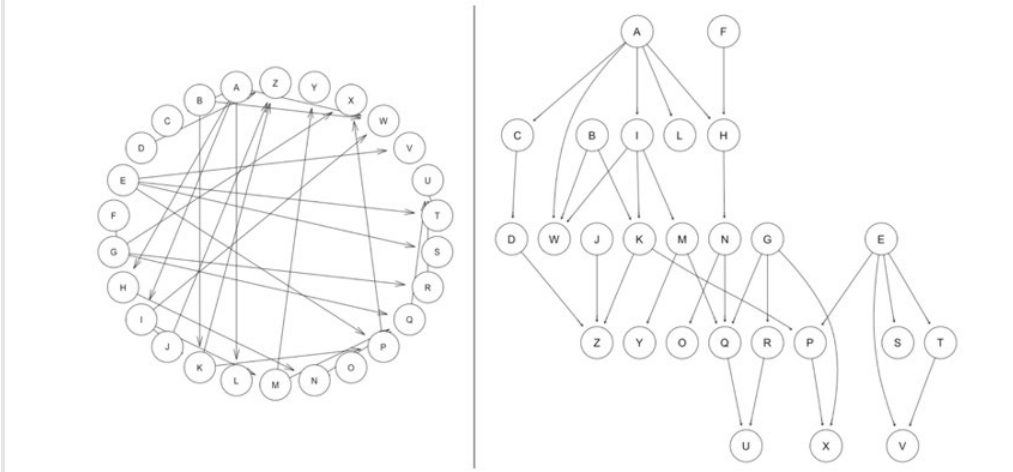


Figure 3.7 A visualization library can use the DAG's partial ordering to unravel the hairball-like DAG on the right into a more readable form (right).

However, sometimes we need a causal DAG to be more explicit about time. For example, we may be modeling causality in a dynamic setting, such as in reinforcement learning. In this case, we can make time explicit in defining and labeling the variables of the model, as in Figure 3.8.

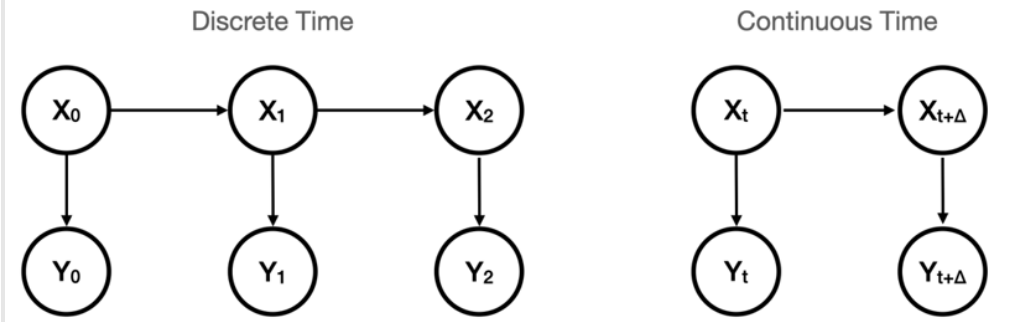


Figure 3.8 If we need a causal DAG to be explicit about time, we can make time explicit in the definition of the variables and labeling of their nodes. We can represent continuous time with interval variables like "Δ."

3.2.3 DAGs link causality to conditional independence

The third benefit of the DAG is that it allows us to use causality to reason about conditional independence. Humans have an innate ability to reason in terms of causality. That ability is how we get the first and second benefits of that causal DAG. But reasoning probabilistically doesn't come nearly as easily. So, the ability to use causality to reason about conditional independence (a concept from probability) is a considerable feature of the DAG.

Consider again the transportation DAG, displayed again in Figure 3.8.

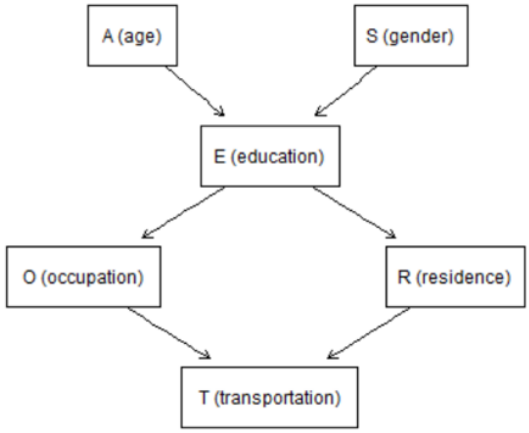


Figure 3.9 The causal DAG for transportation choices.

Those six variables have a joint distribution P(A,S,E,O,R,T). Recall the chain rule from chapter 2, which says that we can factorize any joint probability into a chain of conditional probability factors. For example:

$$
\begin{aligned}
&P(a,s,e,o,r,t)\\
&=P(e)P(s \mid e)P(t \mid s,e)P(a \mid t,s,e)P(o \mid a,t,s,e)P(r \mid o,t,s,e)\\
&=P(t)P(o \mid t)P(r \mid o,t)P(e \mid r,o,t)P(a \mid e,r,o,t)P(s \mid a,e,r,o,t)\\
&= \ldots
\end{aligned}
$$

The chaining works for any ordering of the variables. But instead of choosing any ordering, we'll choose the (partial) ordering of the causal DAG, since that ordering aligns with our assumptions of the causal flow of the variables in the data generating process. Looking at Figure 3.9, the ordering of variables is {(A, S), E, (O, R), T}. Letting A come before S and O, come before R, we get:

$$
\begin{aligned}
&P(a,s,e,o,r,t)\\
&=P(a)P(s \mid a)P(e \mid s,a)P(o \mid e,s,a)P(r \mid o,e,s,a)P(t \mid o,r,e,s,a)
\end{aligned}
$$

Further, we'll use the causal DAG to further simplify this factorization. Each factor is a conditional probability. We'll simplify those factors by conditioning **only the parents of each node** in the DAG. In other words, for each variable, we look at that variable's direct parents in the graph, then we drop everything on the right-hand side of the conditioning bar "|" that isn't one of those direct parents. If we condition only on parents, then we get the following simplification:

$$P(a,s,e,o,r,t)$$
$$=P(a)P(s|a)P(e|s,a)P(o|e,s,a)P(r|o,e,s,a)P(t|o,r,e,s,a)$$
$$=P(a)P(s)P(e|s,a)P(o|e)P(r|e)P(t|o,r)$$

What is going on here? Why should the causal DAG magically mean we can say *P(s|a)* is equal to *P(s)* and *P(r|o,e,s,a)* simplifies to a mere *P(r|e)*? An astute reader of Chapter 2 will realize that stating that *P(s|a) = P(s)* and *P(t|o,r,e,s,a)=P(t|o,r)* is equivalent to saying that S and A are independent, and T is conditionally independent of E, S, and A given O and R. In other words, the causal DAG gives us a way to impose conditional independence constraints over the joint probability distribution of the variables in the data generating process.

Why should you care about things being conditional independent? Conditional independence makes your life as a modeler easier. For example, suppose you were to model the transportation variable T with a predictive model. The predictive model implied by *P(t|o,r,e,s,a)* requires having features O, R, E, S, and A, while the predictive model inspired by *P(t|o,r)* just requires features O and R. The latter model would have less parameters to learn, have more degrees of freedom, take less space in memory, train faster, etc.

But why does the causal DAG give us the right to impose conditional independence?

THE CAUSAL MARKOV PROPERTY

Let's build some intuition about the connection between causality and conditional independence. Consider the example of using genetic data from family members to make conclusions about an individual. For example, the Golden State Killer was a California-based serial killer captured using genetic genealogy. Investigators used DNA left by the killer at crime-scenes to identify genetic relatives in public databases. They then triangulated from those relatives to find the killer.

Suppose you had a close relative and a distant relative on the same line of ancestry. Could the distant relative provide any additional information once we had genetic information about that close relative? Let's simplify a bit by focusing just on blood type. Suppose the close relative was your father, and the "distant" relative was your paternal grandfather, as in Figure 3.10. Indeed, your grandfather's blood type is a cause of yours. If we saw a large dataset of grandfather/grandchild blood type pairs, we'd see a correlation. However, your father's blood type is a more direct cause, and the connection between your grandfather's blood type and yours passes through your father. So, if our goal were to predict your blood type and we already had your father's blood type as a predictor, your father's blood type could provide no additional predictive information. Thus, your blood type and your paternal grandfather's blood type are conditionally independent, given your father's blood type.

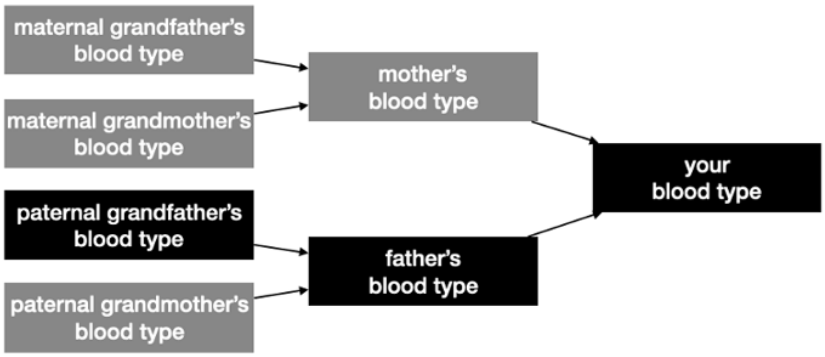


Figure 3.10 Causality implies conditional independence. Your paternal grandfather's blood type is a cause of your father's, which is a cause of yours. But knowing your grandfather's type has no benefit in predicting your type once we know your father's (i.e., conditional independence).

The way causality makes correlated variables conditionally independent is called the **causal Markov property**. In concrete graphical terms, the causal Markov property means that variables are conditionally independent of their non-descendants (e.g., ancestors, "uncles/aunts," "cousins," etc.) given their parents in the graph.

This "non-descendants" definition of the causal Markov property is sometimes called the "local Markov property." An equivalent articulation is called the Markov factorization property, which is exactly the property that if your causal DAG is true, you can factorize a joint probability into conditional probabilities of variables given their parents in the causal DAG:

$$P(a,s,e,o,r,t) = P(a)P(s)P(e \mid s,a)P(o \mid e)P(r \mid e)P(t \mid o,r)$$

If our transportation DAG is a true representation of the data generating process, then the local Markov property should to hold true.

3.2.4 DAGs can provide scaffolding for probabilistic ML models.

Many modeling approaches in probabilistic machine learning use a DAG as the model structure. Examples include:

- Directed graphical models (AKA Bayesian networks).
- Latent variable models (e.g., topic models).
- Deep generative models such as variational autoencoders.

The advantage of building a probabilistic machine learning model on top of a causal graph is, rather obviously, that you have a probabilistic *causal* machine learning model. You can train it on data, and you can use it for prediction and other inferences like any probabilistic machine learning model. Moreover, because it is built on top of a causal DAG, it is a causal model. Therefore, you may be able to use it to make certain causal inferences, provided you

meet the conditions for making those inferences. I discuss those conditions in Part 2 of this book.

3.3 Building a probabilistic machine learn model on a causal DAG

Recall our factorization of the joint probability distribution of the transportation variables over the DAG in the transportation DAG.

$$P(a,s,e,o,r,t) = P(a)P(s)P(e \mid s,a)P(o \mid e)P(r \mid e)P(t \mid o,r)$$

We have a set of factors $\{ P(a), P(s), P(e \mid s,a), P(o \mid e), P(r \mid e), P(t \mid o,r) \}$. From here on, we'll borrow the term "Markov kernel" from probability theory and call these factors *causal Markov kernels*. We'll build our probabilistic machine learning model by implementing the causal Markov kernels in code then composing them into one model. Our implementations for each kernel will be able to return a probability value given input arguments. For example, $P(a)$ will take in an outcome value for A and return a probability value for that outcome. Similarly, $P(t \mid o,r)$ will take in values for T, O, and R and return a probability value for T. Our implementations will also be able to generate from the causal Markov kernels. To do this, these implementations will require parameters that map the inputs to the outputs. We'll use standard statistical learning approaches to fit those parameters from the data.

3.3.1 Training a model on the causal DAG

Consider the data generating process for the transportation DAG. What sort of data would this process generate?

Suppose we administered a survey covering 500 individuals, getting values for each of the variables in this DAG. The data encodes the variables in our DAG as follows.

- **Age (A):** Recorded as *young* (**young**) for individuals up to and including 29 years, *adult* (**adult**) for individuals between 30 and 60 years old (inclusive), and *old* (**old**) for people 61 and over.
- **Gender (S):** The self-reported gender of an individual, recorded as *male* (**M**), *female* (**F**), or other (**O**).
- **Education (E):** The highest level of education or training completed by the individual, recorded either *high school* (**high**) or *university degree* (**uni**).
- **Occupation (O):** *Employee* (**emp**) or a *self-employed* (**self**) worker.
- **Residence (R):** The population size of the city the individual lives in, recorded as *small* (**small**) or *big* (**big**).
- **Travel (T):** The means of transport favored by the individual, recorded as *car* (**car**), *train* (**train**) or *other* (**other**)

Labeling Causal Abstractions

How we conceptualize the variables of a model matters greatly in machine learning. For example, ImageNet, a database of 14 million images, has historically contained anachronistic and offensive labels for racial categories. Even if renamed to be less offensive, race categories themselves are fluid across time and culture. What are the "correct" labels to use in a predictive algorithm?

These are not merely matters of identity politics. Philosophers have long understood that how we define the abstractions in a problem domain impacts the inferences and predictions we make about those abstractions. A famous example is Nelson Goodman's "New Riddle of Induction", a thought experiment that demonstrates label definitions that create paradoxical predictions.

Many data scientists blindly model data and do not think about the data generating process. When you don't think about the data generating process it is easy to take the provenance of the variable definitions in the data for granted. Indeed, most causal inference texts assume you already know what variables you are working with.

But the problem of defining the causal abstractions in our model is non-trivial. The definitions constrain the causal questions we hope to answer with the model. Evidence from cognitive science suggests humans can alternate between causal abstractions to reason about different problems in different domains. This suggests that our problems should define the variables in the DAG and not the other way around.

In chapter 5, I'll introduce the idea of "no causation without manipulation," an idea that provides a useful heuristic for how to define causal variables.

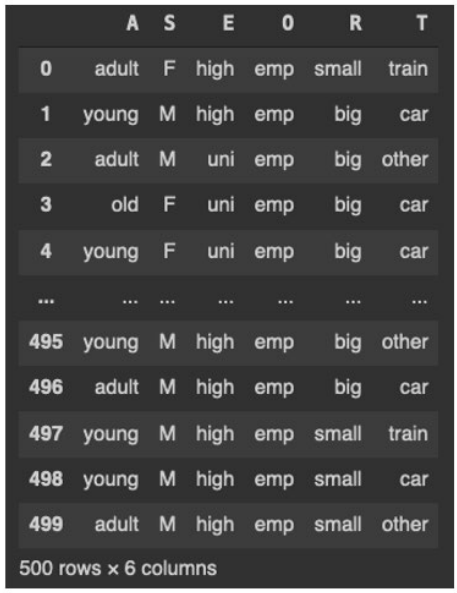
The variables in the transportation data are all **categorical variables**. In this simple categorical case, we can rely on a graphical modeling library like ***pgmpy***.

Listing 3.3 Loading transportation data

```
import pandas as pd
url='https://raw.githubusercontent.com/altdeep/causalML/master/datasets/transportation_surv
       ey.csv'    #A
data = pd.read_csv(url)
data
```

#A We'll load the data into a pandas dataframe with the read_csv method.

This produces the following data frame.



	A	S	E	O	R	T
0	adult	F	high	emp	small	train
1	young	M	high	emp	big	car
2	adult	M	uni	emp	big	other
3	old	F	uni	emp	big	car
4	young	F	uni	emp	big	car
...	...	...	...	...	...	...
495	young	M	high	emp	big	other
496	adult	M	high	emp	big	car
497	young	M	high	emp	small	train
498	young	M	high	emp	small	car
499	adult	M	high	emp	small	other

500 rows × 6 columns

Figure 3.11 An example of data from the data-generating process underlying the transportation model. In this case the data is 500 survey responses.

The `BayesianNetwork` class we initialized in Listing 3.1 has a fit method that will learn the parameters of our causal Markov kernels. Since our variables are categorical, our causal Markov kernels will be in the form of conditional probability tables represented by pgmpy's TabularCPD class.

Listing 3.4 Learning parameters for the causal Markov kernels in the transportation model

```
model = BayesianNetwork(
      [
        ('A', 'E'),
        ('S', 'E'),
        ('E', 'O'),
        ('E', 'R'),
        ('O', 'T'),
        ('R', 'T')
      ]
)
model.fit(data)    #A
causal_markov_kernels = model.get_cpds()    #B
print(causal_markov_kernels)    #B
```

#A The *fit* method on the BayesianNetwork object will estimate parameters from data (a pandas DataFrame).
#B Retrieve and view the causal Markov kernels learned by fit.

```
[<TabularCPD representing P(A:3) at 0x7fb030dd1050>,
 <TabularCPD representing P(E:2 | A:3, S:2) at 0x7fb0318121d0>,
 <TabularCPD representing P(S:2) at 0x7fb03189fe90>,
 <TabularCPD representing P(O:2 | E:2) at 0x7fb030de85d0>,
 <TabularCPD representing P(R:2 | E:2) at 0x7fb030dfa890>,
 <TabularCPD representing P(T:3 | O:2, R:2) at 0x7fb0316c9110>]
```

Let's look at the structure of the causal Markov kernel for the transportation variable T.

```
cmk_T = causal_markov_kernels[-1]
print(cmk_T)
```

This implements the causal Markov kernel *P(T|O,R)* as a conditional probability table, a type of look-up table where given a value of T, O, and R we get the corresponding probability mass value. For example, P(T=car|O=emp, R=big) = 0.7034. Note that these are conditional probabilities. For each combination of values for O and R, there are conditional probabilities for the three outcomes of T that sum to 1. For example, when O=emp and R=big, P(T=car| O=emp, R=big) + (P(T=other| O=emp, R=big) + P(T=train| O=emp, R=big) = 1.

The causal Markov kernel in the case of nodes with no parents is just a simple probability table. For example, the following code block prints the causal Markov kernel for gender (S)

```
print(causal_markov_kernels[2])
+------+-------+
| S(F) | 0.522 |
+------+-------+
| S(M) | 0.478 |
+------+-------+
```

This fit method learns parameters by calculating the proportions of each class in the data. Alternatively, we could have used other techniques for parameter learning.

3.3.2 Different techniques for parameter learning

There are several ways we could go about training these parameters. The following highlights a few common ways of training parameters in conditional probability tables.

MAXIMUM LIKELIHOOD ESTIMATION

The learning algorithm I used in the `fit` method on the `BayesianNetwork` model object was *maximum likelihood estimation* (discussed in chapter 2). Maximum likelihood estimation is the default parameter learning method, so I didn't specify "maximum likelihood" in the call to `fit`. Generally, maximum likelihood estimation seeks the parameter that maximizes the likelihood of seeing the data we use to train the model. In the context of categorical data, maximum likelihood estimation is equivalent to taking proportions of counts in the data. For example, the parameter for P(O=emp|E=high) is calculated as:

$$\frac{\#\text{ observations where O=emp and E=high}}{\#\text{ observations where E = high}}$$

BAYESIAN ESTIMATION

In chapter 2, we also introduced Bayesian estimation. Bayesian inference techniques are generally mathematically intractable and rely on algorithms (e.g., sampling algorithms and variational inference). However, there is an approach that uses "conjugate priors" to derive simple mathematical parameter estimates. Conjugate priors are a special mathematical case of canonical prior distribution that provide pre-determined posteriors with the same canonical form. That means the code implementation can just calculate the parameter value with simple math without the need for complicated Bayesian inference algorithms.

For example, pgmpy implements a Dirichlet conjugate prior for categorical outcomes. In other words, the posterior distribution of the parameters in P(O=emp|E=high) will also have a Dirichlet distribution. pgmpy will use the mean of that distribution to provide point estimates of those parameters.

Listing 3.5 Bayesian point estimation with a Dirichlet conjugate prior

```
from pgmpy.estimators import BayesianEstimator    #A
estimator = BayesianEstimator(model, data)    #A
model.fit(
    data,
    estimator=BayesianEstimator,    #B
    prior_type="dirichlet",
    pseudo_counts=1    #C
)
causal_markov_kernels = model.get_cpds()     #D
cmk_T = causal_markov_kernels[-1]    #D
print(cmk_T)    #D
```

#A Import BayesianEstimator and initialize it on the model and data.
#B Pass the estimator object to the fit method.
#C pseudo_counts refers to the parameters of the Dirichlet prior.
#D Extract the causal Markov kernels and view P(T|O,R).

```
+----------+--------------------+-----+--------------------+----------+
| O        | O(emp)             | ... | O(self)            | O(self)  |
+----------+--------------------+-----+--------------------+----------+
| R        | R(big)             | ... | R(big)             | R(small) |
+----------+--------------------+-----+--------------------+----------+
| T(car)   | 0.7007299270072993 | ... | 0.4166666666666667 | 0.5      |
+----------+--------------------+-----+--------------------+----------+
| T(other) | 0.1362530413625304 | ... | 0.3333333333333333 | 0.25     |
+----------+--------------------+-----+--------------------+----------+
| T(train) | 0.1630170316301703 | ... | 0.25               | 0.25     |
+----------+--------------------+-----+--------------------+----------+
```

In contrast to maximum likelihood estimation, Bayesian estimation of a categorical parameter with a Dirichlet prior acts like a smoothing mechanism. For example, the maximum likelihood parameter estimate says 100% of self-employed people in small towns take a car to work. This is probably extreme. Certainly, some self-employed people bike to work, we just didn't manage to survey any of them. Some small cities, such as Crystal City in the US state of Virginia (population 22,000), have subway stations. I'd wager a least a few of the entrepreneurs in those cities use the train.

Causal Modelers and Bayesians

The Bayesian philosophy goes beyond mere parameter estimation. Indeed, Bayesian philosophy has much in common with causal modeling with a DAG. Bayesians try to encode subjective beliefs, uncertainty, and prior knowledge into "prior" probability distributions on variables in the model. Causal modelers try to encode subjective beliefs and prior knowledge about the data generating process into the form of a causal DAG. The two approaches are compatible. Given a casual DAG, you can be Bayesian about inferring the parameters of the probabilistic model you build on top of the causal DAG. You can even be Bayesian about the DAG itself and compute probability distributions on DAGs!

OTHER TECHNIQUES FOR PARAMETER ESTIMATION

We need not use a conditional probability table to represent the causal Markov kernels. There are models within the generalized linear modeling framework for modeling categorical outcomes. For some of the variables in the transportation model, we might have used non-categorical outcomes. Age, for example, might have been recorded as an integer outcome in the survey. For variables with numeric outcomes, we might use other modeling approaches. You can also use neural network architectures to model individual causal Markov kernels.

Parametric assumptions refer to how we specify the outcomes of a node in the DAG (e.g., category or real number) and how we map parents to the outcome (e.g., table or neural network). Note that the causal assumptions encoded by the causal DAG are decoupled from the parametric assumptions for a causal Markov kernel. For example, when we assumed that age was a direct cause of education level and encoded that into our DAG as an edge, we didn't have to decide if we were going to treat age as an ordered set of classes or as an integer, or as seconds elapsed since birth, etc. Furthermore, we didn't have to know whether to use a conditional categorical distribution or a regression model. That step comes after we specify the causal DAG and want to implement P(E|A, S).

Similarly, when we make predictions and probabilistic inferences on a trained causal model, the considerations of what inference/prediction algorithms to use, while important, are separate from our causal questions. This separation simplifies our work. Often enough, we can build our knowledge and skill set in causal modeling and reasoning independently of our knowledge of statistics, computational Bayes, and applied machine learning.

3.3.3 Learning parameters when there are latent variables

Since we are modeling the data-generating process and not the data, it will be likely that some nodes in the causal DAG will not be observed in the data. Fortunately, probabilistic machine learning provides us with tools for learning latent variables.

LEARNING LATENT VARIABLES WITH PGMPY AND STRUCTURAL EM

To illustrate, suppose the education variable in the transformation survey data were not recorded. *pgmpy* gives us a utility for learning the causal Markov kernel for latent E using an algorithm called *structural expectation maximization*, which is a variant of parameter learning with maximum likelihood.

Listing 3.6 Training a causal graphical model with a latent variable.

```
import pandas as pd
from pgmpy.models import BayesianNetwork
from pgmpy.estimators import ExpectationMaximization as EM
url='https://raw.githubusercontent.com/altdeep/causalML/master/datasets/transportation_surv
       ey.csv'    #A
data = pd.read_csv(url)    #A
data_sans_E = data[['A', 'S', 'O', 'R', 'T']]    #B
model_with_latent = BayesianNetwork(
       [
        ('A', 'E'),
        ('S', 'E'),
        ('E', 'O'),
        ('E', 'R'),
        ('O', 'T'),
        ('R', 'T')
     ],
     latents={"E"}    #C
)
estimator = EM(model_with_latent, data_sans_E)    #D
cmks_with_latent = estimator.get_parameters(latent_card={'E': 2})    #D
print(cmks_with_latent[1].to_factor)    #E
```

#A Download the data and convert to a pandas data frame.
#B Keep all the columns except education E.
#C Indicate which variables are latent when training the model.
#D Run the structural expectation maximization algorithm to learn the causal Markov kernel for E. You have to indicated the cardinality of the latent variable.
#E Print out the learned causal Markov kernel for E. Print it as a factor object for legibility.

The last line prints a factor object.

```
+------+----------+------+--------------+
| E    | A        | S    |   phi(E,A,S) |
+======+==========+======+==============+
| E(0) | A(adult) | S(F) |       0.1059 |
+------+----------+------+--------------+
| E(0) | A(adult) | S(M) |       0.1124 |
+------+----------+------+--------------+
| E(0) | A(old)   | S(F) |       0.4033 |
+------+----------+------+--------------+
| E(0) | A(old)   | S(M) |       0.2386 |
+------+----------+------+--------------+
| E(0) | A(young) | S(F) |       0.4533 |
+------+----------+------+--------------+
| E(0) | A(young) | S(M) |       0.6080 |
+------+----------+------+--------------+
| E(1) | A(adult) | S(F) |       0.8941 |
+------+----------+------+--------------+
| E(1) | A(adult) | S(M) |       0.8876 |
+------+----------+------+--------------+
| E(1) | A(old)   | S(F) |       0.5967 |
+------+----------+------+--------------+
| E(1) | A(old)   | S(M) |       0.7614 |
+------+----------+------+--------------+
| E(1) | A(young) | S(F) |       0.5467 |
+------+----------+------+--------------+
| E(1) | A(young) | S(M) |       0.3920 |
+------+----------+------+--------------+
```

The outcomes for E are 0 and 1 because the algorithm doesn't know the outcome names. Perhaps 0 is "high" and 1 is "uni," but correctly mapping the default outcomes from a latent variable estimation method to the names of those outcomes would require further assumptions.

There are other algorithms for learning parameters when there are latent variables, including some that use special parametric assumptions (i.e., functional assumptions about how the latent variables relate to the observed variables).

LATENT VARIABLES AND IDENTIFICATION

In statistical inference, we say a latent variable is "identified" when it is theoretically impossible to learn its true value given an infinite number of examples in the data. Unfortunately, your data may not be sufficient to learn the latent variables in your causal DAG. More specifically, the observations of the observed variables may not be sufficient to identify the latent variable. If we did not care about representing causality, we could just restrict ourselves to a latent variable graphical model with latent variables that are identifiable from data. A causal DAG is driven by the data generating process, not the data.

That said, even if you have non-identifiable latent variables, you still may be able to identify the quantity that answers your causal question. Indeed, much of causal inference methodology is focused on robust estimation of causal effects (how much a cause affects an effect) despite the presence of latent "confounders." On the other hand, even if your latent variables are identified, the quantity that answers your causal question may not be identified. We'll cover this in detail when we discuss the causal hierarchy in part 3 of this book.

3.3.4 Inference with a trained causal probabilistic machine learning model

A probabilistic machine learning model that models the observational distribution can use computational inference algorithms to infer the conditional probability of an outcome for any set of variables given outcomes for the other variables. We use the variable elimination algorithm for a directed graphical model with categorical outcomes (introduced in chapter 2).

For example, suppose we want to compare education levels amongst car drivers to that of train riders. Then, we can calculate and compare P(E|T) when T=car to when T=train using variable elimination, an inference algorithm for tabular graphical models.

Listing 3.7 Inference on the trained causal graphical model

```
from pgmpy.inference import VariableElimination     #A
inference = VariableElimination(model)
query1 = inference.query(['E'], evidence={"T": "train"})
query2 = inference.query(['E'], evidence={"T": "car"})
print("train")
print(query1)
print("car")
print(query2)
```

#A VariableElimination is an inference algorithm specific to graphical models.

This prints the probability tables for "train" and "car."

```
"train"
+---------+----------+
| E       |   phi(E) |
+=========+==========+
| E(high) |   0.6162 |
+---------+----------+
| E(uni)  |   0.3838 |
+---------+----------+
"car"
+---------+----------+
| E       |   phi(E) |
+=========+==========+
| E(high) |   0.5586 |
+---------+----------+
| E(uni)  |   0.4414 |
+---------+----------+
```

It seems car drivers are more likely to have a university education. That inference is based on our DAG-based causal assumption that university education indirectly determines how people get to work.

In a tool like *pyro*, you have to be a bit more hands-on with the inference algorithm. The following illustrates the inference of P(E|T="train") using a probabilistic inference algorithm called importance sampling.

Listing 3.8 Implementing the trained causal model in pyro

```
import torch
import pyro
from pyro.distributions import Categorical

A_alias = ['young', 'adult', 'old']  #A
S_alias = ['M', 'F']   #A
E_alias = ['high', 'uni']  #A
O_alias = ['emp', 'self']  #A
R_alias = ['small', 'big']  #A
T_alias = ['car', 'train', 'other']  #A

A_prob = torch.tensor([0.3,0.5,0.2]) #B
S_prob = torch.tensor([0.6,0.4]) #B
E_prob = torch.tensor([[[0.75,0.25], [0.72,0.28], [0.88,0.12]],  #B
                       [[0.64,0.36], [0.7,0.3], [0.9,0.1]]])   #B
O_prob = torch.tensor([[0.96,0.04], [0.92,0.08]])  #B
R_prob = torch.tensor([[0.25,0.75], [0.2,0.8]])    #B
T_prob = torch.tensor([[[0.48,0.42,0.1], [0.56,0.36,0.08]],  #B
                       [[0.58,0.24,0.18], [0.7,0.21,0.09]]])   #B

def model():  #C
   A = pyro.sample("age", Categorical(probs=A_prob))  #C
   S = pyro.sample("gender", Categorical(probs=S_prob)) #C
   E = pyro.sample("education", Categorical(probs=E_prob[S][A])) #C
   O = pyro.sample("occupation", Categorical(probs=O_prob[E]))   #C
   R = pyro.sample("residence", Categorical(probs=R_prob[E]))    #C
   T = pyro.sample("transportation", Categorical(probs=T_prob[R][O])) #C
   return{'A': A,'S': S,'E': E,'O': O,'R': R,'T': T}  #C

pyro.render_model(model)   #D
```

#A The categorical distribution only returns integers, so it's useful to write the integers' mapping to categorical outcome names.

#B For simplicity I'll use rounded versions of parameters learned with the "fit" method in pgmpy (Listing 3-4), though I could have learned the parameters in a training procedure.

#C When you implement the model in pyro, you specify the causal DAG implicitly using code logic.

#D You can then generate a figure of the implied DAG using pyro.render_model().

The function `pyro.render_model` draws the implied causal DAG from the pyro model in Figure 3.11.

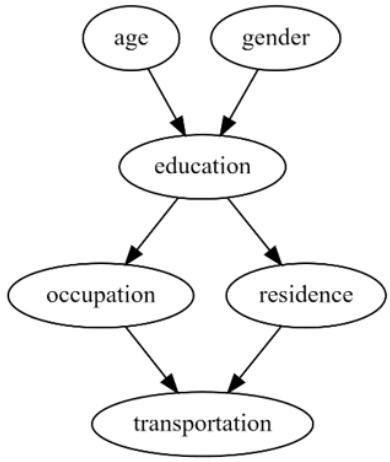


Figure 3.12 You can visualize the causal DAG in pyro by using the pyro.render_model() function.

Pyro provides probabilistic inference algorithms such as importance sampling that we can apply to our causal model.

Listing 3.9 Inference on the causal model in pyro

```
import numpy as np
import pyro
from pyro.distributions import Categorical
from pyro.infer import Importance, EmpiricalMarginal    #A
import matplotlib.pyplot as plt

conditioned_model = pyro.condition(    #B
    model,    #C
    data={'T':torch.tensor(1)}    #D
)

m = 5000    #E
posterior = pyro.infer.Importance(    #F
conditioned_model,    #G
num_samples=m    #H
).run()    #I
E_marginal = EmpiricalMarginal(posterior, "E")    #J
E_samples = [E_marginal().item() for _ in range(m)]    #J
E_unique, E_counts = np.unique(E_samples, return_counts=True)    #K
E_probs = E_counts / m    #K

plt.bar(E_unique, E_probs, align='center', alpha=0.5)    #L
plt.xticks(E_unique, E_alias)    #L
plt.ylabel('probability')    #L
plt.xlabel('E')    #L
plt.title('P(E | T = "train") - Importance Sampling')    #L
```

#A We'll use two inference related classes, ImportanceSampling and EmpiricalMarginal.
#B pyro.condition is a conditioning operation on the model.
#C It takes in the model,
#D and evidence for conditioning on. The evidence is a dictionary that maps variable names to values. The need to specify variable names during inference is why we have the name argument in the calls to pyro.sample. Here we condition on T="train".
#E I'll run a inference algorithm that will generate m samples.
#F Namely, I use importance sampling. The Importance class constructs this inference algorithm.
#G It takes the conditioned model.
#H It also takes the number of samples.
#I Run the random process algorithm with the run method. The inference algorithm will generate from the joint probability of the variables we didn't condition on (everything but T) given the variables we conditioned on (T).
#J However, we only care about T, so EmpiricalMarginal operates on the output of algorithm so we obtain only samples of T.
#K Based on these samples, I produce a Monte Carlo estimation of the probabilities in P(E|T="train").
#L Plot a visualization of the learned probabilities.

This produces the plot in Figure 3.13.

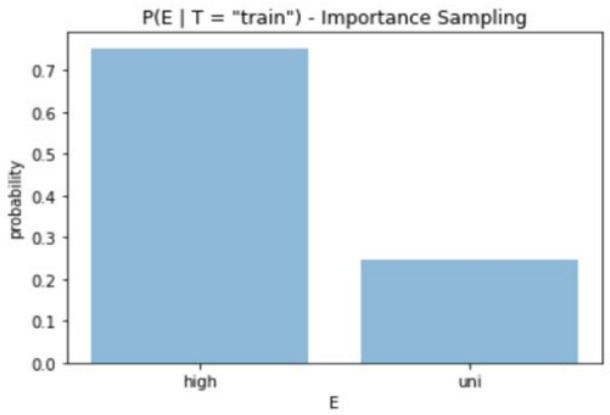


Figure 3.13 Visualization of the P(E|T="train") distribution.

The probabilities in Figure 3.13 are close to the results from the *pgmpy* model, though slightly different, due to different algorithms and the rounding of the parameter estimates to two decimal places.

This probabilistic inference is not yet causal inference. In chapter 8, we'll show how to use probabilistic inference to implement causal inference.

3.4 Your causal question scopes the DAG

Defining the DAG based on the variables in your data is attractive because your DAG has a fixed set of variables; you don't have to wonder about what variables should be in your DAG. But causal modelers model the data generating process, not the data. The true causal structure in the world doesn't care about what happens to be measured in your dataset.

The problem is that, while your data has a fixed set of variables, the variables that could comprise your DGP are only bounded by your imagination. Given a variable, you could include its causes, those causes' causes, those causes' causes' causes, continuing all the way back to Aristotle's "prime mover." Fortunately, there is no need to go back that far. You can use the following procedure to select variables for inclusion in your causal DAG.

INCLUDE VARIABLES CENTRAL TO YOUR CAUSAL QUESTION(S).

Step 1 is to include all the variables central to your causal question. If you intend to ask multiple questions, include all the variables relevant to those questions.

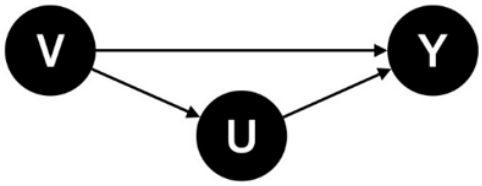


Figure 3.14 Step 1. Include variables central to your causal question(s). Here, suppose you are interested in asking questions about V, U, and Y.

As I discussed in chapter 1, causal effect inference is the most common causal question. As an example, consider Figure 3.14. Suppose that we intend to ask a causal question about V, U, and Y. These become the first variables we include in the DAG.

INCLUDE ANY COMMON CAUSES FOR THE VARIABLES IN STEP 1.

Step 2 is to add any common causes for the variables you included in step 1. To illustrate, you would start variables U, V and Y in Figure 3.14, then trace back their causal lineages and identify shared ancestors. These shared ancestors are common causes. In Figure 3.15, W_0, W_1 and W_2 are common causes of V, U, and Y.

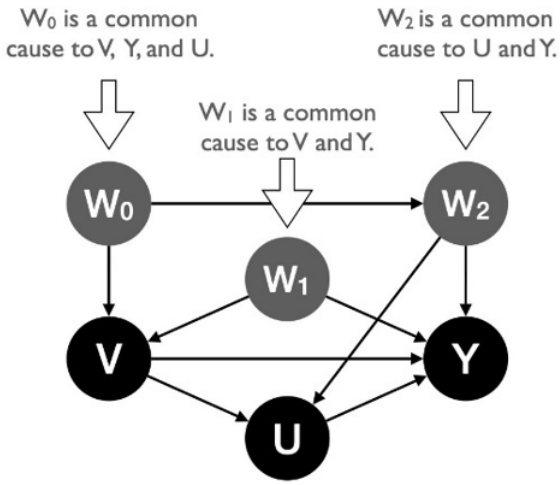


Figure 3.15 Satisfy causal sufficiency; include common causes to the variables from step 1.

In formal terms, a variable is a common cause Z of a pair of variables X and Y if there is a directed path from Z to X that does not include Y and a directed path from Z to Y that does not include X. The formal principle of including common causes is called **causal sufficiency**. A set of variables is causally sufficient if it doesn't exclude any common causes between any pair of variables in the set. Furthermore, once you include a common cause, you don't have to include earlier common causes on the same paths. For example, Figure 3.16 illustrates how we might exclude variables earlier common causes.

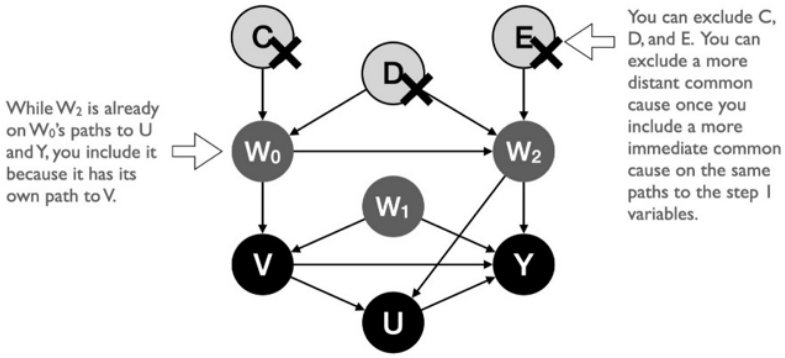


Figure 3.16 Once you include a common cause, you don't have to include any earlier common causes on the same paths to the step 1 variables.

In Figure 3.16, W_2 is on W_0's path to Y and U, but we include W_0 because it has its own path to V. In contrast, while C is a common cause of V, Y, and U, W_0 is on all of C's paths to V, Y and U, so we can exclude it after including W_0. Similarly, W_2 lets us exclude E, and W_0 and W_2 together let us exclude D.

INCLUDE VARIABLES THAT MAY BE USEFUL IN CAUSAL INFERENCE STATISTICAL ANALYSIS

Step 3 is to include variables that may be useful in statistical methods for the causal inferences you want to make. For example, in Figure 3.17, suppose you were interested in estimating the cause effect of V on Y. You might want to include Z so it could function as an "instrumental variable," a type of variable used in certain statistical methods for causal effect inference (we'll cover these in part 4 of this book). X_0 and X_1 could also be of use in the analysis by accounting for other sources of variation in Y.

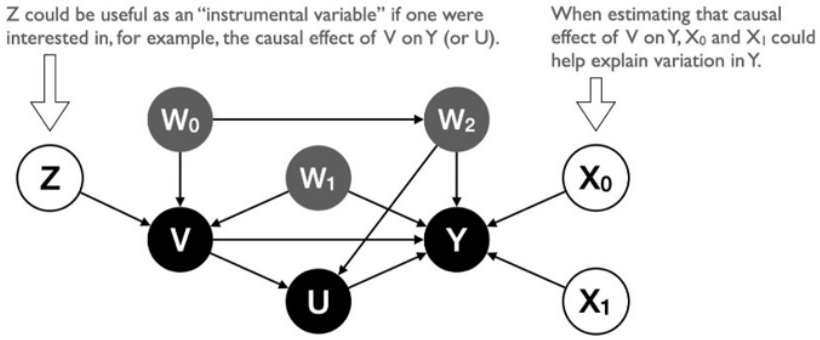


Figure 3.17 Step 3 is to include variables that may be useful in the causal inference statistical analysis.

INCLUDE VARIABLES THAT HELP THE DAG COMMUNICATE A COMPLETE STORY

Step 4 is to include any variables that help the DAG better function as a communicative tool. Consider the common cause D in Figure 3.18.

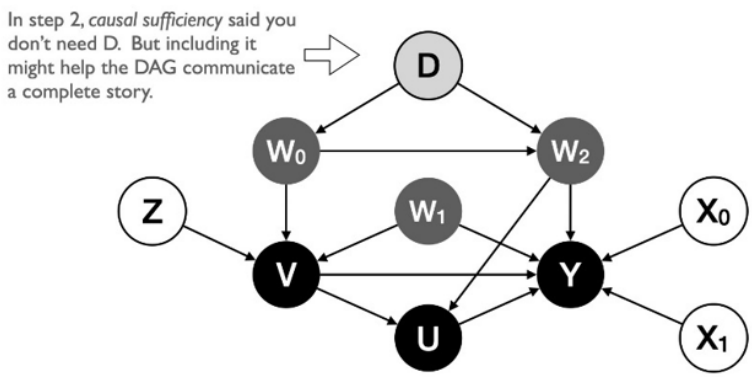


Figure 3.18 Step 4 is to include variables that help the DAG tell a complete story. In this example, despite having excluded D in step 2 (Figure 3.16) we still might want to include D if it is an important variable.

In Figure 3.16, we concluded that common cause D could be excluded after including common causes W_0 and W_2. But perhaps D is an important variable in the domain. If so, including it may help the DAG tell a better story by showing how a key variable relates to the variables you included.

3.4.1 Causal DAGs for causal effect estimation

Building a causal DAG for causal effect inference deserves special attention. When you are interested in causal effects, you have the option of specifying the DAG in terms of the roles certain variables play in causal effect inference. Again, consider the nodes in Figure 3.16. I discussed including some of these nodes because they follow certain roles. W_0, W_1, and W_2 are common causes and Z is an "instrumental variable." Similarly, X_0 and X_1 are potentially useful in causal effect.

We can use those roles to specify a basic DAG. The *dowhy* causal inference library shows us how.

Listing 3.10 Creating a DAG based on roles in causal effect inference

```
from dowhy import datasets

import networkx as nx
import matplotlib.pyplot as plt

sim_data = datasets.linear_dataset(    #A
    beta=10.0,
    num_treatments=1,    #B
    num_instruments=2,    #C
    num_effect_modifiers=2,    #D
    num_common_causes=5,    #E
    num_frontdoor_variables=1,    #F
    num_samples=100,

)
dag = nx.parse_gml(simulated['gml_graph'])    #G
pos = {    #G
 'X0': (600, 350),    #G
 'X1': (600, 250),    #G
 'FD0': (300, 300),    #G
 'W0': (0, 400),    #G
 'W1': (150, 400),    #G
 'W2': (300, 400),    #G
 'W3': (450, 400),    #G
 'W4': (600, 400),    #G
 'Z0': (10, 250),    #G
 'Z1': (10, 350),    #G
 'v0': (100, 300),    #G
 'y': (500, 300)    #G
}    #G
options = {    #G
    "font_size": 12,    #G
    "node_size": 800,    #G
    "node_color": "white",    #G
    "edgecolors": "black",    #G
    "linewidths": 1,    #G
    "width": 1,    #G
}    #G
nx.draw_networkx(dag, pos, **options)    #G
ax = plt.gca()    #G
ax.margins(x=0.40)    #G
plt.axis("off")    #G
plt.show()    #G
```

#A datasets.linear_dataset generates a DAG from the specified variables.
#B I add one treatment variable, like V in Figure 3.18.
#C Z in Figure 3.18 is an example of an instrumental variable; a variable that is a cause of the treatment but its only causal path to the outcome is through the treatment. Here I create two instruments.
#D X_0 and X_1 are in Figure 3.18 are examples of "effect modifiers" that help model heterogeneity in the causal effect. Dowhy defines these as other causes of the outcome (though they needn't be). Here I create two effect modifiers.
#E I add 5 common causes, like the three W_0, W_1, and W_2 in Figure 3.18. Unlike the nuanced structure between in Figure 3.18, the structure here will be simple.
#F Front door variables are on the path between the treatment and the effect, like U in Figure 3.18. Here I add one.
#G This code extracts the graph, creates a plotting layout, and plots the graph.

This code produces the DAG pictured in Figure 3.19.

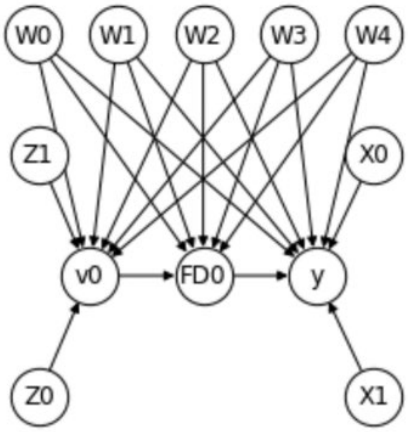


Figure 3.19 A causal DAG built by specifying variables by their role in causal effect inference.

This role-based approach produces a simple template causal DAG. It won't give you the nuance that you have in Figure 3.18, but it will be enough for tackling the pre-defined causal effect query. It's a great tool to use when working with collaborators who are skeptical of DAGs but are comfortable talking about variable roles.

Such a template method could be used for other causal queries as well. You can also use this approach to get a basic causal DAG in a first step that you then build upon to produce a more nuanced graph.

3.5 Causal invariance and parameter modularity

Suppose we were interested in modeling the relationship between altitude and temperature. The two are clearly correlated; the higher up you go, the colder it gets. However, you know temperature doesn't cause altitude, otherwise heating the air within a city would cause the city to fly. Altitude is the cause, and temperature is the effect. So we think of a simple causal DAG that we think captures the relationship between temperature and altitude, along with other causes, and come up with the DAG in Figure 3.20.

Let "A" be altitude, "C" be cloud cover, "L" be latitude, "S" be season, and "T" be temperature. The DAG in Figure 3.14has five causal Markov kernels, {P(A), P(C), P(L), P(S), P(T|A, C, L, S)}. To train your model you need to learn parameters for each of these causal Markov Kernels.

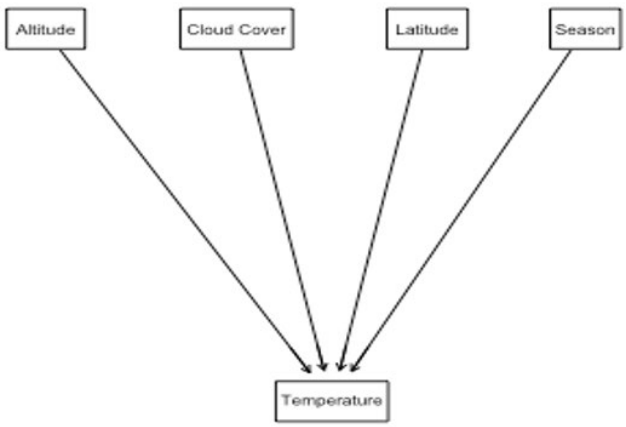


Figure 3.20 A simple model of outdoor temperature

3.5.1 Independence of mechanism and parameter modularity

There are some underlying thermo-dynamic mechanisms in the data generating process underlying these causal Markov kernels. For example, the causal Markov kernel P(T|A, C, L, S) is the conditional probability induced by the physics-based mechanism wherein altitude, cloud cover, latitude, and season drive temperature. That mechanism is distinct from the mechanism that determines cloud cover (according to our DAG). "Independence of mechanism" refers to this distinction between mechanisms.

The independence of the mechanism means that when we train the parameters of P(T|A, C, L, S) and the parameters of P(C), we are learning encodings of those mechanisms. And since those mechanisms are separate, the encodings should be separate. This "parameter modularity" is an important rule to follow when you learn the parameters of a causal graphical model. Parameter modularity is a property of causal graphical models that states that the parameter sets for a causal Markov kernel are unconnected to the parameters of other causal Markov kernels. That "unconnectedness" means when you learn parameters, your learning procedure should treat them as disjoint sets. If you treat the parameters as random variables to learn with Bayesian inference, the parameter sets are a priori independent. Put another way; the training of the parameter sets for each causal Markov kernel should be decoupled. Note that this principle contrasts with most training procedures in machine learning. In machine learning, a loss function is typically defined on the full vector of parameters, and they are optimized jointly.

3.5.2 Causal invariance and domain transfer

You may not be a climatologist or a meteorologist. Still, you know the relationship between temperature and altitude has something to do with air pressure, turbulence and sunlight and such. You also know that whatever the physics of that relationship is, that

physics is the same in Katmandu as it is in El Paso. So, when we train a causal Markov kernel on data solely collected from Katmandu, we learn a causal representation of a mechanism that is invariant between Katmandu and El Paso. This invariance helps with transfer learning; we should be able to use that trained causal Markov kernel to make inferences about El Paso.

Methods that use causal invariance learn components of a causal model on one dataset and use it in making inference about others are generally called causal transfer learning or causal data fusion. For two data sets with two data generating process, these methods have the modeler specify which components of the data generating processes overlap and then use that overlap to allow using one data set to make inferences about the other.

3.5.3 Fitting parameters with common sense

In the temperature model, we have an intuition about the physics of the mechanism that induces P(T|A, C, L, S). However, we rely on the same assumptions for models from non-natural science domains such as econometrics and other social sciences. In these domains, we still assume the causal Markov kernels correspond to distinct causal mechanisms in the real world, assuming the model is true. For example, recall P(T|O, R) in our transportation model. We still assume the underlying mechanism is distinct from the others; if there were some changes to the mechanism underlying P(T|O, R), only P(T|O, R) should change; other kernels in the model should not. If something changes the mechanism underlying P(R|E), the causal Markov kernel for R, this change should affect P(R|E) but have no effect on the parameters of P(T|O, R).

This invariance can help us estimate parameters without statistical learning by reasoning the underlying causal mechanism. For example, let's look again at causal Markov kernel P(R | E) (recall R is residence, E is education).

People who don't get more than a high school degree are more likely to stay in their hometowns. However, people from small towns who attain college degrees are likely to move to a big city where they apply their credentials to get higher-paying jobs.

From this, we might reason about US demographics. Suppose some web search tells you that 80% of the US lives in an urban area (P(R=big) = .8), while 95% of college degree holders live in an urban area (P(R=big|E=uni) = .95). Further, 25% of the overall adult population has a university degree (P(E=uni) = .25). Then, with a bit of math, you calculate your probability values as P(R=small|E=high)=.25, P(R=big|E=high) = .75, P(R=small|E=uni) = .05, P(R=big|E=uni) = .95. The ability to calculate parameters in this manner is particularly useful if data available for parameter learning.

3.6 Looking ahead: model validation and combining causal graphs with deep learning

The Markov property will continue to be of use to us. In the next chapter, we'll examine how to use the property to validate, or more accurately "refute," our causal DAG.

In section 3 of this chapter, we examined how to build a probabilistic machine learning model on top of the causal DAG. For each node in the graph, we constructed a causal Markov kernel (probability distribution of the node given its direct causes in the DAG). We

used simple conditional probability tables to represent these kernels. We used pgmpy's implementation of the structural expectation maximization algorithm to estimate the causal Markov kernels of latent variables as well.

The Markov property allows us to extend this technique from conditional probability tables to deep learning. For each causal Markov kernel, we could replace the conditional probability table with probability parameters with a feedforward neural network (causal parents as input and child as output) with weight parameters. We can use the independence of mechanism assumption to dramatically simplify the search space of those parameters. We can also rely on deep learning's ability to learn latent representations to train the model in the presence of latent variables. Finally, we can make use of deep learning's ability to work with low-level features in enable causal modeling of rich media objects such as images. We'll examine this ability in chapter 5.

3.7 Summary

- The causal DAG is a directed acyclic graph that can represent our causal assumptions about the data generating process.
- The causal DAG is a useful tool for visualizing and communicating your causal assumptions.
- DAGs are fundamental data structures in computer science and admit many fast algorithms we can bring to bear on causal inference tasks.
- DAGs link causality to conditional independence via the causal Markov property.
- DAGs can provide scaffolding for probabilistic ML models.
- We can use various methods for statistical parameter learning to train a probabilistic model on top of a DAG. These include maximum likelihood estimation and Bayesian estimation.
- Given the causal DAG, the modeler can choose from a variety of parameterizations of the causal Markov kernels in the DAG, ranging from conditional probability tables to regression models to neural networks.
- A causally sufficient set of variables contains all common causes between pairs in that set.
- You can build a causal DAG by starting with a set of variables of interest, expanding that to a causally sufficient set, adding variables useful to causal inference analysis, and finally adding any variables that help the DAG communicate a complete story.
- You can specify a DAG by the roles variables play in a specific causal inference task.
- The property of parameter modularity means parameter sets can often be estimated based on common sense alone, without dat.
- When using data, the parameter sets for each causal Markov kernel should be learned independently from one another.

4

Testing the DAG with causal constraints

This Chapter Covers

- Using d-separation to reason about how causality constrains conditional independence
- Using networkx and pgmpy to do d-separation analysis
- Refuting a causal DAG using conditional independence tests
- Refuting a causal DAG using Verma constraints

Causality in the data generating process induces constraints, such as conditional independence, on the joint probability distribution of the variables in that process. We saw a flavor of these constraints in the previous chapter in the form of the Markov property, how effects become independent of indirect causes given their direct causes. These constraints give us the ability to test our model against the data; if the causal DAG we build is correct, we should see evidence of these constraints in the data.

In this chapter, we'll use statistical analysis of the data to test our causal DAG. Namely, we'll try to *refute* our causal DAG; meaning we'll look for ways the data suggests our causal DAG is wrong. In this chapter we learn to test our causal DAG using conditional independence tests and an extension of conditional independence called Verma constraints that we can test when variables in our causal DAG are not observed in data.

To start, we look at the concept of d-separation. D-separation tells us what conditional independence constraints should hold given our causal DAG, and it is the keystone of graphical causal inference analysis.

4.1 Examples of how causality induces conditional independence

Consider again the blood type example in Figure 4. 1. The example illustrates how causality induces conditional independence. Your father's blood type is a direct cause of yours, and you paternal grandfather's blood type is an indirect cause. Despite being a cause of your blood type, your paternal grandfather's blood type is conditionally independent of your blood type given your father's.

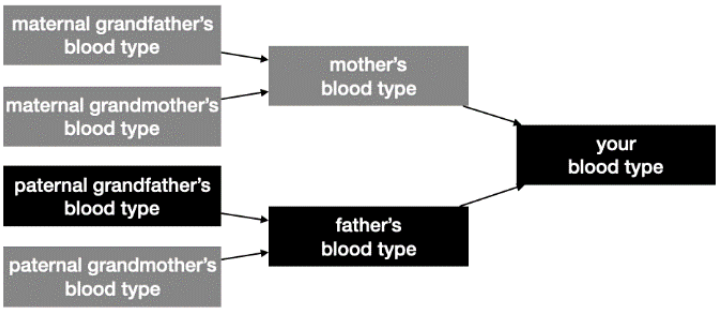


Figure 4. 1 Causality induces conditional independence. Your blood type is conditionally independent of your paternal grandfather's blood type (an indirect cause) given your father's blood type (a direct cause)

We know this from causality; the parents' blood types completely determine the blood type of the child. So your paternal grandfather's and grandmother's blood type completely determined your father's blood type. But your father's and mother's blood type completely determined yours. So once we know your father's blood type, there is nothing more your paternal father's blood type can tell us.

4.1.1 Colliders

Now we consider the **collider**, the interesting way in which causal induces conditional independence. Consider the canonical example in Figure 4. 2. The sprinkler being on and whether or not it rains are causes of whether the grass is wet or not. Knowing that the sprinkler is off won't help you predict whether it's raining. In other words, the. sprinkler state and rain state are independent. But when you know the grass is wet, also knowing that the sprinkler is off tells you it *must* be raining. So while the sprinkler state and rain state are independent, they become conditionally dependent given the state of the grass.

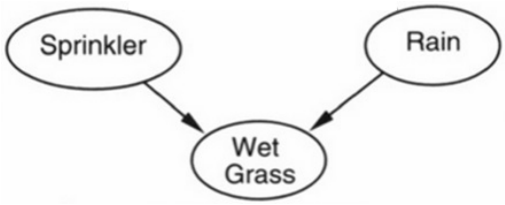


Figure 4. 2 The sprinkler being on and whether or not it rains causes the grass to be wet or not. Knowing that the sprinkler is off won't help you predict whether it's raining; i.e. sprinkler state and rain state are independent. But given the grass is wet, knowing the sprinkler is off tells you it must be raining, i.e. sprinkler state and rain start are conditionally dependent given the state of the grass.

In this case "Wet Grass" is a collider, an effect with at least two independent causes. Colliders are odd and interesting because they illustrate how causal variables can be independent but then become dependent when one conditions on a shared effect variable. In conditional independence terms, the parent causes are independent (Sprinkler $\perp$ Rain), but become dependent after we observe (condition on) the child (Sprinkler $\not\perp$ Rain | Wet Grass).

For another example, let's look at blood type again in Figure 4. 3.

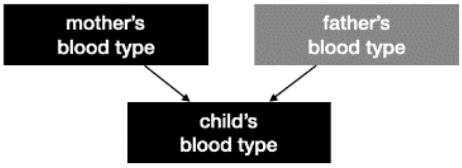


Figure 4. 3 Mothers and fathers are generally unrelated, and thus knowing mother's blood type can't help predict father's blood type. But if we know the mother's blood type and the child's blood type, it narrows down the possible blood types of the father.

Since the mother and father are unrelated, the mother's blood type tells us nothing about the father's blood type - (mother's blood type $\perp$ father's blood type). Suppose we only know the child's blood type.

Let's suppose the child's blood type is B. Does that help us predict the father's blood type? Without getting into probabilities, we can see that the child is a B doesn't narrow down our possibilities for the father. If the child is B, then the father can be A, B, AB, or O. To see this, examine the standard blood type table in Figure 4. 4.

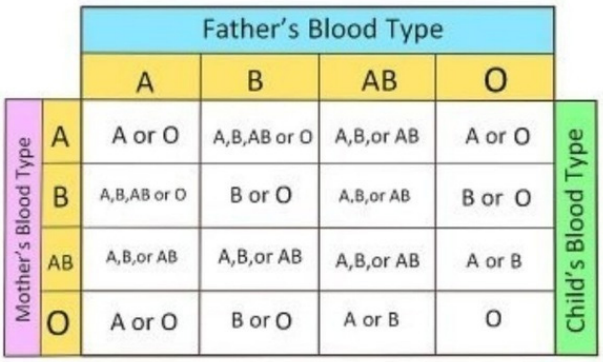


Figure 4. 4 Knowing mother's blood type can help you narrow down the father's blood type if you know the child's blood type.

Each column is a possible outcome for the father, and each of those columns contains cases where the child is a B.

But if, in addition to knowing the child is a B, we know the mother was an A, then we can narrow down the possibilities for the father. We know the father can't be an A or an O, he must be a B or an AB.

So (mother's blood type $\perp$ father's blood type), (mother's blood type $\not\perp$ father's blood type | child's blood type).

In summary, colliders are weird because they describe how causal logic leads to situations where two things are independent but "suddenly" become dependent when you condition on a third variable.

4.1.2 Domain-free reasoning with a causal graph

The problem with the above explanation is that it is in terms of a specific domain. If we want to write code that can help us make causal inferences about different domains, we need a domain-independent way of mapping causal relationships to conditional independence. "D-separation" solves this problem.

"D-separation and connection" refers to how we use graphs to reason about conditional independence. The concept is novel at first glance. But it is your one of your most important tools for graph-based causal reasoning. As a bit of a spoiler for chapter 8, consider the problem of causal effect inference, illustrated in Figure 4. 5.

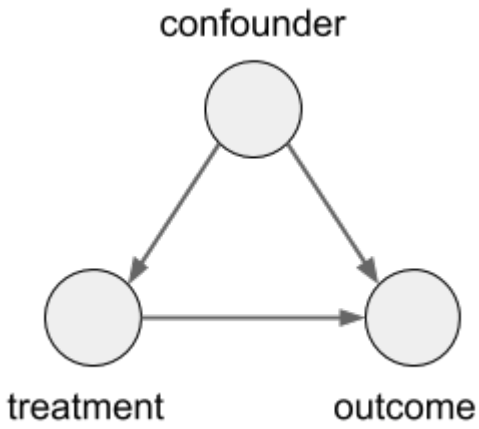


Figure 4. 5 In causal effect inference, we are interested in statistical quantifying how much a cause (treatment) affects an effect (outcome). To do this we have to adjust for non-causal correlation introduced by one or more confounders. D-separation is the backbone of the theory that tells us how.

To estimate the causal effect of the treatment on the outcome, we have to adjust for the confounder. The theoretical justification for doing so is based on "d-separating" the path "treatment ← confounder → outcome" and zooming in on the "treatment → outcome" path.

4.2 D-separation and conditional independence

Recall the following ideas from previous chapters:

1. The causal DAG is a model of the data-generating process.
2. The data-generating process entails a joint probability distribution.
3. Causal relationships induce some variables in the joint probability distribution to be independent and conditionally independent.

D-separation/connection is a graphical language for reasoning about that conditional independence in the joint probability distribution the causal DAG models. The concept refers to nodes and paths in the causal DAG; nodes/paths are "d-connected" or "d-separated." The idea is for a statement like "these nodes are d-separated in the graph" to correspond to a statement like "these variables are conditionally independent."

We want to make this correspondence because reasoning about over graphs is easier than reasoning about probability distributions directly; tracing paths between nodes is easier than taking graduate-level classes in probability theory. Also, recall from chapter 2, that graphs are fundamental to algorithms and data structures, and that statistical modeling benefits from making conditional independence assumptions.

4.2.1 What is d-separation?

D-separation/connection is about making statements about a graph. This statement involves a pair of nodes u and v, and a set of one or more third-party nodes that does not include u and v. Let bold-face letter **Z** represent that set of third-party nodes. The statement uses the verbs "d-separate" and "d-connect" to talk about u, v, and **Z**. The statements are either true or false. So I could ask you, "Hey bud, does set **Z** d-separate nodes v and u in graph G?" and you could either answer "Yes indeed it does!" or "No, I'm afraid it does not."

Specifically, a d-separation statement reads as follows: "Nodes u and v are d-separated by the set of nodes **Z** in graph G. In notation, we write: $u \perp_G v|\boldsymbol{Z}$. In the expression $u \perp_G v|\boldsymbol{Z}$, the subscript "G" on the "$\perp$" distinguishes it from $u \perp v|\boldsymbol{Z}$ which means "u and v are conditionally independent given set **Z.**" D-separation/connection does not rely on order. If u is d-separated from/d-connected to v by **Z**, then v is d-separated from/d-connected to u by **Z**.

D-connection is the opposite of d-separation. So if u and v are d-separated by Z, then they are not d-connected. If u and v are d-connected, then they are not d-separated.

One might ask that since the causal graph is a model of the data generating process, why do we need to distinguish between $u \perp_G v|\boldsymbol{Z}$ and $u \perp v|\boldsymbol{Z}$? Firstly, d-separation is not unique to causal graphs, it is common across graphical modeling including models that give no causal interpretation of the data generating process. Secondly, the causal DAG is a model, and our model could be wrong. Indeed, in this chapter we'll see test if our model is wrong by seeing if a d-separation like $u \perp_G v|\boldsymbol{Z}$ corresponds to actual statistical evidence of $u \perp v|\boldsymbol{Z}$ in the data.

FOUR CONDITIONS FOR D-SEPARATION

Let P be path, meaning a series of edges between two nodes. It does not matter if the nodes on the paths are observed or not in your data (we'll see how the data factors in later). It is okay for the path to have mixed edge directions, for example i ← m → j is a path between nodes i and j where the two edges on the (← , →) are not oriented in the same direction.

A path P is d-separated by node set Z if any of four conditions are met.

- P contains a chain, i → m → j, such that the middle node m is in **Z**
- P contains a chain, i ← m ← j, such that the middle node m is in **Z**
- P contains a child-parent-child structure i ← m → j, such that the middle (parent) node m is in **Z**
- P contains the "collider," i → m ← j, such that the middle node m is not in **Z**, and no descendant of m is in **Z**.

Two nodes u and v are said to be d-separated by Z if all paths between them are d-separated.

A synonym for "d-separates" is "blocks". So, if a set **Z** d-separates u and v, then "Z blocks all paths from u to v". You can also say, "**Z** is a blocker for u and v".

COLLIDERS MAKE D-SEPARATION WEIRD

Colliders (three-node motifs of the form i → m ← j are what make d-separation a bit tricky to grok at first glance. Figure 4. 6 illustrates how colliders affect d-separation. Later in this chapter, we'll see the import causal intuition behind colliders.

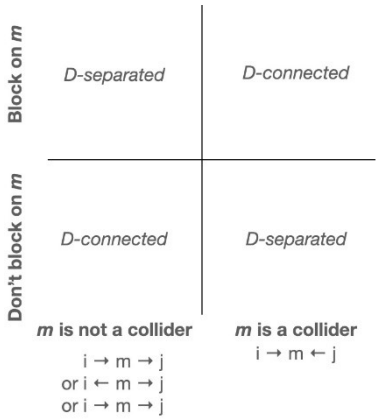


Figure 4. 6 Colliders make d-connection tricky. Given a node m on a path, if m is not a collider, the path is d-connected by default and d-separated when you block on m. If m is a collider, the path is d-separated by default and d-connected when you block on m.

- All paths between two nodes d-connect by default *unless that path has a collider*. A path with a collider is d-separated by default.
- Blocking with any node on a d-connected path will d-separate that path *unless that node is a collider*. Blocking on a collider will d-connect a path by default, as will blocking with a descendant of that collider.

In practice, we ask if two nodes in the graph are d-separated or d-connected. It will always be easier to break that question down in terms of d-separating paths. Look at all the paths between the two nodes and ask yourself if the path is d-separated or blocked by some set Z.

D-SEPARATING SETS OF NODES

D-separation doesn't just apply to pairs of nodes, it applies to pairs of sets of nodes. In the notation , Z can be a set of blockers. U and V can be sets as well. We d-separate two sets by blocking all d-connected paths between members of each set. Other graph-based causal ideas, such as the do-calculus, also generalize to sets of nodes. If you remember that fact, we can build intuition on individual nodes and that intuition will generalize to sets.

4.2.2 Examples of d-separation

A helpful analogy for understanding d-separation is an electronic circuit. Paths without colliders are d-connected and are like closed circuits where electrical current flows uninhibited. "Blocking" on a node on that path d-separate the path and will "break the circuit" so current can't flow. We can think of a collider like an open switch. An open switch blocks current flow in an electronic circuit. When a path has a collider, the collider blocks all current from passing through it. Colliders break the circuit. Blocking on a collider is like

closing the switch, and the current that couldn't pass through before now can pass through (d-connection). Let's work through some examples with some illustrations.

EXAMPLE 1

Consider the DAG in Figure 4. 7 where P is u → i → m → j → v.

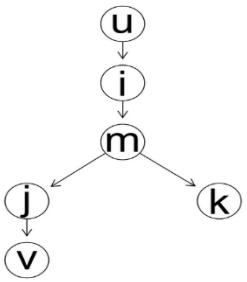


Figure 4. 7 Does the set {m, k} d-separate path u→i→m→j→v?

This path is d-connected by default. Now let Z be the set {m, k}. P contains a chain i → m → j, and m is in Z. If we block on Z the first condition is satisfied and u and v are d-separated. Blocking on Z (specifically, blocking on m, which is in Z) "breaks the circuit" as is illustrated in Figure 4. 8.

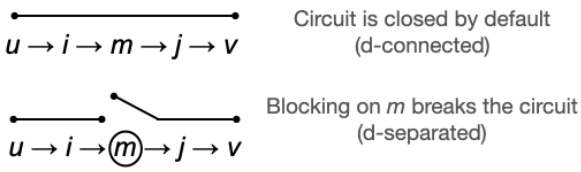


Figure 4. 8 The path is d-connected by default but blocking on m ∈ Z d-separates the path and figuratively breaks the circuit ("∈" means "in").

EXAMPLE 2

Now consider the DAG in Figure 4. 9, where P is u ← i ← m → j → v.

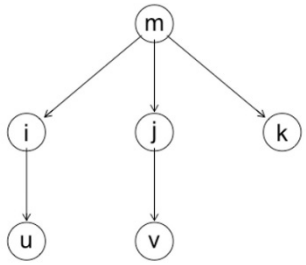


Figure 4. 9 Does the set {m} d-separate path u←i←m→j→v?

This path is also d-connected by default. Note that d-connection can go against the grain of causality. In Figure 4 - 2, the d-connected path from u to v takes steps in the direction of causality; u to i (u → i) then i to m (i → m) then m to j (m → j) then j to v (j → v). But here, we have two *anticausal* (meaning against the direction of causality) steps, namely the step from u to i (u ← i) and i to m (i ← m).

Suppose we block on set Z and Z contains only the node m. Then condition 3 is satisfied and the path is d-separated, as illustrated in Figure 4. 10.

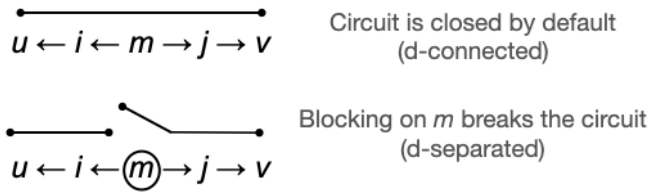


Figure 4. 10 This path from u to v is also d-connected by default even though it has some steps (u to I and i to m) that go against the direction of causality. Again, blocking on m ∈ Z d-separates the path and figuratively breaks the circuit.

MINIMAL D-SEPARATING SETS

When the blocking set Z is the singleton set {m}, this set is sufficient to block the paths u → i → m → j → v in example 1 and u ← i ← m → j → v in example 2. Altogether, the sets {i}, {m}, {j}, {i, m}, {i, j}, {m, j}, and {i, m, j} all d-separate u and v on these two paths. However, {i}, {m}, {j} are the minimal d-separating sets, meaning that all the other d-separating sets are composed of these sets. When reasoning about d-separation and when implementing it in algorithms, we want to focus on minimal d-separating sets. We want to know the minimal requirements for blocking a path.

EXAMPLE 3

In the DAG in Figure 4. 11, is the path u→i→m←j→v d-connected by default?

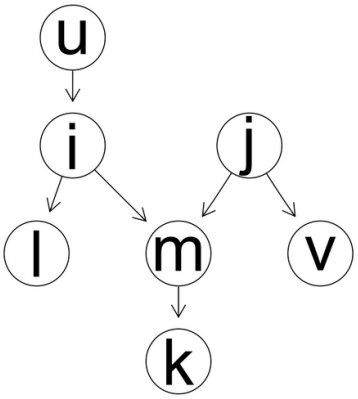


Figure 4. 11 Does the set {m} (or {k} or {m, k}) d-separate path u→i→m←j→v?

No, because the path contains a collider structure m (i→m←j). Paths with colliders are d-separated by default because, according to our circuit analogy, the collider behaves like a resister that blocks all current from passing through.

Now consider what would happen if the blocking set Z included m. In this case, condition 4 is violated and the path *becomes d-connected* as in Figure 4. 12.

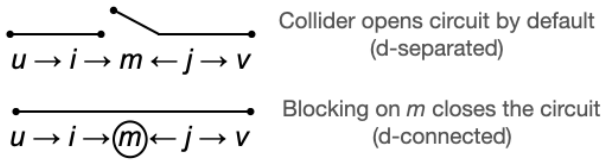


Figure 4. 12 This path from u to v is d-separated by default because it contains a collider m. Blocking on m or any of its descendants d-connects the path and figuratively closes the circuit.

The path would also become d-connected if Z didn't have m but just had k (or if Z included both m and k). Blocking on a decedent of a collider d-connects in the same manner as blocking on a collider.

Can you guess why? It's because the collider's decedent is *d-connected to the collider*. In causal terms, we saw how given your father's blood type, observing your blood type (the collider) might reveal your mother's blood type. Similarly, if instead of observing your blood type we observed your child's blood type, that might help narrow down your blood type and thus narrow down your mother's blood type.

EXAMPLE 4

U and V are d-connection in Figure 4. 13. What sets of nodes are fully required to d-separate U and V?

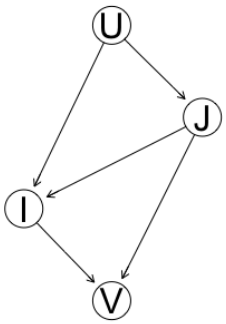


Figure 4. 13 You can d-separate U and V with {U, V}.

In Figure 4. 13, U and V are d-connected through the paths;

- U → I → V
- U → J → V

Again, once d-separate two nodes once we block all the paths between them. So we can block on {I, J}. Are there any other d-connecting paths between U and V initially? Yes, namely U → J → I → V, but blocking on {I, J} has already blocked that path.

EXAMPLE 5

How do we d-separate U and V in Figure 4. 14?

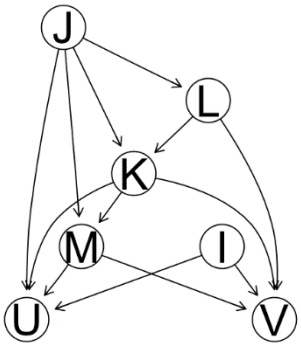


Figure 4. 14 We can d-separate U and V with sets {I, M, K, J} or {I, M, K, L}.

Let's first enumerate three of the shortest paths.

- U ← I → V
- U ← M → V
- U ← K → V

We'll need to block on at least on {I, M, K} to d-separate these three paths. Note that U has another parent J, and there are several paths from U to V through J, but the only one we haven't already d-separated is U ← J ← L → V. Both J and L will block that path. So we could d-separate U and V with minimal sets {I, M, K, J} or {I, M, K, L}.

EXAMPLE 6

The graph in Figure 4. 15 is simple enough that we can enumerate all of the paths.

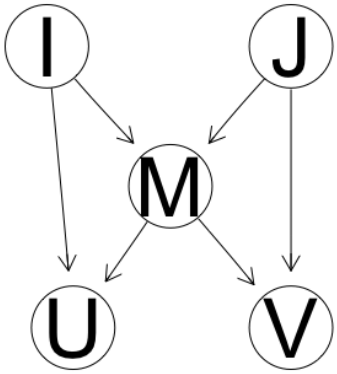


Figure 4. 15 Blocking with M will block the path U ←M→V, but would d-connect the path U←I→ M←J→V because M is a collider between I and J. So we need to additionally block on either I or J to d-separate U←I→ M←J→V.

Let's start with the three d-connecting paths:

- U ← M → V
- U ← I → M → V
- U ← M ← J → V

The easiest way to block all three of these d-connected paths with one node is to block on M. However, note that U ← I → M ← J→ V is not a d-connecting path because M is a collider on that path. If we block on that collider, that path d-connects. So we need to additionally block on I or J. In other words, our minimal d-separating sets are {I, M} and {J, M}.

4.2.3 D-separation in code

If, after that introduction, you are still hazy on d-separation, do not fret. The best approach is to practice using a graph library that implements d-separation. Let's verify our d-separation analysis of the causal DAG in Figure 4. 15.

Listing 4.1 D-separation analysis of the DAG in Figure 4. 15

```
from networkx import d_separated     #A
from pgmpy.base import DAG    #B
dag = DAG(    #B
    [    #B
      ('I', 'U'),    #B
      ('I', 'M'),    #B
      ('M', 'U'),    #B
      ('J', 'V'),    #B
      ('J', 'M'),    #B
      ('M', 'V')    #B
    ]    #B
)    #B
print(d_separated(dag, {"U"}, {"V"}, {"M"}))    #C
print(d_separated(dag, {"U"}, {"V"}, {"M", "I", "J"}))    #D
print(d_separated(dag, {"U"}, {"V"}, {"M", "I"}))    #E
print(d_separated(dag, {"U"}, {"V"}, {"M", "J"}))    #E
```

#A The graph library networkx implements the d-separation algorithm for networkx graph objects such as DiGraph (directed graph).
#B DAG is a base class for the BayesianNetwork class. The base class for DAG is networkx's DiGraph. So d_separated will work on objects of the class DAG (and BayesianNetwork).
#C Builds the graph in Figure 4 - 11.
#C Blocking on a collider M blocks the path U ← M → V but will d-connect the path U ← I → M ← J → V. So this will print False.
#D Blocking on M will block U ← M → V and open (d-connect) U ← I → M ← J → V, but we can block that path with I and J. So this evaluates to True.
#E But blocking on both I and J is overkill, the minimal d-separating sets are {"M", "I"} and {"M", "J"}.

```
False
True
True
True
```

In *pgmpy* there is also a `get_independencies` method in the `DAG` class that enumerates minimal d-separating states that are true given a graph.

Listing 4.2 Enumerating d-separations in pgmpy

```
from pgmpy.base import DAG
dag = DAG(
    [
      ('I', 'U'),
      ('I', 'M'),
      ('M', 'U'),
      ('J', 'V'),
      ('J', 'M'),
      ('M', 'V')
    ]
)
dag.get_independencies()    #A
```

#A Obtain all of the minimal d-separations statements that are true in the DAG.

```
(I ⊥ J)
(I ⊥ V | J, M)
(I ⊥ V | J, U, M)
(V ⊥ I, U | J, M)
(V ⊥ U | I, M)
(V ⊥ I | J, U, M)
(V ⊥ U | J, M, I)
(J ⊥ I)
(J ⊥ U | I, M)
(J ⊥ U | I, M, V)
(U ⊥ V | J, M)
(U ⊥ J, V | I, M)
(U ⊥ V | J, M, I)
(U ⊥ J | I, M, V)
```

pgmpy is efficient in the representation of the d-separation statements. For example, the statements in this graph (V ⊥ I| J, M) and (V ⊥ U | J, M) are both true. But *pgmpy* compresses them into one statement (V ⊥ I, U | J, M).

4.2.4 Don't conflate d-separation with conditional independence

Recall that for conditional independence, we use the notation (X ⊥ Y | Z), and for d-separation, we use (X $\perp_G$ Y | Z). The printed results of the `get_independencies` function does not use the "$\perp_G$" notation. That is unfortunate, because the distinction between d-separation and conditional independence is important. Do not conflate d-separation in the causal graph with conditional independence in the joint probability distribution entailed by the data generating process the graph is meant to model.

The distinction is important for the task of **refuting** your causal DAG. If your causal model is good, then then d-separation in the DAG *implies* conditional independence in the joint distribution (note, the converse is *not* true). In the next sections, we learn how to test if those implications are supported by statistical evidence ofin the data. If the statistical evidence of conditional independence in the data refutes the implications of d-separation in your causal DAG, it is time to return to the drawing board.

4.3 Refuting the Causal DAG

We have seen how to build a causal DAG. Now we evaluate the causal DAG against the data. Of course, we want to find a causal model that fits the data well. We can always use standard goodness-of-fit and predictive statistics to evaluate fit. But here, we're going to focus on *refuting* our causal DAG.

Statistical models fit curves and patterns in the data. There is no "right" model, there are just models that fit the data well. In contrast, causal models go beyond the data to make statements about the data generating process. Those statements are either true or false. As causal modelers, we try to find a model that fits well, but we also try to refute our model by testing the veracity of the statement.

D-separation is our first tool for refutation. You built a causal DAG and (except in trivial cases) it implies conditional independence. If your causal DAG is correct, you should see

statistical evidence of conditional independence in data. Evidence of dependence where your model says there should be statistical independence refutes your model. When this happens, you go back to the drawing board, and build a better model. In causal modeling, you iterate on your model until you can no longer refute it with the data at your disposal. Only then do you optimize the model for fit or prediction; a causal model that is wrong but fits/predicts well is no good for causal inferences.

4.3.1 Revisiting the causal Markov property

Recall that we saw two aspects of the causal Markov property:

- Local Markov property: A node is conditionally independent of its non-descendants given its parents.
- Markov factorization property: The joint probability distribution factorizes into conditional distributions of variables given their direct parents in the causal DAG.

Now we introduce a third face of this property called the Global Markov property. This property states that d-separation in the causal DAG maps to conditional independence in the joint probability distribution. In notation, we write:

$$U \perp_G V|\boldsymbol{Z} \Rightarrow \boldsymbol{U} \perp \boldsymbol{V}|\boldsymbol{Z}$$

In plain words that notation reads as "If U and V are d-separated by Z in the graph G, then they are conditionally independent.[1]" Note that if any of the three facets of the causal Markov pare true, they are all true.

The global Markov property gives us a straightforward way to refute our causal model. We can use d-separations to specify statistical tests for the conditional independence. Failing tests refute the model.

4.3.2 Refutation using conditional independence tests

pgmpy and other libraries make it relatively easy to run conditional independence tests. Let's revisit the transportation model, shown again in Figure 4. 16.

[1] Again, this is not the same as the converse statement, that conditional independence in the joint distribution implies d-separation in the graph. This assumption is called "faithfulness" and isn't testable. Violations of faithfulness assumptions matter more in causal discovery, which we'll cover in chapter 9. Even then, in terms of assumptions violations, you tend to worry more about other things than faithfulness.

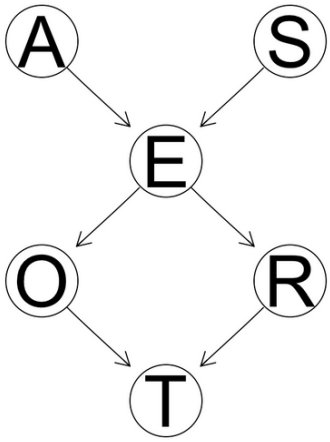


Figure 4. 16 The transportation model. Age (A) and Gender (S) determine Education (E). Education causes Occupation (O) and Residence (R). Occupation and Residence cause Transportation (T).

Recall that for our transportation model we were able to collect the following observations:

- **Age (A):** Recorded as *young* (**young**) for individuals up to and including 29 years, *adult* (**adult**) for individuals between 30 and 60 years old (inclusive), and *old* (**old**) for people 61 and over.
- **Gender (S):** The self-reported gender of an individual, recorded as *male* (**M**) or *female* (**F**).
- **Education (E):** The highest level of education or training completed by the individual, recorded either *high school* (**high**) or *university degree* (**uni**).
- **Occupation (O):** *Employee* (**emp**) or a *self-employed* (**self**) worker.
- **Residence (R):** The population size of the city the individual lives in, recorded as *small* (**small**) or *big* (**big**).
- **Travel (T)**: The means of transport favored by the individual, recorded as car (car), train (train) or other (other)

In the graph, $E \perp_G T \mid O, R$. So let's test if $E \perp T \mid O, R$. Statistical hypothesis tests have a null h**ypothesis** (denoted $\boldsymbol{H}_0$) and an **alternative hypothesis** (denoted $\boldsymbol{H}_a$). For statistical hypothesis tests of conditional independence, it is standard that the null hypothesis $\mathbf{H}_0$ is the hypothesis of conditional independence and $\mathbf{H}_a$ is the hypothesis that they are dependent.

A statistical hypothesis test uses the N data points of observed values of U, V, and Z (from an exploratory data set) to calculate a statistic.

Listing 4.3 Loading the transportation data

```
Import pandas as pd
survey_url =
       "https://raw.githubusercontent.com/altdeep/causalML/master/datasets/transportation_s
       urvey.csv"
fulldata = pd.read_csv(survey_url)

# Create exploratory data set
data = fulldata[0:30]    #A
print(data[0:5])
print(data.shape)
```

#A Subsetting the data to only 30 datapoints.

```
       A  S     E    O      R      T
0  adult  F  high  emp  small  train
1  young  M  high  emp    big    car
2  adult  M   uni  emp    big  other
3    old  F   uni  emp    big    car
4  young  F   uni  emp    big    car
(30, 6)
```

Most conditional independence testing libraries will implement *frequentist* hypothesis tests. These tests will conclude in favor of H_0 or H_a depending on whether a given statistic falls above or below a certain threshold. "Frequentist" in this context means that the statistic produced by the test is called a p-value, and the threshold is called a *significance level*, which by convention is usually (usually .05 or .1).

The test favors the null hypothesis H_0 of conditional independence if the p-value falls above the significance threshold and the alternative hypothesis H_a if it falls below the threshold. This frequentist approach is an optimization that guarantees the significance level is an upper bound on the chances of concluding in favor of dependence when E and T are actually conditional independent.

Most software libraries provide conditional independence testing utilities that make specific mathematical assumptions when calculating a p-value. For example, I run the specific conditional independence test that mathematically derives a test statistic that theoretically follows the Chi-squared probability distribution, then uses this assumption to derive a p-value. The following code runs the test.

Listing 4.4 Chi-squared test of conditional independence

```
from pgmpy.estimators.CITests import chi_square    #A
significance = .05    #B
result = chi_square(    #C
   X="E", Y="T", Z=["O", "R"],    #C
   data=data,    #C
   boolean=False,    #C
   significance_level=significance    #C
)    #C
print(result)
```

#A Import the chi square test function.
#B Set the significance level to .05.
#C When the `boolean` argument is set to false, the test returns a tuple of three elements. The first two are the Chi-square statistic and the corresponding the p-value of 0.56. The last element is a Chi-square distribution parameter called *degrees of freedom*, which is needed to calculate the p-value.

```
(1.1611111111111112, 0.5595873983053805, 2)
```

The p-value is greater than the significance level, so this test favors the null hypothesis of conditional independence. In other words, this particular test did not offer falsifying evidence against our model.

I can jump directly to the result of the test by setting the `chi_square` function's `boolean` argument to `True`. The function will then return `True` if the p-value is greater than the significance value (favoring conditional independence) and `False` otherwise (favoring dependence).

Listing 4.5 Chi-square test with Boolean outcome

```
from pgmpy.estimators.CITests import chi_square    #A
significance = .05    #B
result = chi_square(    #C
   X="E", Y="T", Z=["O", "R"],    #C
   data=data,    #C
   boolean=True,    #C
   significance_level=significance    #C
)    #C
print(result)
```

#A Import the chi square test function.
#B Set the significance level to .05.
#C When the `boolean` argument is set to True, the test returns a simple True or False outcome. It will return True if the p-value is greater than the signficance value, which favors conditional independence. It returns False otherwise, favoring dependence.

```
True
```

Now, let's iterate throw all the d-separation statements we can derive from the transportation graph, and test them one-by-one. The following script will print each d-separation statement along with the outcome of the corresponding conditional independence test.

Listing 4.6 Run a chi-squared test for each d-separation statement

```
from pprint import pprint

from pgmpy.base import DAG
from pgmpy.independencies import IndependenceAssertion

dag = DAG(
    [
      ('A', 'E'),
      ('S', 'E'),
      ('E', 'O'),
      ('E', 'R'),
      ('O', 'T'),
      ('R', 'T')
    ]
)
dseps = dag.get_independencies()

def test_dsep(dsep):
    test_outputs = []
    for X in list(dsep.get_assertion()[0]):
       for Y in list(dsep.get_assertion()[1]):
           Z = list(dsep.get_assertion()[2])
           test_result = chi_square(
                               X=X, Y=Y, Z=Z,
                               data=data,
                               boolean=True,
                               significance_level=significance
           )
           test_outputs.append((IndependenceAssertion(X, Y, Z), test_result))
    return test_outputs

results = [test_dsep(dsep) for dsep in dseps.get_assertions()]
results_flat = [item for sublist in results for item in sublist]
results = {k: v for k, v in results_flat}
pprint(results)


{(O ⊥ A | R, E, T, S): True,
 (S ⊥ R | E, T, A): True,
 (S ⊥ O | E, T, A): True,
 (T ⊥ S | R, O, A): True,
 (S ⊥ O | R, E): True,
 (R ⊥ O | E): False,
 (S ⊥ O | E, A): True,
 (S ⊥ R | E, A): True,
 (S ⊥ R | E, T, O, A): True,
 (S ⊥ R | E, O, A): True,
 (O ⊥ A | E, T): True,
 (S ⊥ O | R, E, T): True,
 (R ⊥ O | E, S): False,
 …
 (T ⊥ A | E, S): True}
```

I can count the number of tests that pass.

Listing 4.7 Calculate the proportion of d-separations with passing tests

```
num_pass = sum(results.values())
num_dseps = len(dseps.independencies)
num_fail = num_dseps - num_pass
print(num_fail / num_dseps)
```

```
0.2875
```

That implies 29% of the d-separations lack corresponding evidence of conditional independence in the data.

The next step is to inspect these cases of apparent dependence where your DAG says there should be conditional independence. If the evidence of dependence is strong, you need to think about how to improve your causal DAG to explain it.

NOTES ON CONFIGURING CONDITIONAL INDEPENDENCE TESTS

Above, I used the `chi_squared` function, which constructs a specific test statistic with a Chi-squared test distribution – the distribution used to calculate the p-value. The Chi-squared distribution is just another canonical distribution, like the normal or Bernoulli distributions. The Chi-squared distribution comes up frequently for discrete variables, because there are several test statistics in the discrete setting that either have a Chi-squared distribution or get closer to one as the size of the data increases. Overall, independence tests have a variety of test statistics with different test distributions. *pgmpy* provides several options by way of calls to Scipy's stats library.

One common concern is that the test makes strong assumptions. For example, some conditional independence tests between continuous variables assume any dependence between the variables would be *linear*. An alternative approach is to use a *permutation test*, which is an algorithm that constructs the p-value without relying on a canonical test distribution. Permutation tests make less assumptions, but are computationally expensive.

Conditional independence testing is a difficult and nuanced subject. Your goal with conditional independence testing is to refute your causal DAG, not to create the Platonic ideal of a conditional independence testing suite. I recommend getting a testing workflow that is *good enough*, and then focusing on model building. For example, if I had a mix of continuous and discrete variables, rather than implementing a test that can accommodate my different data types, I would just discretize my continuous variables and use a vanilla Chi-squared test, so I keep things moving along.

4.3.3 Some tests are more important than others

The analysis above tested all the d-separations implied by a causal DAG. But some might be more important to you than others. Some certain dependence relations and conditional independence relations are pivotal to a downstream causal inference analysis, while others don't affect that analysis.

For example, consider Figure 3.17 from chapter 3. We added variables Z, X_0, and X_1 to the graph because they might be useful for a specific causal query, namely the causal effect of V on Y. We'll discuss the role these variables play in causal effect inference in Part 4 of

this book. For now, suffice to say that for Z, X_0, and X_1 to play those roles, they must be independent of W_0, W_1, and W_2.

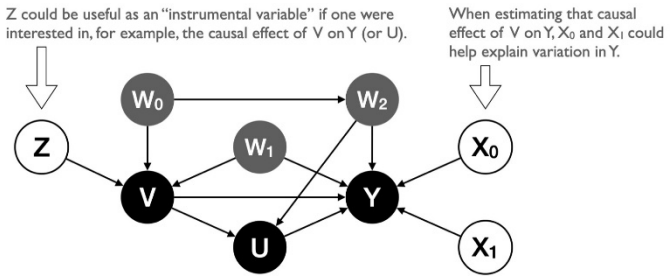


Figure 4.17 from chapter 3, where Z, X0, and X1 were included in the DAG because they play a useful role in analysing the causal effect of U on Y. Their role depends on conditional independence, and it is important to test that they indeed serve those roles.

Similarly, Z must be conditionally independent of Y given V, W_0, W_1, and W_2. So validating those independence relations may be more important than others.

4.4 The trouble with conditional independence tests

Unfortunately, there are several problems with conditional independence tests as a means of testing your causal DAG. In this section, I highlight some of these problems and propose some alternatives.

4.4.1 Statistical tests always have some chance of error

I mentioned that with d-separation, one should not "confuse the map for the terrain"; d-separation is not the same thing as conditional independence. Rather, if your model is a good representation of causality, d-separation *implies* conditional independence. Similarly, conditional independence is not the same as *statistical evidence* of conditional independence**.**

Recall in Chapter 2 how we distinguish between the joint and observational distributions. The causal structure of the true data-generating process imposes conditional independence constraints on the joint probability distribution. You would like to inspect that joint probability distribution for this conditional independence. But you can't do that directly. You can only run statistical tests on the observational distribution.

Just like with prediction, classification, or any other statistical pattern recognition procedure, the model can get it wrong. You can get false negatives; where a pair of variables to be conditionally independent but the statistical independence test concludes they are dependent. You can have false positives, where a statistical independent test finds a pair of variables to be conditional independent when they are not.

4.4.2 The conclusions of conditional independence tests vary with the size of the data

Statisticians developed frequentist independence tests to accommodate small data. They were designed under the explicit assumption that acquiring data was expensive and focused on giving clear results with as few data as possible. In the modern era, many domains are characterized by large data sets. The problem here is that, all else equal, as the size of the data increases, the p-value decreases.

To illustrate, the above test of E ⊥ T | O, R had 30 data points and produced a p-value of 0.56. Below, I run the following bootstrap statistical analysis to show how the estimate of the p-value falls as the size of the data increases.

Listing 4.8 Bootstrap analysis of sensitivity of test of E ⊥ T | O, R to sample size

```
import matplotlib.pyplot as plt
import math
from numpy import mean, quantile

def sample_p_val(sample_size):    #A
   bootstrap_data = fulldata.sample(n=sample_size, replace=True)    #A
   result = chi_square(    #A
       X="E", Y="T", Z=["O", "R"],    #A
       data=bootstrap_data,    #A
       boolean=False,    #A
       significance_level = significance    #A
   )    #A
   p_val = result[1]    #A
   return p_val    #A

def estimate_p_val(sample_size, boot_size=1000):    #B
   samples = []
   for _ in range(boot_size):
       sample = sample_p_val(sample_size)
       samples.append(sample)
   positive_tests = [p_val > significance for p_val in samples]    #C
   prob_conclude_indep = mean(positive_tests)    #C
   p_estimate = mean(samples)    #D
   quantile_05, quantile_95 = quantile(samples, [.05, .95])    #E
   lower_error = p_estimate - quantile_05    #E
   higher_error = quantile_95 - p_estimate    #E
   return p_estimate, lower_error, higher_error, prob_conclude_indep


data_size = range(30, 500, 20)    #F
p_vals, lower_errors, higher_errors, probs_conclude_indep = zip(*[estimate_p_val(size) for
       size in data_size])    #F

plt.title('Amount of data vs. expected p-value (Ind. of E & T given O & R)')    #G
plt.xlabel("Number of examples in data")    #G
plt.ylabel("Expected p-value")    #G
plt.errorbar(data_size, p_vals, yerr=np.array([lower_errors, higher_errors]),
       ecolor="grey", elinewidth=.5)    #G
plt.hlines(significance, 0, 500, linestyles="dashed")    #G
```

#A This function takes the a sample_size argument. It randomly samples "sample_size" number of rows from the full data set. It then runs the chi-squared independence test and returns the p-value.
#B This function conducts a "bootstrap" procedure that runs sample_p_val 1000 times. For a given sample_size, it calculates probability the test will favor conditional independence, as well as the mean and confidence interval for the p-value.
#C Calculate the probability of a test concluding in favor of conditional independence.
#D Calculate the mean of the p-values to get the bootstrap mean.
#E Calculate the 5th and 95th percentiles to get a 90% bootstrap confidence interval
#F Run the extimate_p_value function for a range of sample sizes from 30 to 500.
#G Plot the results.

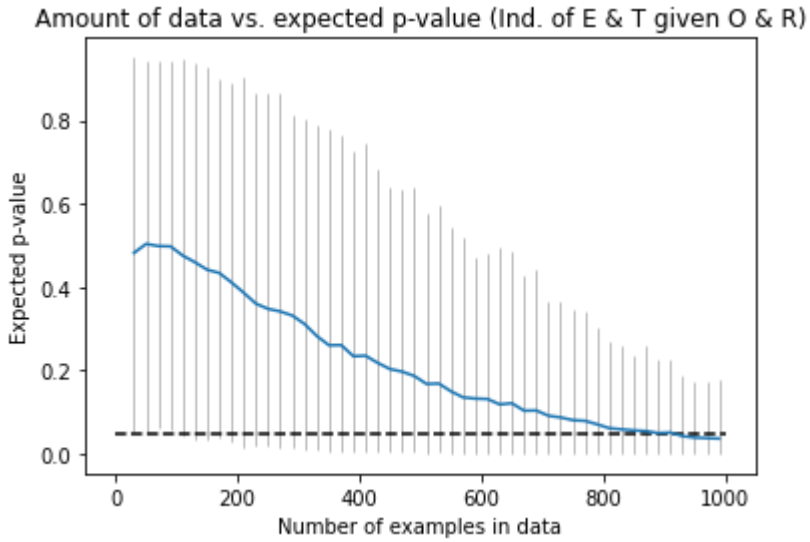


Figure 4. 18 Sample size vs expected p-value of the conditional independence test for E ⊥ T | O, R (blue line). ***The error bars are the 90% bootstrap confidence intervals. The horizontal dashed line is a .05 significance level, above which we favor the null hypothesis of conditional independence and below which we reject it. As the sample size increases, we eventually cross the line. Thus, the result of our refutation analysis depends on the size of the data.***

The blue line in Figure 4. 17 is the expected p-values at different data sizes, the error bars are a 90% confidence interval. By the time we get to a data set of size 1000, the expected p-value is below the threshold, meaning the test favors the conclusion that E ⊥ T | O, R is false.

One might think that as the size of the data increases, the algorithm is detecting subtle dependencies between E and T that were undetectable with small data. Not so, for this transportation data is simulated in such a way that E ⊥ T | O, R is definitely true. So this is a case where more data leads us to rejecting independence because more data leads to more spurious correlation, i.e. patterns that aren't really there.

To drive the point home, the plot in Figure 4. 18 shows how the probability of favoring the true hypothesis (E ⊥ T | O, R) decreases as the size of the data increases.

Listing 4.9 Probability of favoring independence given the amount of data

```
plt.title('Probability of favoring independence given amount of data')
plt.xlabel("Number of examples in data")
plt.ylabel("Probability of test favoring conditional independence")
plt.plot(data_size, probs_conclude_indep)    #A
```

#A Plot the data size on the X axis and the probability of concluding independence on the Y axis.

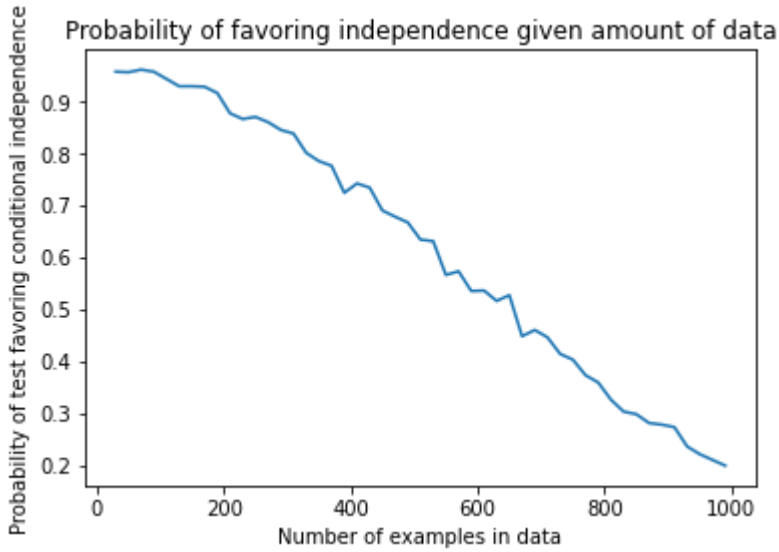


Figure 4. 19 As the size of the data increases, the probability of concluding in favor of this (true) instance of the conditional independence relation E ⊥ T | O, R decreases.

A causal model is either right or wrong about causality in the data generating process it describes. The conditional independence the model implies is either there or it's not. Yet if that conditional independence is there, the test can still conclude in favor of dependence when the data is arbitrarily large.

In practice this is not a huge deal. The *relative* differences between p-values when there is no conditional independence and when there is will be large and obvious. But ideally we could have an analysis that gives the same result for any data size and, alas, that is not the case.

4.4.3 The problem of multiple comparisons

In statistical hypothesis testing, the more tests you run, the more testing errors you rack up. The same is true when running a test for each d-separation implied by a causal DAG. In statistics, this problem is called the *multiple comparisons problem*. The solution is to use statistical approaches that adjust the p-values or calculate something called *false discovery rates*. This may be a good option, especially if you are familiar with computational tools for

multiple testing. However, these methods call attention to a worse problem. Namely, the conditional independence testing approach I've highlighted so far is *wrong*.

4.4.4 Testing causal DAGs with traditional CI tests is fundamentally flawed

I say the proposed conditional independence tests for refutation are "wrong" because the violate the spirit of the statistical hypothesis testing in science. The "spirit" says that the hypothesis we want to test is the *alternative hypothesis* in the statistical test, and the opposite of that hypothesis should be the test's *null hypothesis*. The conditional independence tests in Scilearn and most other statistical libraries assume the alternative hypothesis is dependence and the null hypothesis is conditional independence because in statistical modeling, we're usually trying to find statistical dependence between things. For example, we might want to use "smoking" as a predictor of "lung cancer." But we are focusing on how causality induces conditional independence. Our hypothesis is that if our model is right, certain things should be conditional independent (e.g. "smoking" is independent of "lung cancer" given "tar buildup in the lungs"). Thus, our null hypothesis should be a hypothesis of dependence ("smoking and lung cancer are still dependent given tar buildup in the lungs). That matters because the p-value depends on the null hypothesis. However, you would be hard pressed to find a statistical testing library that makes it easy to do this. The compromise is using the traditional tests where the null hypothesis is of conditional independence, and viewing the results more as a heuristic than as a theoretically rigorous analysis.

4.4.5 CI testing doesn't work well in machine learning settings

Finally, libraries for conditional independence testing are generally limited to one-dimensional variables with fairly simple patterns of correlation between them. However, in machine learning, it is common for variables to have more than one dimension such as vectors, matrices, and tensors. For example, one variable in a causal DAG might represent a matrix of pixels constituting an image. Further, the statistical associations between these variables can be nonlinear and otherwise nuanced[2].

One testing option is to focus on prediction. If two things are independent, then they have no ability to predict one another. Suppose we had two predictive models M_1 and M_2. M_1 predicts Y using Z as a predictor. M_2 predicts Y using X and Z as a predictor. Predictors can have dimension greater than one. If $X \perp Y \mid Z$, then any X has no predictive information about Y beyond what is already provided by Z. So you can test $X \perp Y \mid Z$ by comparing the model predictive accuracy of M_2 to M_1. When the models perform similarly, we have evidence of conditional independence. Note that you'd want to take steps to keep M_2 from "cheating" on its predictive accuracy by overfitting (e.g., regularization, drop-out, etc.)

[2] There has been promise in addressing this issue with conditional independence tests that rely on kernel function classes to represent probability distributions in reproducing kernel Hilbert spaces.

4.5 Refuting a causal DAG given latent variables

As I've mentioned previously, we model the data generating process, not the data. So our efforts to refute our causal DAG should not be limited to the variables observed in the data. However, if a variable in our causal DAG is latent (not observed in the data), we can't run any conditional independence tests involving that variable.

Recall that conditional independence is a *constraint*; if certain cause and effect relationships exist in the data generating process, those relationships constrain the joint probability distribution to have certain conditional independencies between variables. There are other constraints as well, including conditional independencies between *functions* of variables, as well as equalities, inequalities, and bounds. When latent variables prevent certain testing certain d-separations with conditional independence testing, we can sometimes test these other constraints instead. Here, I focus on a specific class of constraints called *Verma constraints*.

4.5.1 Evaluating conditional independence via the Verma Constraints

Verma constraints, like conditional independence constraints, are derived from the Markov property. If a latent variable prevents us from running certain conditional independence tests, there may be testable Verma constraints.

I'll introduce a Verma constraint with an example. The causal DAG in Figure 4. 19 illustrates a simple model of the causal relationship between smoking and lung cancer.

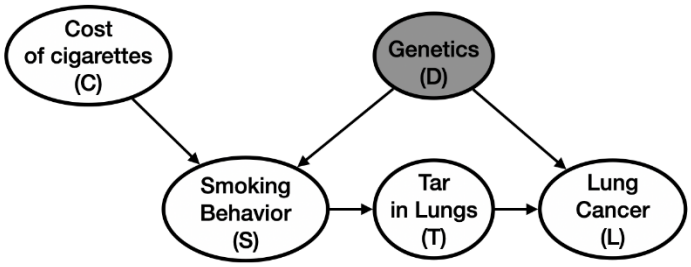


Figure 4. 20 A causal DAG representing smoking's effect on cancer. The variable for genetics is grey because it is unobserved in the data, so we can't run tests for conditional independencies involving D. However, we can test a Verma constraint.

Figure 4. 19 represents how smoking behavior (S) is influenced both by the cost of cigarettes (C) as well as genetic factors (denoted D as in "DNA") that make one more or less prone to nicotine addiction. Those same genetic factors influence one's likelihood of getting lung cancer (L). In this model, smoking's effect on cancer is *mediated* through tar buildup (T) in the lungs.

In practical terms, it is fairly easy to get data on the cost of cigarettes (C) and smoking behavior (S). Let's assume we can quantify tar damage (T) and lung cancer (L) with some non-invasive chest scans relatively easily. The challenge observing the genetics variable (D),

which would involve a genetic screen that (at least historically) would involve more time, money, and resources.

If Genetics (D) were observed, we could test the following d-separations; (C $\perp_G$ T | S), (C $\perp_G$ L | D, T), (C $\perp_G$ L | D, S), (C $\perp_G$ D), (S $\perp_G$ L | D, T), and (T $\perp_G$ D | S). But if it is unobserved, we can only test (C $\perp_G$ T | S).

However, there is a Verma constraint, specified as follows:

$$\sum_{S} P(l|c,s,t)\,P(s|c) \perp_G C$$

This is an independence constraint between the cost of cigarettes (C) and a function of certain conditional probability masses/densities of variables in the causal DAG. That function involves the cost of cigarettes, but when we aggregate over all the different smoking behaviors, the function output doesn't depend at all on the cost of cigarettes. At least, that true according to this causal DAG (again, the difference between implied dependence and actual dependence is why why I still use the subscript in "$\perp_G$").

4.5.2 Verma constraint intuition

We can derive Verma constraints via algorithm. The intuition behind the algorithm is similar to that of the local Markov property. Recall that the definition of the local Markov property is that each node in the graph is independent of its non-descendants given its parents. However, when there are latent variables, we'll have a latent variable version of the property.

> **Each node is independent of its non-descendants given its *effective parents*, non-descendant variables in the node's *confounded component*, and the *effective parents* of the non-descendant variables in its *confounded component*.**

A *confounded component* (usually abbreviated as *c-component*) for a given observed node is the subset that includes itself and other observed nodes that are d-connected through common latent ancestors. Confounder components can overlap, but it is also possible for a node's confounded component to include only itself. In Figure 4. 19, the confounder components for C and T are {C} and {T}, respectively; they contain only the node themselves. But S and L have the same confounded component {S, L}, by way of their common latent ancestor D.

An *effective parent* of a node is the first observed ancestor node on an incoming path through a latent direct parent node. As a mnemonic device, imagine a single father raising a child with the help of the maternal grandparents; given the absence of the mother, the maternal grandparents are *effective parents*. Think of confounded components as cousins and similar relatives related through missing parents.

The algorithm derives Verma constraints from this latent version of the local Markov property, much as we can derive ordinary d-separation from the ordinary local Markov property. The fact that we can derive Verma constraints using a graph-based algorithm demonstrates yet another benefit of working with DAGs as our causal representation.

In the following code I build the causal DAG in Figure 4. 19 using *pgmpy* and the Python library Y^0 (pronounced "why not") to derive the Verma constraint automatically.

Y^0 relies on several external dependencies and is in active development. So I have created a fork of the Y^0 repository for the code examples in this text. I encourage you to explore the original Y^0 library, as it has many useful tools beyond the scope of this chapter, and I expect it will continue to produce interesting new causal functionality.

Listing 4.10 Deriving Verma constraints from a causal DAG and a latent variable

```
!pip install git+https://github.com/altdeep/y0.git    #A
from pgmpy.base.DAG import DAG
from y0.altdeep import verma_from_digraph    #B
dag = DAG()    #C
dag.add_edges_from(    #C
   [    #C
     ('C','S'),    #C
     ('S','T'),    #C
     ('T','L'),    #C
     ('D','S'),    #C
     ('D','L'),    #C
   ]    #C
)    #C
for v in dag.nodes:    #D
   if v == "D":    #D
       dag.nodes[v]['latent'] = True    #D
   else:    #D
       dag.nodes[v]['latent'] = False    #D
verma_constraints = verma_from_digraph(dag)    #E
$rhs.cfactor    #F
[1] "Q[\\{L\\}](T,L)"    #F

$rhs.expr    #F
[1] "\\sum_{u_{1},Tar}P(L|u_{1},Tar)P(Tar)P(u_{1})"    #F

$lhs.cfactor    #G
[1] "\\sum_{S}Q[\\{S,L\\}](C,S,T,L)"    #G

$lhs.expr    #G
[1] "\\sum_{S}P(L|C,S,T)P(S|C)"    #G

$vars    #H
[1] "C"    #H
```

#A Install the forked version of Y^0 from the Altdeep organization. Note that this repository requires Python 3.8 or above.

#B Import the DAG class from pgmpy and the helper function that I added to the fork `verma_from_digraph`.

#C Build the DAG.

**#D Next, label the latent nodes as latent. pgmpy relies on a networkx DiGraph to represent the DAG. We label nodes as latent by creating a node data key called "latent" and setting values for each node to True or False. **

#E The Verma constraints are printed automatically in LaTeX math-type code.

#F The term on the right-hand-side: $Q[\{L\}](T,L) = \sum_{\{u_1,Tar\}} P(L|U_1,Tar)P(Tar)P(u_1)$

#G The term on the left-hand-side: $\sum_S Q[\{S,L\}](C,S,T,L) = \sum_S P(L|C,S,T)P(S|C)$

#H C (cost) is the variable constrained by this Verma constraint. We can use the above math to derive a quantity that should be conditionally independent with C.

The result is a list object containing Verma constraint objects. These objects will print a compiled math-type image when called. The `lhs_expr` method of the Verma constraint object will print the left-hand-side of the independence statement.

```
verma_constraints[0].lhs_expr
```

$$\sum_S P(L|C,S,T)P(S|C)$$

Figure 4. 21 Image generated by the verma_constraints method.

The object indicates that this term should be independent of a set of one or more variables, in our case "C". We can use that output to design an analysis that statistically tests this independence assumption with the data.

4.5.3 Testing a Verma constraint

Let's examine how we'd test this independence assumption in code. Specifically, we'll to calculate $\sum_S P(l|c,s,t)\,P(s|c)$ for each row in the data and assign the value to a new column in the data. Then we'll look at evidence of independence between the data in the "C" column and this new column.

Let's start with some imports.

Listing 4.11 Importing and formatting cigarettes and cancer data

```
from functools import partial
import numpy as np
import pandas as pd


data_url =
      "https://raw.githubusercontent.com/altdeep/causalML/master/datasets/cigs_and_cancer.
      csv"
data = pd.read_csv(data_url)    #A
cost_lower = np.quantile(data["C"], 1/3)    #B
cost_upper = np.quantile(data["C"], 2/3)    #B
def discretize_three(val, lower, upper):    #B
   if val < lower:    #B
       return "Low"    #B
   if val < upper:    #B
       return "Med"    #B
   return "High"    #B

data_disc = data.assign(    #B
   C = lambda df: df['C'].map(    #B
           partial(    #B
               discretize_three,    #B
               lower=cost_lower,    #B
               upper=cost_upper    #B
           )    #B
       )    #B
)    #B
data_disc = data_disc.assign(    #C
   L = lambda df: df['L'].map(str),    #C
```

```
)    #C
print(data_disc)
```

#A Then load the csv into a Panda data frame.
#B Discretize cost (C) into a discrete variable with three levels to facilitate conditional impendence tests.
#C Turn lung cancer (L) from a Boolean to a string, so the conditional independence test will treat it as a discrete variable.

```
       C     S     T      L
0   High   Med   Low   True
1    Med  High  High  False
2    Med  High  High   True
3    Med  High  High   True
4    Med  High  High   True
..   ...   ...   ...    ...
95   Low  High  High   True
96  High  High  High  False
97   Low   Low   Low  False
98  High   Low   Low  False
99   Low  High  High   True

[100 rows x 4 columns]
```

Our first task is to model P(l|c,s,t) and P(s|c). There are many ways to model these conditional probability distributions. One thing I could do is fit a generative model on the causal DAG then apply a graphical modeling inference algorithm. Fitting a causal generative model on data with latent variables is sometimes possible; it is possible in this case. However, a simpler approach would to mode P(l|c, s, t) and P(s| c) directly. I do so in the following code using naive Bayes classifiers. The following code fits a model for P(l| c, s, t).

The naive Bayes classifier is a probabilistic model that *naively* assumes that, in the case of P(l|c,s,t), cost (C), smoking (S), and tar (T) are conditionally independent given lung cancer status (L). According to our causal DAG, that is clearly not true. But that does not matter if all we want is a good way to calculate probability values for L given C, S, and T. A naive Bayes classifier will do that well enough.

The following code will calculate P(l|c,s,t) and P(s|c) and then sum their product over S.

Listing 4.12 Modeling the conditional probabilities in the Verma constraint

```
from pgmpy.estimators import BayesianEstimator
from pgmpy.inference import VariableElimination
from pgmpy.models import NaiveBayes
import statsmodels.api as sm
from statsmodels.formula.api import ols

model_L_given_CST = NaiveBayes()    #A
model_L_given_CST.fit(data_disc, 'L', estimator=BayesianEstimator)    #A
infer_L_given_CST = VariableElimination(model_L_given_CST)    #A

def p_L_given_CST(L_val, C_val, S_val, T_val, engine=infer_L_given_CST): #A

   result_out = engine.query(    #A
       variables=["L"],    #A
       evidence={'C': C_val, 'S': S_val, 'T': T_val},    #A
       show_progress=False    #A
   )    #A
   prob = dict(zip(result_out.state_names["L"], result_out.values))    #A
   return prob[L_val]    #A


model_S_given_C = NaiveBayes()    #B
model_S_given_C.fit(data_disc, 'S', estimator=BayesianEstimator)    #B

infer_S_given_C = VariableElimination(model_S_given_C)    #B

def p_S_given_C(S_val, C_val, engine=infer_S_given_C):    #B
       #B
   result_out = engine.query(    #B
       variables=['S'],    #B
       evidence={'C': C_val},    #B
       show_progress=False    #B
   )    #B
       #B
   prob = dict(zip(result_out.state_names["S"], result_out.values))    #B
   return prob[S_val]    #B
```

```
cstl_outcomes = pd.DataFrame(    #C
   [    #C
       (C, S, T, L)    #C
       for C in ["Low", "Med", "High"]    #C
       for S in ["Low", "Med", "High"]    #C
       for T in ["Low", "High"]    #C
       for L in ["False", "True"]    #C
   ],    #C
   columns = ['C', 'S', 'T', 'L']    #C
)    #C

cstl_outcomes    #C
>    #C
       C     S     T      L    #C
0    Low   Low   Low  False    #C
1    Low   Low   Low   True    #C
2    Low   Low  High  False    #C
3    Low   Low  High   True    #C
```

```
4    Low   Med   Low  False    #C
…    #C
32  High  High   Low  False    #C
33  High  High   Low   True    #C
34  High  High  High  False    #C
35  High  High  High   True    #C
```

```
cs_outcomes = pd.DataFrame(    #D
    [    #D
        (C, S)    #D
        for C in ["Low", "Med", "High"]    #D
        for S in ["Low", "Med", "High"]    #D
    ],    #D
    columns = ['C', 'S']    #D
)    #D

print(cs_outcomes)    #D
>    #D
      C     S    #D
0   Low   Low    #D
1   Low   Med    #D
2   Low  High    #D
3   Med   Low    #D
4   Med   Med    #D
5   Med  High    #D
6  High   Low    #D
7  High   Med    #D
8  High  High    #D

l_cst_dist = cstl_outcomes.assign(    #E
   p_L_CST = cstl_outcomes.apply(    #E
       lambda row: p_L_given_CST(    #E
           row['L'], row['C'], row['S'], row['T']), axis = 1    #E
   )    #E
)    #E

s_c_dist = cs_outcomes.assign(    #F
   p_S_C = cs_outcomes.apply(    #F
       lambda row: p_S_given_C(row['S'], row['C']), axis = 1    #F
   )    #F
)    #F

dist = l_cst_dist.merge(s_c_dist, on=['S', 'C', 'T', 'L'], how='left')   #G

print(dist)    #G
       C     S     T      L   p_L_CST     p_S_C    #G
0    Low   Low   Low  False  0.382637  0.131410    #G
1    Low   Low   Low   True  0.617363  0.131410    #G
2    Low   Low  High  False  0.243640  0.131410    #G
3    Low   Low  High   True  0.756360  0.131410    #G
4    Low   Med   Low  False  0.602541  0.246795    #G
…    #G
32  High  High   Low  False  0.399487  0.576324    #G
33  High  High   Low   True  0.600513  0.576324    #G
34  High  High  High  False  0.256916  0.576324    #G
35  High  High  High   True  0.743084  0.576324    #G
    #G
```

```
[144 rows x 8 columns]    #G

dist_mod = dist.assign(    #H
   product = dist.p_L_CST * dist.p_S_C    #H
)    #H
    #H
print(dist_mod)    #H
>    #K
       C     S     T      L   p_L_CST     p_S_C   product    #H
0    Low   Low   Low  False  0.382637  0.131410  0.050282    #H
1    Low   Low   Low   True  0.617363  0.131410  0.081128    #H
2    Low   Low  High  False  0.243640  0.131410  0.032017    #H
3    Low   Low  High   True  0.756360  0.131410  0.099393    #H
4    Low   Med   Low  False  0.602541  0.246795  0.148704    #H
…    #H
32  High  High   Low  False  0.399487  0.576324  0.230234    #H
33  High  High   Low   True  0.600513  0.576324  0.346090    #H
34  High  High  High  False  0.256916  0.576324  0.148067    #H
35  High  High  High   True  0.743084  0.576324  0.428257    #H

dist_verma = dist_mod.groupby(    #I
   ['C', 'T', 'L']    #I
   ).agg('sum').reset_index().drop(    #I
       columns=["p_L_CST", "p_S_C"]    #I
   ).rename(    #I
       columns = {'product':'sum_product'}    #I
   )    #I

print(dist_verma)    #I
>    #I
       C     T      L  sum_product    #I
0   High  High  False     0.348536    #I
1   High  High   True     0.651464    #I
2   High   Low  False     0.496355    #I
3   High   Low   True     0.503645    #I
4    Low  High  False     0.264353    #I
5    Low  High   True     0.735647    #I
6    Low   Low  False     0.399869    #I
7    Low   Low   True     0.600131    #I
8    Med  High  False     0.373653    #I
9    Med  High   True     0.626347    #I
10   Med   Low  False     0.523694    #I
11   Med   Low   True     0.476306    #I
```

```
df_verma = data_disc.merge(dist_verma, on=['C', 'T', 'L'], how='left')   #J
print(df_verma)    #J
>    #J
       C     S     T      L  sum_product    #J
0   High   Med   Low   True     0.503645    #J
1    Med  High  High  False     0.373653    #J
2    Med  High  High   True     0.626347    #J
3    Med  High  High   True     0.626347    #J
4    Med  High  High   True     0.626347    #J
..   ...   ...   ...    ...          ...    #J
95   Low  High  High   True     0.735647    #J
96  High  High  High  False     0.348536    #J
97   Low   Low   Low  False     0.399869    #J
```

```
98  High   Low   Low  False     0.496355    #J
99   Low  High  High   True     0.735647    #J
    #J
[100 rows x 5 columns]    #J

df_verma.boxplot("sum_product", "C")    #K
```

#A We'll use a naïve Bayes classifier in pgmpy to calculate the probability value for a given value of L given values of C, S, and T. In this case I use variable elimination.
#B We do the same with P(s|c).
#C Now I'll calculate these values for each possible combination of outcomes of L, C, S, and T. First, I use list comprehensions to make a data frame containing all the combinations.
#D Next I create a data frame containing all combinations of s and c.
#E Then I use the `assign` and `apply` methods on the Pandas data frame to apply our conditional probability functions to each combination, creating a new column for P(l|c,s,t) on the data frame containing combinations of l, c, s, and t.
#F And I do the same on the data frame containing combinations of s and c, creating a column for P(s|c).
#G Then we join the two data frames so we have combinations P(s|c) and P(l|c, s, t) for all combinations of {s, c} and {l, c, s, t} in one data frame.
#H Now I create a new column for the product of P(s|c) and P(l|c, s, t).
#I I then implement the summation over S in $\sum_S \boldsymbol{P(l|c,s,t)P(s|c)}$ using the DataFrame class's agg method for aggregating across grouped rows.
#j Now, I merge these combinations into the data, so we can add $\sum_s \boldsymbol{P(l|c,s,t)P(s|c)}$ as a column to that data.
#K The Verma constraint says C and $\sum_s \boldsymbol{P(l|c,s,t)P(s|c)}$ should be independent, so we use a box plot plot that plots values of $\sum_s \boldsymbol{P(l|c,s,t)P(s|c)}$ against values of C to visualize inspect whether C and $\sum_s \boldsymbol{P(l|c,s,t)P(s|c)}$ look independent.

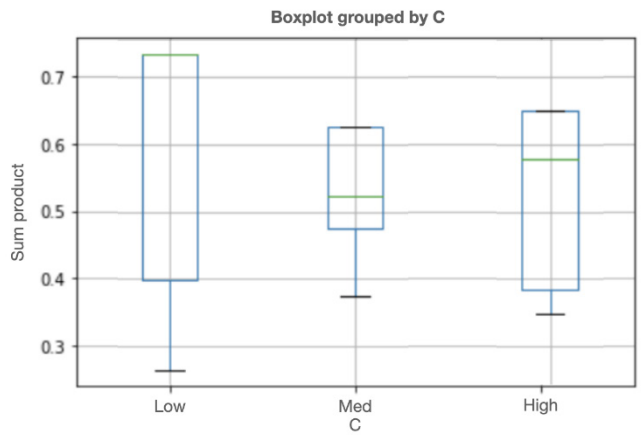


Figure 4. 22 A box plot visualization of cost (C) on the x-axis and the function $\sum_S P(l|c,s,t)P(s|c)$ on the y-axis (labeled "Sum product"). Overlap of the distributions of *sum product* for each value of C support the Verma constraints assertion that these two quantities are independent.

The x-axis in Figure 4. 21 is different levels of cost (low, medium, and high). The y-axis represents the distribution of our sum product across each level of C. The width of the boxes, the black and green horizontal lines represent quantiles

In summary, it looks as though the distributions of the sum-product don't change much across the different levels of cost; that is what independence is supposed to look like.

I can also derive p-value again. This time I use an analysis of variance (ANOVA) approach to derive a p-value (using an F-test rather than a chi-square test). The p-values are in the "PR(>F)"[3].

Listing 4.13 Using ANOVA to evaluate independence

```
model = ols('sum_product ~ C', data=df_verma).fit()    #A
aov_table = sm.stats.anova_lm(model, typ=2)    #A
aov_table["PR(>F)"]["C"]    #A
> 0.19364363417824293    #B


model = ols('sum_product ~ T', data=df_verma).fit()    #C
aov_table = sm.stats.anova_lm(model, typ=2)    #C
aov_table["PR(>F)"]["T"]    #C
> 0.07522736142622223    #C

model = ols('sum_product ~ L', data=df_verma).fit()    #C
aov_table = sm.stats.anova_lm(model, typ=2)    #C
aov_table["PR(>F)"]["L"]    #C
> 7.0547852477839215e-28    #C
```

#A A recipe for doing ANOVA using the statmodels library
#B A p-value that high is definitely not evidence against independence. It looks like the data supports the constraint.
#C Just as a sanity check, we can run the same test for T and L. As we expected, these are much smaller. In the latter case, the number (in scientific notation) is effectively zero.

These are lower, both falling below the common .1 threshold where a standard hypothesis test would reject the hypothesis that the sum-product is independent of T and L.

4.5.4 Final takeaways

That was a lot of math and code, so I want to highlight the high-level takeaways. The general idea certain constraints, like conditional independence constraints, can't be tested if the variables we need for the test aren't observed in the data. But there may be some testable implications of our causal DAG that are still testable. Verma constraints are one example of this. There are others, such as upper and lower bounds on certain observable quantities. The math here can be a bit more subtle, but of course, software libraries can make that analysis more accessible.

4.6 Summary

- Causal modeling induces conditional independence constraints on the joint probability distribution. D-separation provides a graphical representation of conditional independence constraints.

3 "PR(>F)" means the probability of seeing an F-statistic for a given variable (in our case C) at least as large as the F-statistic calculated from the data assuming that variable is independent of sum_product, i.e., the p-value.

- Building an intuition for d-separation is important for reasoning about causal effect inference and other queries.
- The colliders might make d-separation confusing. But you can build intuition by using d-separation functions in `networkx` and `pgmpy`.
- Using traditional conditional independence testing libraries to test d-separation has its challenges. The tests are sensitive to sample size, the hypotheses are misaligned, and they don't work well in many machine learning settings.
- When there are latent variables, the causal DAG still has testable implications called Verma constraints.

5

Connecting causality and deep learning

This chapter covers

- **Using deep learning to enhance a causal graphical model**
- **Training a causal graphical model with a variational autoencoder**
- **Using causal methods to enhance machine learning**

So far in this book, I have not said much about deep learning. Deep learning refers to a machine learning approach that uses artificial neural networks, which are essentially nonlinear regression models stacked together in layers. "Deep" refers to using neural nets with many layers. More layers mean more modeling power, particularly in terms of modeling of high dimensional and nonlinear data such visual media and natural language text. Neural nets have been around for awhile, but deep neural nets were hard to train. But recent advancements in hardware and computational tools for gradient-based optimization made training deep nets much easier. That ease is why in recent years there have been multiple cases of deep learning's "superhuman" performance on machine learning benchmarks, and why it will continue making headlines for some time.

No doubt, you've heard of these accomplishments of deep learning. You've seen large language models hold nuanced and coherent conversations. You've seen examples of deep reinforcement learning defeating experts at international gaming tournaments. You've heard of deep generative models producing works of art so impressive that they sell for millions at auction and wins fine arts competitions.

So when you watch a presentation on causal inference from an economist and the modeling approach sounds like linear regression with bells and whistles, it feels like there's a major disconnect. What is going on?

Clearly causal inference researchers are talking about something important, after all they are winning Nobel Prizes and Turing Awards. But they either say little about deep learning or are downright critical. Meanwhile, deep learning enthusiasts and critics argue on social media about whether deep learning can learn causality. Are causal inference and deep learning competing paradigms? What is the connection?

Indeed, causality and deep learning are compatible. In this chapter, I demonstrate a few ways that deep learning and causality interact. For code notebooks, links to the data, and citations for the methods and case studies we discuss, visit www.altdeep.ai/p/causalmlbook. The examples in this chapter will build intuition about the connection between deep learning and causal modeling. In other parts of the book, we'll look at other ways deep learning and machine learning in general connect to causality. For example, we'll explore double machine learning for causal effect inference in chapter 11.

5.1 Using deep learning to enhance causal inference

In this section, I'll focus on one example of how deep learning can enhance a causal model. Specifically, we'll use a deep probabilistic machine learning approach called a variational autoencoder framework to train a causal graphical model.

5.1.1 Modeling images with a deep generative causal model

Recall the MNIST data, comprised of images of digits and their digit "labels" illustrated in Figure 5.1.

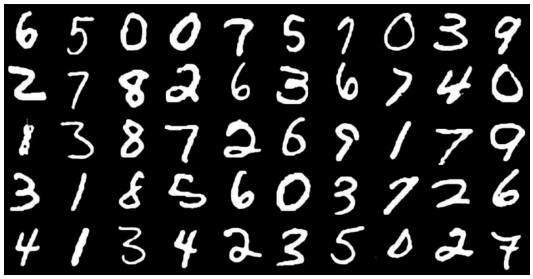


Figure 5.1 MNIST data featuring images of handwritten digits and their digit labels.

There is a related dataset called Typeface MNIST (TMNIST) that also features digit images and their digit labels. However, instead of handwritten digits, the images are digits rendered in 2,990 different fonts illustrated in Figure 5.2. For each image, in addition to a digit label, there is a font label. Examples of the font labels include "GrandHotel-Regular," "KulimPark-Regular", and "Gorditas-Bold."

Figure 5.2 Examples from the Typeface MNIST comprised of typed digits with different typefaces. In addition to a digit label for each digit, there is a label for one of 2,990 different typefaces (fonts).

In this analysis, we'll combine these datasets into one, and build a simple deep causal generative model on that data. We'll simplify the "fonts" label into a sample binary label that indicates "handwritten" for MNIST images, and "typed" for the TMNIST images.

We have seen how to build a causal generative model on top of the DAG. We factorized the joint distribution into a product of *causal Markov kernels* representing the conditional probability distributions for each node conditional on their parents in the DAG. In our previous examples in *pgmpy*, we fit a conditional probability table for each of these kernels.

You can imagine how hard it would be to use a conditional probability table to represent the conditional probability distribution of pixels in an image. But there is nothing stopping us from modeling the causal Markov kernel with a deep neural net, which we know is flexible enough to with high dimensional features like pixels. In this section, I demonstrate how to use deep neural nets to model the causal Markov kernels defined by a causal DAG.

5.1.2 Leveraging the universal function approximator

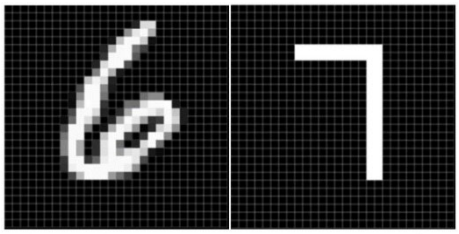
Deep learning is exceedingly useful at mapping inputs to outputs. We can frame the problem in terms of universal function approximation. We imagine there exists some function that maps some set of inputs to some set of outputs, but we either don't know the function or its too hard to write down in math or code. Given enough examples of those inputs and those outputs, deep learning can approximate that function with high precision. Even when that function is nonlinear and high dimensional, with enough data deep learning will learn a good approximation.

We certainly work with functions in causal modeling and inference. And sometimes it makes sense to approximate them, so long as the approximations preserve causal invariances. For example, the causal Markov property makes us interested in functions that map values of a node's parents in the causal DAG to values (or probability values) of that node. In this section, we'll do this mapping between a node and its parents with the variational autoencoder (VAE) framework. We'll train two deep neural nets in the VAE, and one of them maps parent cause variables to an outcome variable. This example showcases the use of deep learning when causality is nonlinear and high dimensional; the effect variable will be an image

represented as a high-dimensional array, and the cause variables will represent the contents of the image.

5.1.3 Causal abstraction and plate models

But what does it mean to build a causal model of an image? Images are comprised of pixels arranged in a grid. As data, we can represent that pixel grid as a matrix of numerical values corresponding to color. In the case of both MNIST and TMNIST, the image is a 28 x 28 matrix of grayscale values, as illustrated in Figure 5.3.



An image of handwritten "6" (as in MNIST)

An image of typed "7" (as in Typeface MNIST)

Figure 5.3 An MNIST image of "6" (left) and a TMNIST image of a "7." In their raw form, these are 28 x 28 matrices of numeric values corresponding to greyscale values.

A typical machine learning model looks at this 28 by 28 matrix of pixels as 28*28=784 features. The machine learning algorithm learns statistical patterns connecting the pixels to one another and their labels. Based on this fact, one might be tempted to treat each individual pixel as a node in the naive causal DAG as in Figure 5.4, where for visual simplicity I draw 16 pixels (an arbitrary number) instead of all 784.

In Figure 5.4 there are edges from the digit and is handwritten variables to each pixel. Further, there are examples of edges representing possible causal relationships *between* pixels. Causal edges between pixels imply the color of one pixel is a cause of another. Perhaps most of these relationships are between nodes that are close, with a few far-reaching edges. But how would we know if one pixel causes another? If two pixels are connected, how would we know the direction of causality?

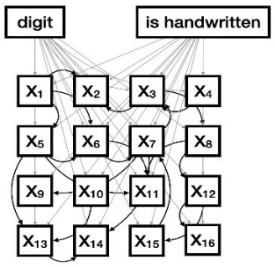


Figure 5.4 What a naive causal DAG might look like for an image represented by a 4x4 matrix.

WORKING AT THE RIGHT LEVEL OF ABSTRACTION

But with these interpixel connections, the naïve DAG we see in Figure 5.4 is already quite unwieldy with only 16 pixels. It would be much worse at 784 pixels. Aside from the unwieldiness of a DAG, the problem with a pixel-level model is that our causal questions are generally not at the pixel level. In other words, the pixel is too low a level of abstraction. That low level of abstraction is why thinking about causal relationships between individual pixels feels a bit absurd.

In applied statistics domains such as econometrics, social science, public health, and business, our data has variables like "per capital income", "revenue", "location", "age", etc. These variables are typically already at the level of abstraction we want to think about. But modern machine learning focuses on many perception problems from raw media, such as images, video, text, and sensor data. We don't generally want to do causal reasoning at the level of these low-level features. Our causal questions are about the high-level abstractions behind these low-level features. So we need to model at these higher abstraction levels.

So instead of thinking about individual pixels, we'll think about the entire image. We'll define a variable X to represent how the image appears, i.e., the outcome for a matrix random variable. Figure 5.5 illustrates a causal DAG for the TMNIST caste. Simply, the identity of the digits (0 – 9) and the font (2,990 possible values) are the causes, and the image is the effect.

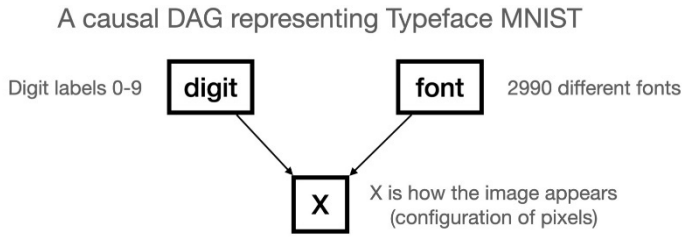


Figure 5.5 Simple causal DAG that represents the implied data generating process behind Typeface MNIST.

In this case we are using the causal DAG to make an assertion that the label causes the image. That is not the case in general, as we'll discuss in our case study on semi-supervised learning in section 5.3. As with all causal models, it depends on the data generating process within a domain. Let's explore why we might make this assertion in the case of MNIST.

WHY SAY THAT THE DIGIT CAUSES THE IMAGE?

In Plato's *Allegory of the Cave*, Socrates describes a group of people who have lived in a cave all their lives, without seeing the world. They face a blank cave wall and watch shadows projected on the wall from objects passing in front of a fire behind them. The shadows are simplified and sometimes distorted representations of the true objects passing in front of the fire. In this case, we can think of the form of the objects being a cause of the shadow. Analogously, the true form of the digit label (the actual encoded character) causes the representation in the image.

The MNIST images were written by people. Suppose we are looking at MNIST image for the number 8. While we are talking about Greek philosophers, we can imagine the writer had a *Platonic* ideal of the number 8 in his head as he was writing. The resulting written image is a version of that "8" distorted by motor variation in the hand, the angle of the paper, the friction of the pen on the paper, and other factors, i.e., a "shadow" of that "8" in his mind.

Further, the MNIST data are photographs of written digits. Photographic images are created by light falling on surfaces, like cave walls only more photosensitive. For you, that surface is the retina in your eyes. For a robot, that surface might be an image sensor in a mounted camera. For the robot, the objects in the environment are the causes of the forms that appear on the image sensor. And as you read this page, the existence and form of the words on this page are the causes of the forms on your retina, which your brain turns into what you *see*. If your mind could cause images to appear as patterns of activated rods and cones on your retina, and then those images caused new objects to physically manifest in the real world, that would be impressive.

In computer vision, this concept is sometimes called *vision as inverse graphics*. Generalizing beyond computer vision, in probabilistic machine learning the concept is sometimes called *analysis as synthesis*. The basic idea is that when you have some device, like a camera, that collects raw signals from the environment and the task is to infer the actual objects or events that resulted in those signals, then causality flows from those objects/events to the signals. The inference task is to use the observed signals (shadows on the cave wall) to infer the nature of the causes (objects in front of the fire).

That said, images can be causes too. For example, if you were modeling how people behave after seeing an image in a mobile app (e.g., "click", "like", "swipe left"), then you could model the image as cause of the behavior.

PLATE MODELING

Modeling 2,990 fonts in our TMNIST data is overkill for our purposes here. Instead, I combined these datasets into one, half from MNIST and half from typeface MNIST. Along with the digit label, I'm just going to have a simple binary label called "is handwritten", which is 1 (true) for images of handwritten digits from MNIST and 0 (false) for images of "typed" digits from TMNIST. So we modify our causal DAG to get Figure 5.6.

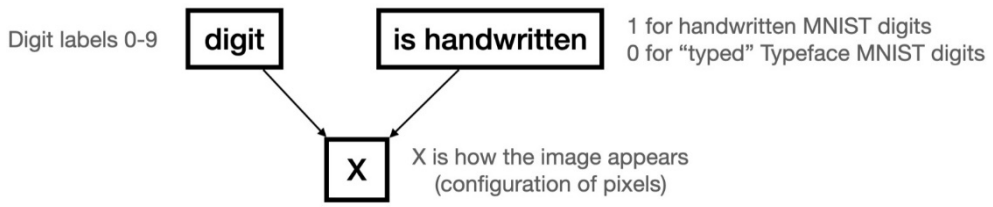


Figure 5.6 A causal DAG representing the combined MNIST and TMNIST data, where "is handwritten" is 1 (MNIST images) or 0 (TMNIST images).

Plate modeling is a visualizing technique used in probabilistic machine learning that provides an excellent way to visualize the higher-level abstractions while preserving the lower-level dimensional detail. Plate notation is a method of visually representing variables that repeat in a DAG – in our case we have repetition of the pixels. Instead of drawing each of the 784 pixels as an individual node, we use a rectangle or "plate" to group repeating variables into subgraphs. We then write a number on the plate to represent the number of repetitions of the entities on the plate. Plates can nest within one another to indicate repeated entities nested within repeated entities. Each plate gets a letter subscript indexing the elements on that plate. Figure 5.7 illustrates an example of our plate model.

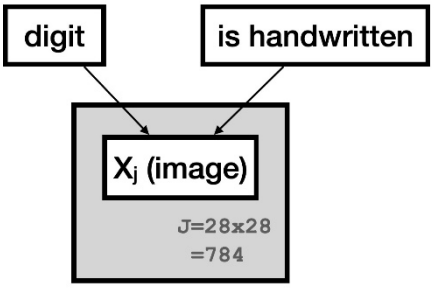


Figure 5.7 A plate model representation of the causal DAG. Plates represent repeating variables, in this case 28x28=784 pixels. X_j is the j^{th} pixel.

The causal DAG in Figure 5.7 represents one image. During training, we have a large set of training images. Next, we'll modify the DAG to capture all the images in the training data.

5.2 Training the neural causal model

Our training data has N example images, so we need our plate model to represent all N images in the training data, half "handwritten" and half "typed." So we add another plate corresponding to repeating N sets of images and labels as in Figure 5.8.

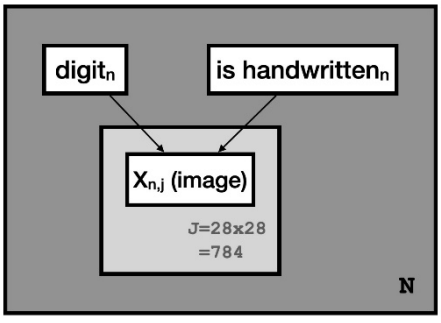


Figure 5.8 The causal model with an additional plate for the N images in the data.

Now we have a causal DAG that illustrates both our desired level of causal abstraction as well as the dimensional information we need to setup training the neural nets in the model.

Let's first load Pyro, other libraries and this combined data in Python.

Listing 5.1 Initial setup for the deep causal model

```
import random

import torch
import pandas as pd
import numpy as np
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import os
from torchvision.transforms import ToTensor, Lambda

import torch.nn as nn
import torchvision.transforms as transforms

import pyro
import pyro.distributions as dist
import pyro.contrib.examples.util
from pyro.infer import SVI, Trace_ELBO
from pyro.optim import Adam

import matplotlib.pyplot as plt

assert pyro.__version__.startswith('1.8.1')   #A
pyro.distributions.enable_validation(False)


class CombinedDataset(Dataset):    #B
    def __init__(self, csv_file):
        self.data_frame = pd.read_csv(csv_file)

    def __len__(self):
        return len(self.data_frame)

    def __getitem__(self, idx):
        images = self.data_frame.iloc[idx, 3:]    #C
        images = np.array([images])/255.    #C
        images = images.reshape(28, 28)    #C
        images = images.astype('float32')    #C
        transform = transforms.ToTensor()    #C
        images = transform(images)    #C

        digits = self.data_frame.iloc[idx, 2]    #D
        digits = np.array([digits])    #D
        digits = digits.astype('int')    #D
        digits = int(digits.squeeze())    #D

        is_handwritten = self.data_frame.iloc[idx , 1]    #E
        is_handwritten = np.array(is_handwritten)    #E
        is_handwritten = is_handwritten.astype('int')    #E

        return images, digits, is_handwritten    #F
```

```
def setup_data_loaders(batch_size=64, use_cuda=False):    #G

    combined_dataset = CombinedDataset(

       "https://raw.githubusercontent.com/altdeep/causalML/master/datasets/combined_mnist_t
       mnist_data.csv"
    )
    n = len(combined_dataset)

    train_length = int(n*0.7)
    test_length = n - train_length
    train_set, test_set = torch.utils.data.random_split(
        dataset=combined_dataset,
        lengths=[train_length, test_length],
        generator=torch.Generator().manual_seed(43)
    )

    # Create data loaders for train and test
    kwargs = {'num_workers': 1, 'pin_memory': use_cuda}
    train_loader = torch.utils.data.DataLoader(
        dataset=train_set,
        batch_size=batch_size,
        shuffle=True,
        **kwargs
    )
    test_loader = torch.utils.data.DataLoader(
        dataset=test_set,
        batch_size=batch_size,
        shuffle=False,
        **kwargs
    )
    return train_loader, test_loader
```

#A This code was written for Pyro 1.8.1. See www.altdeep.ai/p/causalmlbook for any updates to new versions.
#B This class loads and processes a dataset that combines the MNIST and Typeface MNIST. The output is a torch.utils.data.Dataset object.
#C Load, normalize, and reshape the images to a 28x28 pixel.
#D Get and process the digits labels, 0-9.
#E 1 for handwritted digits (MNIST) 0 for "typed' digits (TMNIST)
#F Return tuple of the image, the digit label, and the is_handwritten label
#G Setup data loader that loads the data and splits it into training and test sets

5.2.1 Fitting the model with a variational autoencoder

The variational autoencoder (VAE) is perhaps the simplest deep probabilistic machine learning modeling approach. In the typical setup for applying VAE to images, we introduce a latent continuous variable Z that has lower dimension than the image date. For each image in the data, there is a corresponding latent Z-value. That value represents an "encoding" that contains compressed information in the image. A Z encoding vector takes information in X and compresses it into a lower dimension. This setup is illustrated in Figure 5.9.

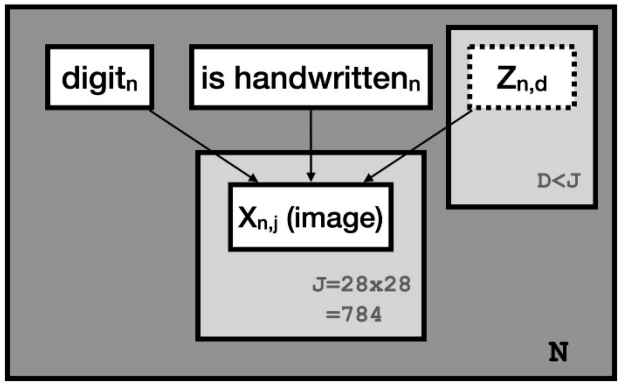


Figure 5.9 The causal DAG plate model extended to include an "encoding" variable Z. The variable is latent, indicated by the dashed line.

Z appears as a new parent in the causal DAG, but it is important to note that the classical VAE framework does not define Z as causal. But now that we are thinking causally, we'll give Z a causal interpretation. Specifically, we'll think of it as a continuous latent *stand-in* for all the causes of what you see in the images that are not causally explained by "digit" and "is handwritten." For example, all the nuance of the various fonts in the TMNIST labels that we are deliberately ignoring will get captured by Z after training. That said, it is important to remember that the representation we learn for Z is not the same as learning actual latent causes.

The VAE setup will train two deep neural networks, one that encodes an image into a value for Z and one that decodes a value of Z into an image. The benefit of using deep learning is that the neural nets can capture the complex and nonlinear relations needed to model the image as an effect caused by "digit" and "is handwritten." Modeling images would be difficult with the conditional probability tables and other simple parametrizations of causal Markov kernels we've discussed previously.

The first neural network is called a "decoder." The decoder generates an image from the digit label, its handwritten label, and a Z value, as in Figure 5.10.

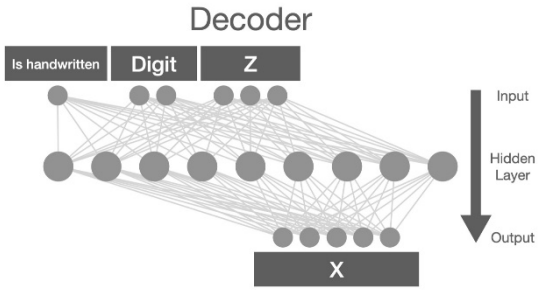


Figure 5.10 The decoder network generates an image X from Z and the labels "is handwritten" and "digit".

The decoder acts like a rendering engine; given a Z encoding value and the values for *digit* and *is handwritten*, it renders an image. We implement the decoder as follows.

Listing 5.2 Implement the decoder

```
class Decoder(nn.Module):    #A
    def __init__(self, z_dim, hidden_dim):
        super().__init__()
        self.softplus = nn.Softplus()    #B
        self.sigmoid = nn.Sigmoid()    #B
        self.fc1 = nn.Linear(z_dim + 10 + 2, hidden_dim)    #C
        self.fc21 = nn.Linear(hidden_dim, 784)    #D

    def forward(self, z, digit, is_handwritten):    #E
        input = torch.cat([z, digit, is_handwritten], 1)    #F
        hidden = self.softplus(self.fc1(input))    #F
        X_img_parameter = self.sigmoid(self.fc21(hidden))    #H
        return X_img_parameter
```

#A The decoder method of a VAE class. Visit www.altdeep.ai/p/causalmlbook for links to a notebook with the full code.

#B The softplus and sigmoid are nonlinear transforms (activation functions) used in mapping between layers.

#C fc1 will combine with the softplus to map from the Z vector, the 10 dimensional one-hot encoded digit label vector, and the two dimensional one-hot-encoded is_handwritten variable to the hidden layer.

#D The fc21 linear transform will combine with the sigmoid to map the hidden layer to the 28x28=784 dimensional output image vector space.

#E Define the forward computation from the latent Z variable value to a generated X variable value.

#F First combine Z and the labels.

#G Then compute the hidden layer.

#H Finally, pass the hidden layer to a linear transform, then to a sigmoid transform to output a parameter vector of length 784. Each element of the vector corresponds to a Bernoulli parameter value for an image pixel. The model will use this vector to parameterize a multivariate Bernoulli probability distribution for the image variable and calculate a likelihood for each image.

We use the decoder in the causal model. Our causal DAG acts as the scaffold for a causal probabilistic machine learning model that, with the help of the decoder, defines a joint probability distribution on {is-handwritten, digit, X, Z}, where Z is latent. We can use the model to calculate the likelihood of the training data for a given value of Z. The following model code is a class method for a Pytorch neural network module. We'll see the entire class later.

Listing 5.3 The causal model

```
    def model(self, x, digit, is_handwritten):    #A
        pyro.module("decoder", self.decoder)    #A
        batch_size = x.size(0)
        options = dict(dtype=x.dtype, device=x.device)
        with pyro.plate("data", x.shape[0]):    #B

            z_loc = x.new_zeros(torch.Size((x.shape[0], self.z_dim)))    #C
            z_scale = x.new_ones(torch.Size((x.shape[0], self.z_dim)))   #C
            z = pyro.sample(    #C
                    "Z",    #C
                    dist.Normal(z_loc, z_scale).to_event(1)    #C
            )    #C

            digit_prob = torch.ones(batch_size, 10, **options) / 10.0   #D
            digit = pyro.sample(    #D
                    "digit",    #D
                    dist.OneHotCategorical(digit_prob),    #D
                    obs=digit    #E
            )

            hw_prob = torch.ones(batch_size, 2, **options) / 2.0    #F
            is_handwritten = pyro.sample(    #F
                    "is_handwritten",    #F
                    dist.OneHotCategorical(hw_prob),    #F
                    obs=is_handwritten    #G
            )

            X_img_param = self.decoder.forward(    #H
                    z,    #H
                    digit,    #H
                    is_handwritten    #H
            )    #H
            X_img = pyro.sample(    #I
                  "obs",    #I
                  dist.Bernoulli(X_img_param).to_event(1),    #I
                  obs=x.reshape(-1, 784)    #J
            )
        return X_img
```

#A model is a method of the VAE (see Listing 5.7 for the full class). Within the method we register the decoder, a PyTorch module, with Pyro. This lets Pyro know about the parameters inside of the decoder network.

#B This context manager represents the N-size plate representing repeating examples in the data in Figure 5.9. In this case, N is the batch size. It works like a for loop iterating over each data unit in the batch.

#C We model the joint probability of Z, digit, and is_handwritten sampling each from canonical distributions. We sample Z from a multivariate normal with location parameter z_loc (all zeros) and scale parameter z_local (all ones).

#D We also sample the digit from a one-hot categorical distribution. Equal probability is assigned to each digit.

#E The obs argument allows us to condition the model on the actual digits.

#F We similarly sample the is_handwritten variable with the binary outcome. Note that since is_handwritten is binary, a Bernoulli distribution would work here.

#G Again, we condition this distribution on training values of is_handwritten.

#H The decoder maps digit, is_handwritten, and Z to a probability parameter vector.

#I That parameter vector is passed to the Bernoulli distribution, which models the pixel values in the data.

#J And again, this distribution is conditioned on actual images.

Our probabilistic ML model models the joint distribution of {Z, X, digit, is-handwritten}. But since Z is latent, the model will need to learn P(Z|X, digit, is-handwritten). But given we use the decoder neural net to go from Z and the labels to X, the distribution of Z given X and the labels will be complex. So we will use *variational inference*, a technique where we first define an approximating distribution Q(Z|X, digit, is-handwritten) and try to make that distribution as close to P(Z|X, digit, is-handwritten) as we can.

The main ingredient of the approximating distribution is the second neural net in the VAE framework, the "encoder," illustrated in Figure 5.11. The encoder maps an observed image and its labels in the training data to a latent Z variable.

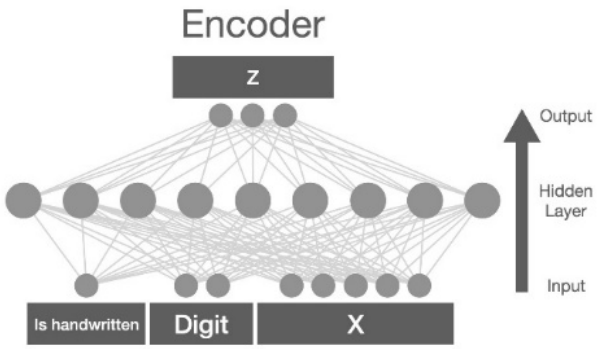


Figure 5.11 The encoder maps actual images to latent Z variable.

The encoder does the work of compressing the information in the image into a lower dimensional encoding. The following list implements the encoder, which will infer the latent Z variable given an image, is_handwritten, and the digit label.

Listing 5.4 Implement the encoder

```
class Encoder(nn.Module):    #A
    def __init__(self, z_dim, hidden_dim):
        super().__init__()
        self.softplus = nn.Softplus()    #B
        self.fc1 = nn.Linear(784 + 10 + 2, hidden_dim)    #C
        self.fc21 = nn.Linear(hidden_dim, z_dim)    #D
        self.fc22 = nn.Linear(hidden_dim, z_dim)    #D


    def forward(self, x, digit, is_handwritten):    #E
        x = x.reshape(-1, 784)    #F
        input = torch.cat([x, digit, is_handwritten], 1)    #G
        hidden = self.softplus(self.fc1(input))    #H
        z_loc = self.fc21(hidden)    #I
        z_scale = torch.exp(self.fc22(hidden))    #I
        return z_loc, z_scale
```

#A The encoder is an instance of a Pytorch module.
#B In the encoder, we'll only use the softplus transform (activation function).

#C The linear transform fc1 combines with the softplus to map the 784 dimensional pixel vector, 10 dimensional digit label vector, and 2 dimensional is_handwritten vector to the hidden layer.
#D The linear transforms fc21 and fc22 will combine with the softplus to map the hidden vector to Z's vector space.
#E Define the reverse computation from an observed X variable value to a latent Z variable value.
#F Flatten the 28x28 pixel matrix into a flat 784 dimensional vector.
#G Combine the image vector, digit label, and is-handwritten label into one input.
#H Map the input to the hidden layer.
#I The VAE framework will sample Z from a Normal distribution that approximates P(x|z, digit, is-handwritten). The final transforms map the hidden layer to a location and scale parameter for that Normal distribution.

The encoder is used in a random function called a "guide function" that represents the *variational distribution* Q(Z|X, is_handwritten, digit) that approximates P(Z|X, is_handwritten, digit). The guide is a function that generates samples of Z given actual images and labels from the data. Specifically, Z is sampled from a Normal distribution. The image and the labels are passed to the encoder, which returns the parameters of that Normal distribution.

Listing 5.5 The guide function

```
    def guide(self, x, digit, is_handwritten):    #A
        pyro.module("encoder", self.encoder)    #B
        with pyro.plate("data", x.shape[0]):    #C
            z_loc, z_scale = self.encoder(x, digit, is_handwritten)    #D
            pyro.sample(    #E
                    "Z",    #E
                    dist.Normal(z_loc, z_scale).to_event(1)    #E
            )    #E
```

#A guide is a method of the VAE which will use the encoder.
#B Register the encoder so Pyro is aware of its weight parameters.
#C This is the same plate context manager for iterating over the batch data that we see in the module.
#D Use the encoder to map an image and its labels to parameters of a Normal distribution
#E Sample Z from that normal distribution

During training, autoencoder architectures iteratively pass training images through the encoder to generate an encoding, then pass that encoding through the decoder, which tries to reconstruct the original image. The training procedure optimizes the parameters of the decoder and encoder such that the reconstructed image looks as close to the original image as possible. We can write a method that does this reconstruction manually.

Listing 5.6 Define a helper function for reconstructing and viewing the images

```
    def reconstruct_img(self, x, digit, is_handwritten):    #A
        z_loc, z_scale = self.encoder(x, digit, is_handwritten)    #B
        z = dist.Normal(z_loc, z_scale).sample()    #C
        loc_img = self.decoder(z, digit, is_handwritten)    #D
        return loc_img
```

#A reconstruct_img is a method of the VAE class
#B A Encode image x and the labels
#C Sample a latent Z encoding variable
#D Decode the image from the latent Z. There's no need to use the decoded value to sample from the Bernoulli.

We combine these elements into one PyTorch neural network module.

Listing 5.7 Full VAE class

```
class VAE(nn.Module):
    def __init__(self, z_dim=50, hidden_dim=400, use_cuda=False):    #A
        super().__init__()
        self.encoder = Encoder(z_dim, hidden_dim)    #B
        self.decoder = Decoder(z_dim, hidden_dim)    #B

        if use_cuda:    #C
            self.cuda()    #C
        self.use_cuda = use_cuda
        self.z_dim = z_dim

    def model(self, x, digit, is_handwritten):    #D
        pyro.module("decoder", self.decoder)    #D
        batch_size = x.size(0)    #D
        options = dict(dtype=x.dtype, device=x.device)    #D
        with pyro.plate("data", x.shape[0]):    #D
    #D
            z_loc = x.new_zeros(torch.Size((x.shape[0], self.z_dim)))    #D
            z_scale = x.new_ones(torch.Size((x.shape[0], self.z_dim)))   #D
            z = pyro.sample(    #D
                    "Z",    #D
                    dist.Normal(z_loc, z_scale).to_event(1)    #D
            )    #D
                 #D
            digit_prob = torch.ones(batch_size, 10, **options) / 10.0    #D
            digit = pyro.sample(    #D
                    "digit",    #D
                    dist.OneHotCategorical(digit_prob),    #D
                    obs=digit    #D
             )    #D
    #D
            hw_prob = torch.ones(batch_size, 2, **options) / 2.0    #D
            is_handwritten = pyro.sample(    #D
                    "is_handwritten",    #D
                    dist.OneHotCategorical(hw_prob),    #D
                    obs=is_handwritten    #D
            )    #D
                #D
            X_img_param = self.decoder.forward(    #D
                    z,    #D
                    digit,    #D
                    is_handwritten    #D
            )    #D
            X_img = pyro.sample(    #D
                 "obs",    #D
                 dist.Bernoulli(X_img_param).to_event(1),    #D
                 obs=x.reshape(-1, 784)    #D
            )    #D
        return X_img    #D

    def guide(self, x, digit, is_handwritten):    #E
        pyro.module("encoder", self.encoder)    #E
        with pyro.plate("data", x.shape[0]):    #E
            z_loc, z_scale = self.encoder(x, digit, is_handwritten)    #E
            pyro.sample(    #E
                    "Z",    #E
                    dist.Normal(z_loc, z_scale).to_event(1)    #E
```

```
            )    #E

    def reconstruct_img(self, x, digit, is_handwritten):    #F
        z_loc, z_scale = self.encoder(x, digit, is_handwritten)    #F
        z = dist.Normal(z_loc, z_scale).sample()    #F
        loc_img = self.decoder(z, digit, is_handwritten)    #F
        return loc_img    #F
```

#A Our latent space variable Z is 50-dimensional and we use 400 hidden units.
#B Initialize the encoder and decoder networks.
#C Calling cuda() here will put all the parameters of the encoder and decoder networks into gpu memory.
#D The model.
#E The guide.
#F Image reconstructor.

The variational autoencoder minimizes the difference between the target distribution P(Z|X, is_handwritten, digit) and the *variational distribution* Q(Z|X, is_handwritten, digit) that it is meant to approximate the target. It does this by minimizing the *KL-divergence* between the two distributions; the KL-divergence is a way of quantifying how two distributions differ. Mathematically, KL-divergence has a lower bound called the *expected lower bound* or ELBO. Minimizing the KL-divergence is equivalent to maximizing ELBO. Pyro implements a utility `Trace_ELBO` that calculates a ELBO-based loss function during training. The key parts of the training procedure are as follows:

Listing 5.8 Key elements of the training algorithm

```
from pyro.infer import TraceELBO    #A
from pyro.infer import SVI    #B
from pyro.optim import Adam    #C

vae = VAE()    #D
optimizer = Adam({"lr": 1.0e-3})    #E
svi = SVI(vae.model, vae.guide, optimizer, loss=Trace_ELBO())    #F

…
epoch_loss += svi.evaluate_loss(x, digit, is_handwritten)    #G
```

#A A utility for calculating loss.
#B The stochastic variational inference (SVI) class for representing the variational training objective.
#C The Adam optimizer will use gradient-based search for values of the parameters in the decoder and encoder that optimize the loss function.
#D First we initialize the VAE,
#E and the optimizer (here, 'lr' is a training parameter called *learning rate*),
#F We initialize SVI,
#G then for every epoch, for every training image and its corresponding labels in a batch of data, we calculate and aggregate the loss.

The full training procedure is as follows.

Listing 5.9 The training procedure

```
LEARNING_RATE = 1.0e-3    #A
USE_CUDA = False    #A
NUM_EPOCHS = 1000    #A
TEST_FREQUENCY = 5    #A

def train(svi, train_loader, use_cuda=False):
    epoch_loss = 0.    #B
    for x, digit, is_handwritten in train_loader:    #C
        if use_cuda:    #D
            x = x.cuda()    #D
        digit = torch.nn.functional.one_hot(digit, 10)    #E
        is_handwritten = torch.nn.functional.one_hot(is_handwritten, 2)  #E
        epoch_loss += svi.step(x, digit, is_handwritten)    #F

    normalizer_train = len(train_loader.dataset)    #G
    total_epoch_loss_train = epoch_loss / normalizer_train    #G
    return total_epoch_loss_train    #G

def evaluate(svi, test_loader, use_cuda=False):    #H
    test_loss = 0.    #H
    for x, digit, is_handwritten in test_loader:    #I
        if use_cuda:    #J
            x = x.cuda()    #J
        digit = torch.nn.functional.one_hot(digit, 10)    #K
        is_handwritten = torch.nn.functional.one_hot(is_handwritten, 2)  #K
        test_loss += svi.evaluate_loss(x, digit, is_handwritten)
    normalizer_test = len(test_loader.dataset)
    total_epoch_loss_test = test_loss / normalizer_test
    return total_epoch_loss_test

def plot_regenerate(x, digit, is_handwritten):    #L
    fig = plt.figure    #L
    plt.imshow(x.view(28, 28).cpu().data.numpy(), cmap='gray')    #L
    plt.show()    #L
    x_generated = vae.reconstruct_img(x, digit, is_handwritten)    #L
    plt.imshow(x_generated.view(28, 28).cpu().data.numpy(), cmap='gray') #L
    plt.show()    #L

train_loader, test_loader = setup_data_loaders(
    batch_size=256, use_cuda=USE_CUDA
)

pyro.clear_param_store()    #M
vae = VAE(use_cuda=USE_CUDA)    #N

adam_args = {"lr": LEARNING_RATE}    #O
optimizer = Adam(adam_args)    #O

svi = SVI(vae.model, vae.guide, optimizer, loss=Trace_ELBO())    #P

train_elbo = []
test_elbo = []

for epoch in range(NUM_EPOCHS):    #Q
    total_epoch_loss_train = train(svi, train_loader, use_cuda=USE_CUDA) #Q
    train_elbo.append(-total_epoch_loss_train)    #Q
    print("[epoch %03d] average training loss: %.4f" % (epoch, total_epoch_loss_train))  #Q
```

```
    if epoch % TEST_FREQUENCY == 0:    #R
        total_epoch_loss_test = evaluate(    #R
svi, test_loader, use_cuda=USE_CUDA    #R
)    #R
        test_elbo.append(-total_epoch_loss_test)   #R
        print("[epoch %03d] average test loss: %.4f" % (epoch, total_epoch_loss_test))   #R
        test_samples = random.choices(list(test_loader), k=3)   #R
        for x_test, digit_test, is_handwritten_test in test_samples:    #R
            x_shaped = x_test[0][0].reshape(1, 28*28)    #R
            digit_shaped = torch.nn.functional.one_hot(digit_test[0], 10)[None, :]    #R
            is_handwritten_shaped = torch.nn.functional.one_hot(is_handwritten_test[0],
      2)[None, :]    #R
            plot_regenerate(    #R
x_shaped, digit_shaped, is_handwritten_shaped    #R
)    #R
            print("-------------------------------")   #R
```

#A Training options.
#B Initialize the loss accumulator.
#C Do a training epoch over each mini-batch of images returned by the data loader.
#D If on GPU put mini-batch into CUDA memory.
#E Convert digit and in_handwritten to one-hot encoding.
#F Calculate ELBO gradient and accumulate loss.
#G Return epoch loss.
#H Initialize loss accumulator.
#I Compute the loss over the entire test set.
#J If GPU is True, put mini-batch into CUDA memory
#K Convert labels to one-hot encoding.
#H Compute ELBO estimate and accumulate loss.
#L Plot original images alongside regenerated images.
#M Clear the parameter store.
#N Setup the VAE.
#O Setup the optimizer.
#P Setup the inference algorithm.
#Q Main training loop.
#R Report test diagnostics.

This code will train the full VAE. Visit www.altdeep.ai/p/causalmlbook for a link to a Jupyter notebook with the full code.

The code will train the parameters of the encoder that maps images and the labels to the latent variable. It will train the decoder that maps the latent variable and the labels to the image. That latent variable is a fundamental feature of the VAE. But we should take a closer look at how to interpret the latent variable in causal terms.

5.2.2 How should we causally interpret Z?

I said we can view Z as a "stand-in" for all the independent latent causes of the object in the image. Z is a representation we learn from the pixels in the images. It is tempting to treat that representation like a higher-level causal abstraction of those latent causes. But it is probably not doing a great job as a causal abstraction. The autoencoder paradigm trains an encoder that can take an image and embed it into a low-dimensional representation Z. It tries to do so in a way that it can reconstruct the original image as well as possible. In order to

reconstruct the image with little loss, the framework tries to encode as much information in the original image as it can in that lower dimensional representation.

A good *causal* representation, however, shouldn't try to capture as much information as possible. Rather, it should strive to capture *only* the *causal* structure in the images and ignore everything else. Solving this problem is a challenge, particularly as the definition of "causal structure" can be domain specific. We investigate causal representation learning in chapter 9.

5.2.3 Using the trained deep causal model

You can visit www.altdeep.ai/p/causalmlbook for a link to full notebook tutorial where you can run this code that trains this VAE. Figure 5.12 illustrates decreasing loss on the test data (in this implementation loss is just negative expected lower bound) as we train the VAE.

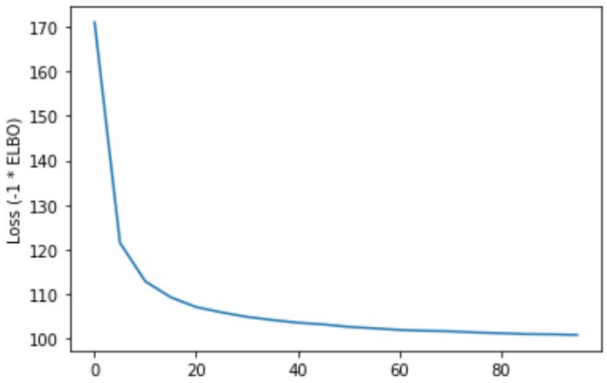


Figure 5.12 Loss on the test data, which is -1 times the ELBO. Test loss was calculated every five epochs.

Figure 5.13 illustrates how well model can reconstruct (encode and then decode) the original image after a modest 100 epochs.

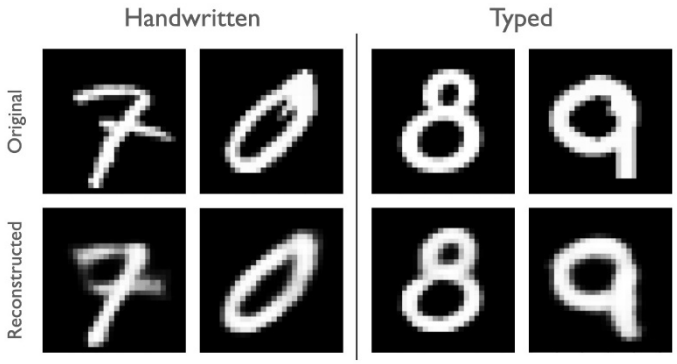


Figure 5.13 Results of the training after 100 epochs. The top row contains images in the test set, bottom row is the reconstructed image. The left two columns are handwritten (MNIST) images, the right two columns are typed (TMNIST) digits.

So what can we do with this trained model? So far, what we've demonstrated is how to use this VAE to fit the conditional probability functions entailed by the causal DAG. We could similarly use another deep probabilistic machine learning framework, such as a generative adversarial network (GAN).

The next step would be to use a model like this to make actual causal inferences. But we still have some conceptual groundwork to lay before we do that. In later chapters, as we learn how to do causal reasoning with a causal generative model, we'll revisit this type of deep neural causal model and demonstrate how to use them to make causal inferences.

5.3 Using causal inference to enhance deep learning

In this section, I illustrate one common example of how causality helps with understanding when we'd expect a deep learning technique (or another machine learning technique) to work and when it wouldn't. This example is taken from a paper called *On Causal and Anti-Causal Learning*, by Schölkopf, Janzing, Peters, Sgouritsa, Zhang, and Mooij; see the chapter notes at www.altdeep.ai/p/causalmlbook for this and related citations.

5.3.1 Case study: Causality and semi-supervised learning

Suppose we have two variables X and Y. We are interested in using X to predict Y. In machine learning terms, X is a feature, and Y is a label.

In order to predict Y from X, the predictive algorithm has to learn something about the conditional probability distribution of Y given X, i.e. P(Y|X). Perhaps it learns a representation of the entire distribution, or perhaps it just focuses on how to get highly probable values of Y for a given X.

In supervised learning, the training data consists of N samples of X, Y pairs; (x_1, y_1), (x_2, y_2), …, (x_N, y_N). The data is "supervised" because every x is paired with a y. We can use these pairs to learn P(Y|X).

In unsupervised learning, the data is unsupervised, meaning we have no labels, no observed value of Y. Our data looks like (x_1), (x_2), …, (x_N). With this data alone we can't learn anything about the P(Y|X), we can only learn about P(X), the marginal distribution of X.

Semi-supervised learning asks the question, supposed we had a combination of supervised and unsupervised data. Could these two sets of data be combined in a way such that our ability to predict Y was better than if we only used the supervised data? In other words, can the unsupervised data somehow augment the learning of the supervised data?

Semi-supervised learning works in humans. If you took a child to the zoo and point out a few examples of mammals and non-mammals, you would find that seeing additional animals would enhance their ability to distinguish between mammals and non-mammals even if you didn't tell them which was which. Further, the semi-supervised question is quite practical. It is common to have abundant unsupervised examples while labeling those examples is costly. For example, suppose you worked at a social media site and were tasked with building an algorithm that classified whether or not an uploaded image contained sexually explicit content. The first step is to create supervised data by having humans manually label images as explicit or not. Not only does this cost many people-hours, but it is psychically stressful for the labelers. A successful semi-supervised approach would mean you could minimize the amount of labeling needed to achieve a required level of classification accuracy.

We can use causal reasoning to gauge how effective semi-supervised learning might perform. For semi-supervised learning to work, the needs to be some connection between the P(X) we can learn from the unsupervised data and the properties of P(Y|X) we can learn from the supervised data. Causality can help us understand when that might be possible.

To see this, let's limit ourselves to two possibilities, one where X is a cause of Y and one where Y is a cause of X. In both cases, our goal is to learn as much as we can about P(Y|X). We'll assume there are no other variables in the data generating process to concern ourselves with.

Let's consider the first case where X (the feature) is the cause of Y (the label). Thus, our causal DAG is $X \rightarrow Y$. So we'll call this the *causal learning case* because the direction of the prediction is from the cause to the effect., as illustrated in Figure 5.14.

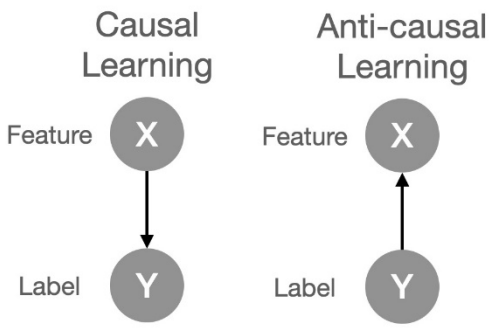


Figure 5.14 In causal learning, the features cause the label. In anti-causal learning, the label causes the features.

According to the Markov factorization property, the joint distribution P(x, y) factorizes into P(x)P(y|x). As we discussed in chapter 3, we expect causally Markovian factors P(X) and P(Y|X) to obey a causal invariance principle called "independence of mechanism." The principle is P(Y|X) and P(X) each represent two distinct causal mechanisms in the data-generating process, one where X is assigned its value, and one where Y is assigned its value given X.

In chapter 3, I discussed an example about elevation and temperature. We saw basic knowledge of physics tells us that the mechanism by which a city's elevation affects temperature is the same regardless of whether we're in a high-elevation city like Lhasa or a coastal city. Further, knowing the distribution of elevations across cities doesn't tell me anything about *how* elevation affects urban temperatures. And if I know how elevation affects urban temperatures, that doesn't tell me anything about the variation of elevations across cities.

The consequence of independence of mechanism is *parameter modularity*; if P(X) and P(Y|X) represent independent mechanisms, then the parameters the algorithm learns for P(X) should be decoupled from the parameters my algorithm learns for P(Y|X). That means, any observed crosstalk between the learned representations of P(X) and P(Y|X) should be illusionary, signs of overfitting. The unsupervised data can only help learn a representation of P(X). Thus, in this causal learning case, semi-supervised learning should be no better than supervised learning since that learned representation of P(X) should not inform P(Y|X).

Now consider the case where Y (the label) is the cause of X (the feature). The causal DAG is X ← Y. We'll call this the *anti-causal learning* case because now we're trying to look backward from effects to predict causes. In this case, P(x)P(y|x) is *not* the Markov factorization of the DAG, ergo P(X) and P(Y|X) does *not* represent distinct causal mechanisms. Thus, we do *not* expect parameter modularity to hold for the algorithm's representations of P(X) and P(Y|X), and therefore the unsupervised data should help inform the representation of P(y|x) in the case anti-causal learning.

5.3.2 Takeaways from semi-supervised learning case

Your takeaway should *not* be that there is some "causal semi-supervised learning" that requires you to make all sorts of strong causal assumptions about the data. The *causal learning* case (feature vector X is a cause of the labels Y) and the *anti-causal learning case* (labels Y is a cause of the feature vector) is an intentional simplification of typical practical settings. In practical settings, it is common for some of the elements of the feature vector to be causes of the label and some to be effects of the label. Other elements of the feature vector may correlate with the label by way of an unknown and unmeasured (latent) common cause.

The key takeaway is semi-supervised learning is eminently useful but that sometimes it worked well than other times less well. Causal analysis helps us understand when it should work and when it shouldn't. That can help the modeler make better decisions about whether or not to spend her time and resources applying semi-supervised learning in her problem domain. Further, she can use the insight to implement causally informed semi-supervised learning algorithms, for example, one that gains efficiency by excluding features in the unsupervised data that are likely causes of the label.

5.3.3 What it looks like when causality helps deep learning

AI researcher Ali Rahimi has compared modern machine learning to alchemy. While controversial, the simile is useful for understanding how causal methods can enhance deep learning and what it looks like when they succeed in doing so.

Rahimi's comparison to alchemy wasn't pejorative; he points out that alchemy "worked."

> **Alchemy worked. Alchemy invented metallurgy, ways to dye textiles, modern glass-making processes, and medications. Then again, alchemists also believed they could cure diseases with leeches and transmute base metals into gold.**

In other words, alchemy works, but alchemists lacked understanding of the underlying scientific principles that made it work when it did. That made it hard to know when it would fail. As a result, alchemists wasted considerable effort on dead-ends (philosopher's stones, immortality elixirs, etc.).

Similarly, deep learning also "works" in that it achieves good performance on a wide variety of prediction and inference tasks. But we often have an incomplete understanding of why and when it works. That lack of understanding has led to problems with reproducibility, robustness, and safety. Like the alchemists, machine learning engineers and researchers spend significant resources, including many millions of dollars of corporate, government, and investor capital, on dead-ends. Some of these dead-ends have adverse externalities, such as published work on that attempt to predict behavior (e.g., criminality) from profile photos. Such efforts are the machine learning analog of the alchemical immortality elixirs that contained toxins like mercury; they don't work *and* they cause harm.

The semi-supervised learning case gives us a template for how causality can remedy this alchemy problem. Semi-supervised learning often worked, but it wasn't clear for which problems it would work well and which it would work less well. Causal analysis helped remedy this problem. In general, the pattern is as follows:

1. An algorithm has demonstrated impressive performance on a benchmark.
2. However, sometimes it's hard to reproduce that performance and it's not clear why.
3. Causal analysis helps provide insight into why.
4. This helps the modeler make better decisions about whether and when to spend their time and resources on this algorithm.
5. Further, the causal insights can help the modeler tweak the algorithm to be more robust and efficient.

The last item takes many shapes. For example, causal insights might identify hyperparameters, elements of neural net architecture, and boilerplate configuration and training code that don't contribute to general performance. Often, these enhancements do *not* improve performance beyond the impressive numbers reported in the original publication and PR release. But they help reliably reproduce good performance across various settings.

5.3.4 AI Alchemy, and making "superhuman" performance reliable

ROBUST "SUPERHUMAN" PERFORMANCE

We often hear about the "superhuman" performance of deep learning. Speaking of superhuman ability, imagine an alternative telling of Superman's origin story. Imagine if, when Superman made his first public appearance, his superhuman abilities were unreliable? He demonstrated astounding superhuman feats like flight, super strength, and laser vision. Yet, sometimes his flight ability failed and his super strength faltered. Sometimes his laser vision was dangerously unfocused resulting in terrible collateral damage. The public would be impressed and hopeful that he could do some good, but unsure if it would be safe to rely on this guy when the stakes were high.

Now imagine that his adoptive Midwestern parents, experts in causal inference, used causal analysis to study why the superpowers fluctuate and to engineer a pill that stabilizes the superpowers. Note, this is not a pill that enhances Superman's powers, just one that makes them more reliable. The work of developing that pill would get less headlines than flight and laser vision, but it would be the difference between merely having superpowers, and being Superman.

This analogy helps us understand the role causal methods often take in enhancing deep learning and other machine learning methods. We see deep learning achieve superhuman results in certain circumstances. But then the performance seems to be unstable across different settings. Causal analysis can often help explain what conditions are required for that superhuman performance. If modeler can tell in advance that those conditions cannot be met, she saves time, effort, computational resources, and money. In other cases, causal analysis can show us how to tweak the deep learning algorithm to achieve those conditions required for performance. The deep learning approach becomes more robust, reproducible, explainable, and engineerable.

5.3.5 Causality can help by formalizing "inductive bias"

We can also use the concept of "inductive bias" to understand how causal reasoning enhances deep learning. "Inductive bias" refers to elements of an inference algorithm that makes it prefer certain inferences or predictions over others. Examples of inductive bias include "Occam's Razor" and the assumption in forecasting that trends in the past will continue into the future.

Modern deep learning relies on using neural network architectures and training objectives to encode inductive bias. For example, "convolutions" and "max pooling" are architectural elements in convolutional neural networks that encode inductive biases called "translation invariance"; i.e., a kitten is still a kitten regardless of whether it appears on the left or right of an image. However, insights into the connection between neural network architectures and inductive biases tend to be more heuristic than formal (though formalizing inductive bias is an active area of AI research).

Causal inference also relies heavily on inductive biases, though most researchers in the domain would simply say "assumptions" and "constraints." But, in contrast to deep learning's inductive biases, causal inference relies on explicit and formal assumptions (such as a causal DAG). Explicit and formal inductive biases enable formal reasoning (such as with d-separation)

that lead to the formal guarantees about when you get causality from correlation. That emphasis on formalism exists because the stakes for causal inferences are higher than for predictive inferences; if you publicly claim your vaccine prevents a disease, you'd better be right.

But deep learning can leverage causality's formalism to achieve more robust results. For example, inductive biases in deep learning are often described in terms of *invariances*, meaning elements of how the data is generated that we expect to be consistent across datasets. Causal analysis can reveal the causal sources of invariance, which in turn help learn more robust representations and make more robust predictions.

5.4 Summary

- Deep learning can be used to enhance causal modeling and inference.
- In particular, causal modeling can leverage the ability of deep learning to scale, deal with high-dimensional nonlinear relationships.
- Deep learning and help learn representations of high-level causal abstractions, though this is hard.
- The variational autoencoder framework allows you to fit a causal DAG with deep learning.
- The framework allows you to use deep learning to represent a causal Markov kernel for an image distribution.
- The image is the effect, the labels representing the content of the image are causes.
- The decoder maps the causes and a latent variable to the image variable. The encoder maps the image variable back to the latent variable.
- We can view the learned representation of the latent variable as a stand-in for unmodeled causes, but it lacks the qualities we'd expect from a causal representation.
- Causality often enhances deep learning and other machine learning methods by helping elucidate the underlying principles that make it work.
- For example, causal analysis shows semi-supervised learning should work in the case of *anti-causal learning* (when the features are *caused by* the label) but not in the case of *causal learning* (when the features cause the label).
- Such causal insights can help the modeler avoid spending time, compute, person-hours, and other resources on a given algorithm when it is not likely to work in a given problem setting.
- It can also help make the algorithm more robust and efficient.