CHAPTER 2

Foundations for MLOps Systems

"All models are wrong, but some are useful."

—George Box

In this chapter, we will discuss foundations for MLOps systems by breaking down the topic into fundamental building blocks that you will apply in future chapters. While we will discuss programming nondeterministic systems, data structures and algorithmic thinking for data science, and how to translate thoughts into executable code, the goal is not to give a fully comprehensive introduction to these areas in a single chapter but instead provide further resources to point you in the right direction and answer an important question: Why do you need to understand mathematics to develop and deploy MLOps systems?

This book would be remiss without laying out the core mathematical and computational foundations that MLOps engineers need to understand to build end to end systems. It is the responsibility of the MLOps engineer to understand each component of the system even if it appears like a "black box."

Toward this end, we will create a logistic regression model (both classical and Bayesian) from scratch piece by piece to estimate the parameters of the hypothesis using stochastic gradient descent to illustrate

D. Sorvisto, *MLOps Lifecycle Toolkit*, https://doi.org/10.1007/978-1-4842-9642-4_2

how models built up from simple mathematical abstractions can have robust practical uses across various industries. First, let's define what we mean by model by taking a look at some statistical terminology.

Mathematical Thinking

Mathematics is the foundation of data science and AI. Large language models like ChatGPT are transforming our lives. Some of the first large language models such as BERT (an encoder) are based on either an encoder or decoder transformer architecture. While the attention layers that are part of this model are different both in form and in use cases, they are still governed by mathematics.

In this section, we lay out the rigorous foundations for MLOps, diving into the mathematics behind some of the models we use.

Linear Algebra

Linear algebra is the study of linear transformations like rotation. A transformation is a way of describing linear combinations of vectors. We can arrange these vectors in a matrix form, and in fact you can prove every linear transformation can be represented in this way (with respect to a certain basis of vectors). You'll find linear algebra used throughout applied mathematics since many natural phenomena can be modeled or approximated by linear transformations. The McCulloch-Pitts neuron or perceptron combines a weight vector with a feature vector using an operator called the dot product. When combined with a step activation or "threshold" function, you can build linear classifiers to solve binary classification problems.

Though matrices are two-dimensional, we can generalize the idea of a matrix to higher dimensions to create tensors. Since many machine learning algorithms can be written in terms of tensor operations. In

fact tensors themselves can be described as multilinear maps[1]. You can imagine how important linear algebra is to understanding neural networks and other machine learning algorithms. Another important reason for studying linear algebra is it is often the first exposure to writing proofs, developing mathematical arguments and mathematical rigor.

Probability Distributions

By model we really mean a probability distribution. The probability distribution will be parameterized so that we can estimate it using real-world data either through algorithms like gradient descent or Bayes' rule (this may be difficult under some circumstances as we'll discuss). We're usually interested in two types of probability distributions: joint probability distributions and conditional distributions.

Join probability distribution: Given two random variables X and Y, if X and Y are defined on the space probability space, then we call the probability distribution formed by considering all possible outcomes of X and Y simultaneously the *joint probability distribution*. This probability distribution written as P(X, Y) encodes the marginal distributions P(X) and P(Y) as well as the conditional probability distributions. This is an important concept as many of the models we'll be looking at will be attempting to compute or sample from a joint probability distribution to make some prediction.

Conditional probability distribution: Conditional probability is the probability of an event, Y occurring given an event X has already occurred. We write this conditional probability as P(Y | X) often read as "probability of Y given X." Let's look at a few examples of models we might use as data scientists in various industries to understand how these abstractions are built up from mathematical concepts.

[1] An introduction to linear algebra can be found in Hoffman, K. A. (1961). *Linear Algebra*.

Understanding Generative and Discriminative Models

A generative model is synonymous with a joint probability distribution P(X, Y) (however, this is not strictly true since, e.g., GANs belong to the class of generative models) since for a classification problem it will assume some functional form of P(Y) and P(X | Y) in terms of some parameters and estimated from the training data. This is then used to compute P(Y | X) using Bayes' rule. These types of models have some interesting properties, for instance, you can sample from them and generate new data. Data augmentation is a growing area especially within the healthcare and pharmaceutical industry where data from clinical trials is costly or not available.

The simplest examples of a generative model include Gaussian distributions, the Bernoulli model, and Naive Bayes models (also the simplest kind of Bayesian network).

In contrast, a discriminative model such as logistic regression makes a functional assumption about the form of P(Y | X) in terms of some parameters W and b and estimates the parameters directly from the training data. Then we pick the most likely class label based on these estimates. We'll see how to compute parameters W and b in the lab: algorithmic thinking for data science[2] where we'll actually use stochastic gradient descent and build a logistic regression model from the ground up.

[2] For a full introduction to algorithmic thinking and computer programming, the reader is directed to Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs, second edition*. MIT Press.

Bayesian Thinking

We chose logistic regression as an example in this chapter for another reason: Logistic regression is a good example of a probabilistic model. When you train it, it automatically gives you an estimate of the probability of success for new data points. However, classical logistic regression is a *frequentist* model. The classical logistic regression model does not tell us if we can rely on the results, if we have enough data for training, or anything about the certainty in the parameters.

To illustrate this point, let's suppose we train a logistic regression model to predict who should receive a loan. If our training data is imbalanced, consisting of 1000 people and 900 of which are examples of people we should not lend to, our model is going to overfit toward a lower probability of loan approval, and if we ask what is the probability of a new applicant getting a loan, the model may return a low probability. A Bayesian version of logistic regression would solve this problem. In the lab, you will solve this problem of imbalance data by using a Bayesian logistic regression model and generating a trace plot to explore the parameters and ensure that the parameters are well calibrated to the data. Figure 2-1 shows a trace plot generated from this lab.

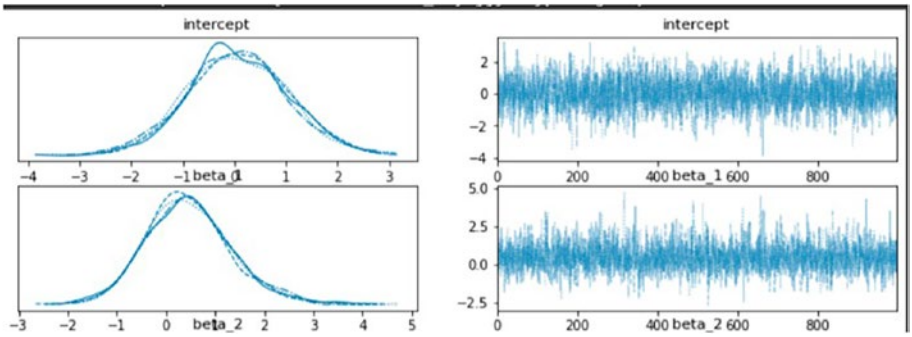

Figure 2-1. *A trace plot showing the history of parameters in a Bayesian model*

Of course, we need to understand yet another mathematical primitive: Bayes' rule. Unlike in frequentist statistics, where we have parameters and point estimates, in Bayesian statistics, we have probability distributions as we defined earlier. In fact, every unknown in our model is a probability distribution called a prior that encodes our current knowledge about that parameter (in the lab, we have three parameters we want to estimate, with priors chosen from normal distributions).

Bayes' rule updates beliefs about the parameters by computing a posterior probability distribution.

- The **prior** distribution can be interpreted as the current knowledge we have on each parameter (it may only be a best guess).
- The **likelihood function** is the probability of observing a data set given certain parameters θ of our model.
- The **evidence** is the probability of the observed data itself over all possible models and is very difficult to compute, often requiring multivariate integrals in three or more dimensions. Fortunately, for many problems, this is only a constant of proportionality that can be discarded[3].

We speak of "turning the Bayesian crank" when the posterior of one problem (what we are interested in estimating) becomes the prior for future estimates. This is the power of Bayesian statistics and the key to generative models. Listing 2-1 shows the different parts of Bayes' rule.

[3] Hoff, P. D. (2009). A First Course in Bayesian Statistical Methods. In *Springer texts in statistics*. Springer International Publishing. https://doi.org/10.1007/978-0-387-92407-6.

Listing 2-1. Bayes' rule

$$\mathrm{P}(\theta \mid X, y) = \frac{P(y|X,\theta)P(\theta)}{P(y|X)}$$

Bayes rule was actually discovered by Thomas Bayes, an English Presbyterian minister and statistician in the eighteenth century, but the work *LII. An Essay Towards Solving a Problem in the Doctrine of Chances* wasn't published until after Bayes' death and to this day is often a graduate level course not taught in undergraduate statistics programs.

So how can we develop some intuition around Bayes' rule? Let's start by asking a question:

What is the probability of a coin coming up heads? Take a few minutes to think about it before you answer; it's a bit of a trick question.

Okay …I've asked this question to a few people and most would say it depends. It depends if the coin is fair or not. Ok so assuming it's a fair coin, the usual answer is the chance of coming up heads is 50% or 0.5 if we're using probability.

Now let's switch this question up; let's suppose that the coin has already been flipped but you cannot see the result. What is the probability? Go ahead and ask your colleagues or friends this question, and you might be surprised by the range of answers you'll receive.

A *frequentist* position is that the coin has already been flipped, and so it is either heads or tails. The chance is either 0% heads if the coin landed tails or it is 100% if it landed heads. However, there's something unsatisfactory about this perspective; it does not take into consideration the uncertainty in the model.

A Bayesian approach would be to quantify that uncertainty and say, it's still 50% chance of heads and 50% chance of tails; it depends on *what we know* at the moment. If we observe the coin has landed heads, then we can update our hypothesis. This allows us to adapt to change and accommodate new information (remember, MLOps is all about being able to adapt to change). In the next section, we will look at some specific examples of Bayesian models.

Gaussian Mixture Models

K-means, one of the oldest clustering methods, behaves poorly when clusters are of different sizes, shapes, and densities. While K-means requires knowing the number of clusters as a parameter before going ahead with clustering, it is closely related to nonparametric Bayesian modeling in contrast to the Gaussian mixture model (GMM) shown in Figure 2-2.

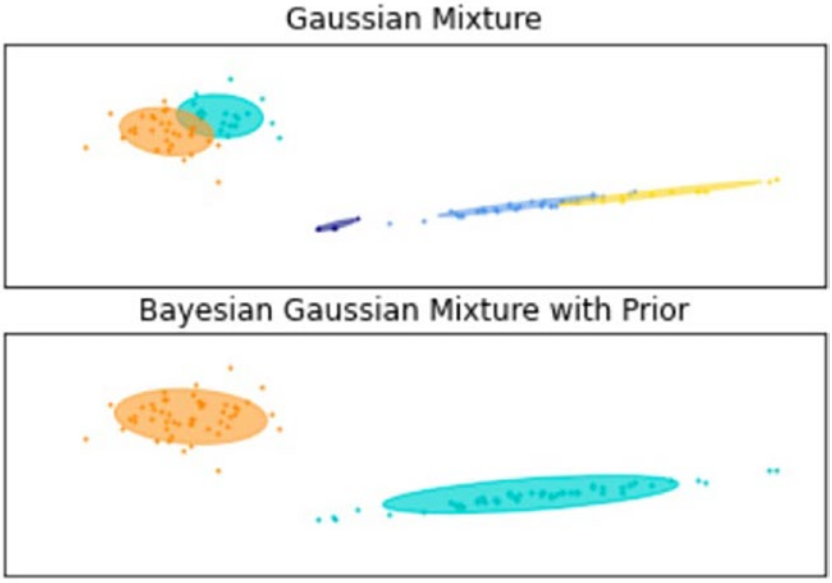

Figure 2-2. *Clustering using a Bayesian Gaussian mixture model*

A Gaussian mixture model is a probabilistic model that makes the assumption that the data generating process is a mixture of finite Gaussian distributions (one of most important probability distributions for modeling natural phenomena and so widely used in science, engineering, and medicine). It is a parametric model where the parameters of the Gaussian components are unknown. We can think of GMMs as a finite weighted sum of Gaussian component densities. Listing 2-2 shows an equation that governs the GMM.

Listing 2-2. An equation that describes the Gaussian mixture model

$$p(x) = \sum_{i=1}^{m} \theta_i \mathcal{N}\left(\boldsymbol{x} \mid \boldsymbol{\mu}_i, \sum_i\right)$$

However, these parameters are computationally expensive and do not scale well since they are computed often through maximal likelihood and EM (expectation-maximization) algorithms and are modeled as latent variables. Despite scalability problems, GMMs are used in many industries in particular healthcare, to model more meaningful patient groupings, diagnostics, and rehabilitation and to support other healthcare activities.

General Additive Models

With generalized additive models (GAMs), you don't have to trade off accuracy for interpretability. These are a very powerful extension to linear regression and are very flexible with their ability to incorporate nonlinear features in your data (imagine having to do this by hand if all we had was a linear regression model?)

If random forests are data driven and neural networks are model driven, the GAMs are somewhere in the middle, but compared to neural nets, SVMs, or even logistic regression, GAMs tend to have relatively low misclassification rates which make them great for mission critical applications where interpretability and misclassification rate are utmost importance such as in healthcare and financial applications.

If you've never used a GAM before, you can look at splines to start. Splines are smooth functions used to model nonlinear relationships and allow you to control the degree of smoothness through a smoothing parameter. Figure 2-3 shows some of the trade-offs between these different models.

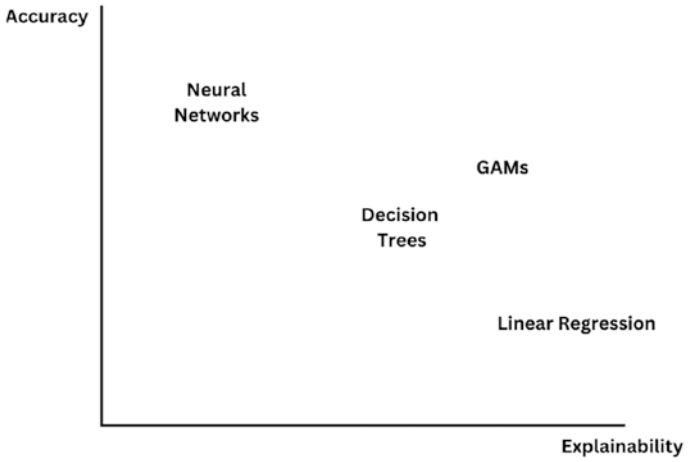

Figure 2-3. *Trade-offs between explainability and model accuracy*

Kernel Methods

The best known example of kernel methods, the support vector machine (SVM), allows us to use the "kernel trick" and work well on text based data which is naturally high dimensional. This means that we can embed features in a higher, possibly infinite, dimensional space without ever having to explicitly compute the embedding.

Kernel methods are particularly important in geostatistics applications such as kriging (Gaussian process regression) especially in the oil and gas industry. The main use case of kriging is to estimate the value of a variable over a continuous spatial field. For example, you may have sensor readings such as temperature and pressure in an oil and gas reservoir, but you may not know the sensor readings at every position in the reservoir. Kriging provides an inexpensive way to estimate the unknown sensor readings based on the readings that we do know. Kriging uses a covariance matrix and a kernel function to model the spatial relationships and spatial dependencies of data points throughout the reservoir by encoding similarity between features.

Where did these models come from? Fundamentally, these algorithms are based on logic and mathematical properties and primitives such as probability distributions. If we know about the Gaussian distribution, it's not a far stretch to understand how to build a Gaussian mixture model. If we know about covariance matrices, we can understand kernel methods and maybe Gaussian processes, and if we understand linear systems, we can build on top of this abstraction to understand GAMs.

I want to illustrate this point further, by building a model from fundamental principles. Logistic regression is particularly interesting to use for this as most people are familiar with it, but it is not a toy model; you can use logistic regression to solve many real-world problems across various industries since it is fairly robust. We can also use logistic functions to build many more complex models. For instance, the classical version of logistic regression is used to model binary classification (e.g., predicting likelihood of success or failure), but by combining multiple logistic regression models into strategies like one-vs.-all or one-vs.-one, we can solve more complex multi-class classification problems with a single response variable via softmax, a generalization of logistic regression. Logistic functions can also be used to create neural networks: It's no coincidence that in a neural network, the activation function is often a logistic sigmoid (pictured in the following). In this chapter's lab on algorithmic thinking, you're going to walk through some of these mathematical tools and build a logistic regression model from scratch. Figure 2-4 shows an example of a logistic sigmoid curve.

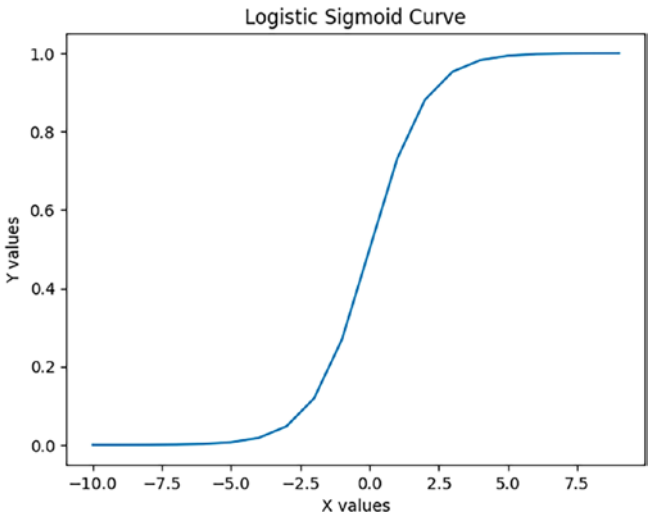

Figure 2-4. *A logistic sigmoid curve*

The parameter mu takes on the probability value ½ which makes intuitive sense.

Higher Dimensional Spaces

I want to cover one more mathematical tool for the toolkit because in data science, unlike pure statistics, we deal with big data but also because it is fascinating to be able to develop intuition on higher dimensional spaces by learning to think geometrically.

You've probably heard of the "curse of dimensionality." Things behave strangely in high dimensions, for example, if we could measure the volume of a unit sphere as we embed it into higher dimensional space, that volume would actually shrink as the dimension increases! That is incredibly counterintuitive. Figure 2-5 shows an artistic rendition of a shrinking sphere in higher dimensional space (since we can only visualize in three dimensions).

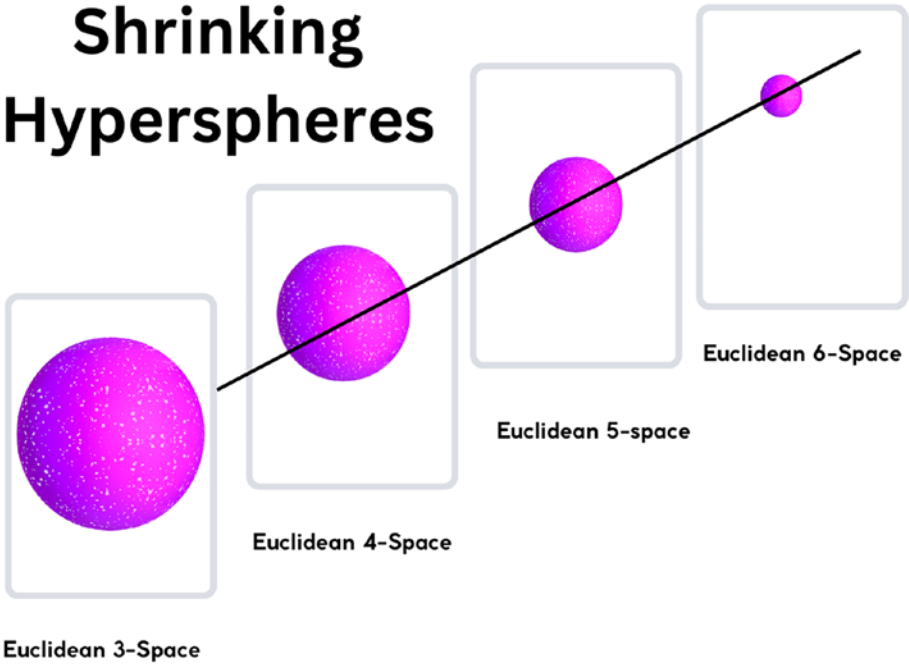

Figure 2-5. *A shrinking sphere illustrating the unintuitive nature of higher dimensions*

In data science in the real world, we have at minimum hundreds of features. It is not uncommon to have 1000 or more features, and so naturally we need a way to try to reduce the number of features. Mathematically speaking, this means we want a way to embed our data that lives in a high dimensional space to a lower dimensional space while preserving information.

Going back to our favorite example, logistic regression, we can illustrate another important mathematical tool to handle high dimensionality, regularization.

Regularization is extremely important when applying logistic regression because without it, the asymptotic nature of the logistic curve at +infinity and -infinity (remember the sigmoid?) would translate into zero loss in high dimensions. Consequently, we need strategies to dampen

the model complexity. The most common way is L2 regularization which means we'll give a higher penalty to model parameters that are nonzero. We can also use an L1 norm (a different way of measuring distance in high dimensional spaces). The penalty is defined as minus the square of the L2 norm multiplied by a positive complexity parameter lambda. Lambda controls the amount of shrinkage toward zero.

Models that use L1 regularization are called Lasso regression, and models that use L2 are called Ridge regression. If you would like to gain a deeper understanding of the types of norms that can exist and higher dimensional spaces, in the next section, you will have the opportunity to learn more about mathematical statistics in a hands-on lab.

Lab: Mathematical Statistics

Before proceeding to the next section, you can complete the optional lab on mathematical statistics. This will give you hands-on experience with probability distributions by looking at an important and fundamental tool in mathematical statistics: characteristic functions.

You'll program a characteristic function from scratch. Characteristic functions have many interesting properties including completely characterizing a probability distribution and are even used in the most basic proofs of the central limit theorem. The steps are as follows:

Step 1. Open the notebook MLOps_Lifecycle_Toolkit_Mathematical_Statistics_Lab.ipynb (available at github.com/apress/mlops-lifecycle-toolkit).

Step 2. Import the math, randon, and numpy packages by running cell #2.

Step 3. Create a function for computing the characteristic function of a random normal with unit standard deviation by running cell #3.

Step 4. Run the remaining cells, to set up a coin toss experiment and recover the probability of a fair coin from the characteristic function. Was the coin fair?

Although this lab is optional because it requires some advanced math, it's recommended since it covers some deep mathematical territory from probability distributions, Fourier transforms, complex numbers, and more.

Programming Nondeterministic Systems

In order to build real-world systems, we need to understand the types of data structures (arrays, lists, tensors, dataframes) and programming primitives (variables, loops, control flow, functions) that you'll likely encounter to know what the programming is doing and to be able to read other data scientists code.

Knowledge of data structures, algorithms, and packages can be applied regardless of language. If you use a package, even an R package, you should read the source code and understand what it's doing. The danger of not understanding what the statistical black box means the result of an analysis that uses your code could come out inaccurate or, worse, introduce non-determinism into your program.

Sources of non-determinism in ML systems

- *Noisy data sets*
- *Poor random initialization of model parameters*
- *Black box stochastic operations*
- *Random shuffling, splits, or data augmentation*

Programming and Computational Concepts

Let's look at some basic programming concepts.

Loops

Loops are a mechanism to repeat a block of code. Why use loops? One reason is you may have a block of code you want to repeat and, without a loop, you would have to copy paste the code, creating redundant code that is hard to read and reason about.

Another reason we use loops is for traversing a data structure such as a list or dataframe. A list or array has many items and a dataframe has many rows, in a well-defined order, and it is a natural way to process each element one by one; whether that element be a row or a list depends on the data structure.

Loops can be complex, and there's a programming adage that goes you should never modify a variable in a loop.

One reason loops are important in data science is twofold:

1) Many tensor operations naturally unfold into loops (think dot product or tensor operations).

2) By counting the number of nested loops, you can get an idea on the asymptotic behavior (written in Big-O notation) of your algorithm; in general, nested loops should be avoided if possible being replaced by tensor operations.

The last technique is actually an optimization tool called **vectorization**. Often, vectorized code can take advantage of low level instructions like single instruction, multiple data, or SIMD instructions. In fact, most GPUs use a SIMD architecture, and libraries like JAX can take this idea to the next level if you need to run NumPy code on a CPU, GPU, or even a TPU for high performance machine learning.

Variables, Statements, and Mathematica Expressions

What is the difference between a statement and an expression?

A statement does something that assigns a value to a variable. An example in Python is x = 1.

This simple statement assigns the value 1 to a variable x. The variable, x, in this case points to a memory location used to store information.

An expression on the other hand needs to be evaluated by the interpreter (or compiler in a compiled language like C++ or Haskell) and returns a value. Expressions can be building blocks of statements or complex mathematical expressions. An example of an expression (but not a statement) is the following:

```
(1 + 2 + x)
```

We can also have Boolean expressions which we'll look at next and are very important for making decisions.

Control Flow and Boolean Expressions

Control flow refers to the order in which individual statements, commands, instructions, statements, or function calls are executed. Changing the order of statements or function calls in a program can change the program entirely. In imperative languages (e.g., Python can be coded in an imperative style), control flow is handled explicitly by control flow statements such as if statements that control branching. Usually at each branch, a choice is made and the program follows one path depending on a condition. These conditions are called Boolean expressions.

Boolean expressions involve logical operations such as AND, OR, NOT, and XOR. These Boolean expressions can be combined in complex ways using parentheses and as mentioned are used in control flow statements in your program to make complex decisions.

For example, let's suppose you have a computer program with variables that store true and false values. You have one variable that stores the percent missing and a second variable that stores the number of rows in your data, and you want to exclude rows that have over 25% missing values when your data is more than 1000 rows. You can form a Boolean expression as follows:

```
If (percent_missing > 25) AND (num_rows  >  1000):
        // drop rows
```

Of course, in a library like Pandas, there are functions like dropna for dataframes that do this sort of low level logic for you, but you can read the source code to understand exactly what is happening under the hood for the functions you care about.

Tensor Operations and Einsums

A tensor is, simply put, a generalization of vectors to higher dimensions. There is some confusion on the use of the term since there are also tensors in physics, but in machine learning, they're basically a bucket for your data. Many libraries including NumPy, TensorFlow, and PyTorch have ways of defining and processing tensors, and if you've done any deep learning you're likely very familiar with tensors, but a cool tool I want to add to your toolkit is Einsums.

Einsums are essentially shorthand for working with tensors, and if you need to quickly translate complex mathematical equations (e.g., ones that occur in data science or machine learning papers), you can often rewrite them in Einsum notation in very succinct, elegant ways and then execute

them immediately in a library like PyTorch. For example, the following Einsum equation codifies matrix multiplication, and we can implement it in PyTorch in Listing 2-3:

Listing 2-3. An example of Einsum notation

```
a = torch.arange(900).reshape(30, 30)
b = torch.arange(900).reshape(30, 30)
torch.einsum('ik,kj->ij', [a, b])
```

Okay, we've covered quite a bit. We talked about variables, loops, and control flow and ended with tensors, a kind of bucket for high dimensional data. However, there are many more "buckets" for your data that are useful in data science. These are called data structures, the subject of computer science. We'll cover a few data structures in the next section.

Data Structures for Data Science

This section is about data structures. While computer science has many data structures, data scientists should be familiar with a few core data structures like sets, arrays, and lists. We will start by introducing sets, which might be the simplest data structure to understand if you come from a math background.

Sets

Sets are collections of elements. A set can contain elements, and you can use sets for a variety of purposes in data science for de-duplication of your data to checking set membership (that is to say, the set data structure comes with an IN operator).

It is important to note that a set has no order (actually there is the well-ordering principle that says exactly the opposite, but in Python, for instance, and other languages, sets have no order). If we want to impose an order when storing elements, we should use a linear data structure like an array or a list, which we'll cover next.

Arrays and Lists

The most fundamental distinction between an array and a list is that a list is a heterogeneous data structure, and this mean it can store a mix of data types, for example strings, floats, Booleans, or even more complex user defined types.

An array on the other hand is homogenous; it only is designed to store one type of value.

In Python, lists are a primitive data type and part of the core language. The ability to use list comprehensions instead of loops for mathematical constructs is very useful in data science. However, for efficient processing of data, we can use a library like NumPy which has a concept of arrays. This is known as a trade-off, and in this case, the trade-off exists between efficiency and convenience.

Part of being a good technical decision-maker is understanding these types of technical trade-offs and the consequences on your own project. For example, if you decide to profile your code and find you're running into memory errors, you might consider changing to a more efficient data structure like a NumPy array, maybe even with a 32 bit float if you don't need the extra precision of a 64 bit floating point number.

There are many different types of data structures and we'll provide resources for learning about more advanced types (one of the core subjects of computer science), but for now, we'll take a look at a more complex type that you should be aware of such as hash maps, trees, and graphs.

Hash Maps

Hash maps are an associative data structure; they allow the programmer to associate a key with a value.

They provide very fast lookup by keys, allowing you to retrieve a value corresponding to a key in O(1) time by using dynamically sized arrays under the hood and allow you to retrieve a value you've associated with your key.

If you didn't have this kind of associative data structure, you'd have to, for instance, store your elements as an array of tuples and would need to write code to search for each key you wanted to locate in the array. This would not be very efficient, so when we want to associate one piece of information with another and only care about being able to retrieve the value we've mapped to a particular key, we should consider hash maps. The point is, having a command of data structures can simplify your code drastically and make it more efficient.

In Python, a hash map is called a dictionary. One point to keep in mind when using hash maps is that the keys should be hashable, meaning a string is OK for a key but a mutable data type like a list that can be changed is not allowed.

Trees and Graphs

A graph is a mathematical data structure consisting of nodes and edges. The nodes are also called vertices. The difference between a tree and a graph is that a tree has a root node. In a graph there is no root node that is unique but both structures can be used for representing many different types of problems. Graph neural networks and graph databases are huge topics today in machine learning and MLOps, and part of the reason is that a graph, like a set, is a very general mathematical way of representing relationships between concepts that can be easily stored on a computer and processed.

You should be aware of a couple kinds of trees and graphs in particular binary trees and DAGs.

Binary Tree

A binary tree is a tree (it has a root node), and each node including the root has either 2 (hence binary) children or 0 children (in this case, we call it a leaf node). A picture of a binary tree is shown in Figure 2-6.

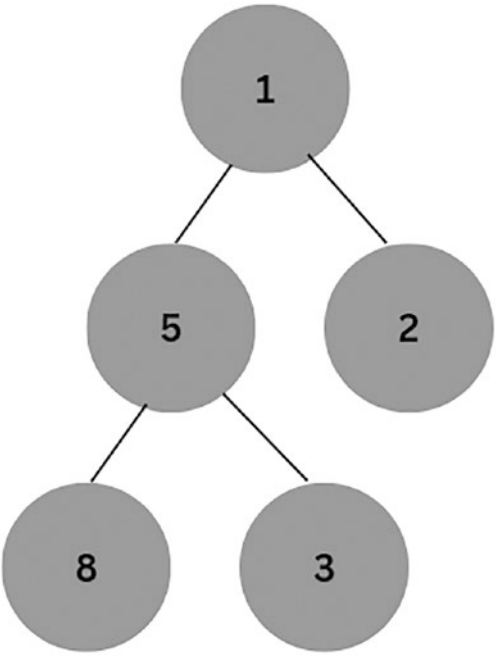

Figure 2-6. *A binary tree*

Binary trees can be complete or perfect or have additional structure that makes them useful for searching such as binary search trees.

DAGs

A graph is a generalization of a tree. A graph however can have cycles, meaning if you were to visit every node and follow its neighbor, you may find yourself in an infinite loop. An important type of graph with no cycles is called an acyclic graph and is often used in MLOps tools like Airflow to represent data flow. Directed acyclic graphs are called "DAGs" and have a variety of uses in MLOps (e.g., the popular Airflow library uses DAGs for creating pipelines).

SQL Basics

We've covered programming languages like Python, but you also need to know how to manipulate data in your programs. SQL is actually based on relational algebra and the set data structure we covered previously (the foundations were written by Edger F. Codd). SQL consists of queries and the queries can be broken down into statements. A SQL statement consists of the following clauses executed in the following order:

- FROM
- JOINS on other tables
- WHERE clause for filtering data
- GROUP BY for aggregating by multiple columns
- HAVING for filtering after aggregation
- SELECT for selecting columns or fields you want to use in your data set
- ORDER BY for sorting data by one or more columns (this can cause performance issues and should only be used sparingly)

A common table expression or CTE is a very useful construct when operationalizing data science code. The reason it is so powerful is that a CTE allows you to think algorithmically, by breaking down your SQL query into a series of steps. Each step can depend on previous steps and is materialized as a kind of "virtual table." A simple example of a CTE is given in the following; this CTE first creates a base table called Sensor_CTE and then selects from it in Listing 2-4.

Listing 2-4. An example of a common table expression or CTE

```
-- An example of a CTE
WITH Sensor_CTE (SalesPersonID, SalesOrderID, SalesYear)
AS
-- Define the CTE query.
(
    SELECT ID as Component, MAX(Pressure) as Pressure,
    AVG(Temperature) as Temperature
    FROM Sensor.Readings
    WHERE ID IS NOT NULL
    GROUP BY ID
)
-- Define the outer query referencing the CTE name.
SELECT Component, Temperature
FROM Sensor_CTE;
```

Understanding how joins and common table expressions (CTEs) work is typically what separates beginners from advanced SQL users. Most data science code requires multiple passes on data sets, and CTEs are a natural way to write more complex SQL code that requires multiple steps to process data.

Algorithmic Thinking for Data Science

An algorithm is essentially a set of rules or instructions for performing calculations that occur in a certain sequence. You can think of it like a recipe. Unlike recipes though, algorithms will usually involve data structures for storing data, and the heart of the algorithm will be manipulating these data structures to solve a problem. Unfortunately, you need to learn algorithmic thinking, and by doing so, we've created a lab for you. In the lab, you're going to start with data structures we've learned to build some basic mathematical primitives like sigmoid function and logistic curve and combine these abstractions to build your own logistic regression model. Refer to the Jupyter notebook labs for this chapter entitled "Building a Logistic Regression Model from Scratch," and complete the lab before continuing to the next section.

Core Technical Decision-Making: Choosing the Right Tool

Beyond this section, we're going to assume you've completed the labs and have a basic grasp on programming fundamentals. Before covering specific packages and frameworks for translating experiments and thoughts into executable code, I want to discuss technical decision-making briefly and how we should think about choosing the right framework for our problem.

The most important criterion in the real world is considering what tools and frameworks are already being used by your organization, colleagues, and the community behind the framework. Although you might be tempted to use a package from a language like Julia or Haskell, you should carefully consider whether or not you'll have to translate your problem into another language at some point in the future if either the package is no longer supported or because nobody in your organization has the skill set required.

Translating Thoughts into Executable Code

You might want to choose one of the following packages and dive deeper into some frameworks that are used in the real world to build machine learning models. In later chapters, we'll walk you through how to create your own packages. The important thing here is understanding that these tools we depend on in data science like Pandas or Numpy or PyTorch are just packages someone (or a team of people) have written and created. You too can learn to create your own packages, but first we need to understand why we use packages and how it makes our lives as both data scientists and MLOps engineers easier.

Understanding Libraries and Packages

What is the point of a software package? Why not use a notebook? Packages allow us to bundle code together and give it a name, import it, and reference objects inside the package so we can reuse them without having to rewrite those objects. Packages can also be versioned (see semantic versioning[4]).

For example, you may have heard of RStudio package manager for R or pip for Python. Before experimenting with any of the packages listed in the following, you should understand the package manager in your language of choice so you can install the package. We also recommend environments to isolate dependencies. We'll cover the gritty details of package managers and environments in Chapter 3, but for now here is a broad overview of some of the most interesting packages you might come across as an MLOps engineer.

[4] Semantic versioning 2.0.0 can be found at `https://semver.org/`.

PyMc3 Package

An active area of research is in probabilistic programming. The PyMc3 library contains various primitives for creating and working with random variables and models. You can perform MCMC (Markov chain Monte Carlo) sampling and directly translate statistical models into code.

Something to keep in mind is at the current time, these algorithms may not be very scalable, so you'll usually only see Bayesian optimization applied to the hyperparameter search part of a machine learning lifecycle using libraries like HyperOpt; however, we mention probabilistic programming as Bayesian statistics is slowly becoming a part of mainstream data science.

Numpy and Pandas

Numpy and Pandas are the bread and butter of most data science workflows. We could write an entire chapter covering just these libraries, but we'll mention for the uninitiated that Pandas is a data wrangling library. It provides a data structure called a DataFrame for processing structured data and various methods for reading csv files and manipulating dataframes. NumPy has the concept of ndarrays and allows you to process numerical data very fast without having to know much about C++ or low level hardware.

R Packages

R uses a package system called CRAN which makes available R binaries. Unlike Python, CRAN packages typically have higher dependency on other packages and tend to be focused on specific areas of statistical computing and data visualization.

The reason data scientists still use R is many packages written by researchers and statisticians are written in R. However, you should be aware of the following interoperability and scalability issues with R:

- R is not as widely supported; for example, the machine learning SDK uses R, but there is a lag between when features are released in Python and when they become available in R.
- Writing clear, concise and easy to read code in R requires considerable skill and even then there are leaky abstractions which make code difficult to maintain such as
- R is not scalable and has memory limitations. For scalable R, we recommend Databricks using SparkR.

A lot of R packages revolve around the TidyVerse. You should be familiar with the following basic R packages:

Deplyr: Deplyr is a package that is similar to Pandas in Python and is used for data wrangling. The package provides primitives such as filter and melt.

Shiny: The R ecosystem's answer to dashboarding in data science, Shiny is a package for authoring dashboards in R and fulfills the same need as Dash in Python. The advantage of ShinyR is you can build web apps without having to know how web development works. The web apps can be interactive, and you can interact with different panels of the dashboard and have multiple data sources to visualize data sets. We don't recommend Shiny as it can be hard to deploy to a web server securely.

SAS: SAS is a language of statistical programming. SAS is a procedural language. SAS requires a SAS license and is common in healthcare and finance industry where exact statistical procedures need to be executed.

MATLAB/OCTAVE: MATLAB and the open source version Octave are libraries for linear algebra. If you are prototyping a machine learning algorithm whose primitives can be expressed using matrix operations

(which is a lot of machine learning), then you might consider using one of these languages. MATLAB is also particularly popular in engineering disciplines for simulations and is used in numerical computing.

PySpark: Spark is a framework for distributed computing and has a tool called PySpark that allows you to write code similar to Pandas using dataframes but in a scalable way. You can translate between Pandas and Pyspark using the latest Pandas API for spark (replacement for Koalas) and process gigabytes or even terabytes of data without running into out of memory errors. Other alternatives are called "out of core" solutions and include Dask or Modin that utilize disk storage as an extension of core memory in order to handle memory-intensive workloads.

Important Frameworks for Deep Learning

There are many frameworks in Python for deep learning and working with tensors. PyTorch and TensorFlow 2.0 with Keras API are the most popular. Although we could implement our own routines in a package like the NumPy example to build your own 2D convolutional layer and use these functions to build a convolutional neural network, in reality, this would be too slow. We would have to implement our own gradient descent algorithm, auto differentiation, and GPU and hardware acceleration routines. Instead, we should choose PyTorch or TensorFlow.

TensorFlow

TensorFlow is an end to end machine learning framework for deep learning. TensorFlow is free and open sourced under Apache License 2.0 and supports a wide variety of platforms including MacOS, Windows, Linux, and even Android. TensorFlow 1.0 and TensorFlow 2.0 have significant differences in APIs, but both provide the tensor as a core abstraction allowing the programmer to build computational graphs to represent machine learning algorithms.

PyTorch

The advantage is PyTorch is class oriented, and if you have a strong Python background, you can write a lot of custom code in an object oriented style without having to be very familiar with how the APIs work like in TensorFlow. PyTorch for this reason is used in academic papers on machine learning and is a solid choice for prototyping machine learning solutions.

Theano

PyMC3 is written on top of Theano as well as some other interesting projects, but Theano is no longer supported so it is not recommended for ML development or MLOps.

Keras

Prior to the introduction of the Keras API, developers required specific knowledge of the API. Keras however is very beginner friendly, and some useful features of TensorFlow are GPU awareness (you do not need to change your code to use a GPU if one is available, as TensorFlow will detect if for you); the Keras API is very intuitive for beginners, and there is a large community around TensorFlow so bugs and CVEs (security vulnerabilities) are patched regularly. Post TensorFlow 2.0 release, you can also do dynamic execution graphs.

Further Resources in Computer Science Foundations

We've covered a lot of ground, discussed data structures and algorithmic thinking, and covered the basics of computer science required to work

with data such as graphs, dataframes, tables, and the basics of SQL. We've talked about R and Python, two common languages for data science, and some of their common packages.

However, it is important to stress this is only the minimum. It would not be possible to cover a complete course in computer science for data scientists in this chapter, and so the best we can do is provide some recommended reading so you can educate yourself on topics you're interested in or fill in gaps in your knowledge to become better programmers. We've curated the following list of books on computer science that we think would be most valuable for data scientists.

- Introduction to Algorithms by Rivest[5]
- Bayesian Methods for Hackers by Davidson Pilon[6]

In general, you can read a book on functional analysis (for infinite dimensions) or linear algebra (for finite dimensional spaces) provided in the following.

Further Reading in Mathematical Foundations

Although we covered some mathematical concepts in this chapter, it would not be possible to cover even the simplest areas like linear algebra in detail without further resources. Some areas you may be interested in pursuing on your own are Bayesian statistics[7] (understanding Bayes' rule, Bayesian

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

[6] Davidson-Pilon, C. (2015). *Bayesian Methods for Hackers: Probabilistic Programming and Bayesian Inference*. Addison-Wesley Professional.

[7] McElreath, R. (2015). *Statistical Rethinking: A Bayesian Course With Examples in R and Stan*. Chapman & Hall/CRC.

inference, and statistical thinking), statistical learning theory[8] (the rigorous foundations of the many learning algorithms we use in MLOps), and of course linear algebra[9] (in particular finite dimensional vector spaces are a good stepping stone to understand more advanced concepts).

Summary

In this chapter, we discussed the importance of understanding mathematical concepts and how MLOps systems can be viewed as stochastic systems that are governed by mathematical abstractions. By understanding these mathematical abstractions and having an understanding of data structures and algorithmic thinking, we can become better technical decision-makers. Some of the topics we covered in this chapter include the following:

- Programming Nondeterministic systems
- Data Structures for Data Science
- Algorithmic Thinking for Data Science
- Translating Thoughts into Executable Code
- Further Resources on Computer Science

In the next chapter, we will take a more pragmatic perspective and look at how we can use these abstractions as tools and software packages when developing stochastic systems in the real world.

[8] Hastie, T., Tibshirani, R., & Friedman, J. (2013). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Science & Business Media.

[9] Halmos, P. (1993). *Finite-Dimensional Vector Spaces*. Springer.