Chapter 8

Building a Machine Learning Pipeline

Audrey Altman
Digital Public Library of America

As a new machine learning (ML) practitioner, it is important to develop a mindful approach to the craft. By mindful, I mean possessing the ability to think clearly about each individual piece of the process, and understanding how each piece fits into the larger whole. In my experience, there are many good tutorials available that will help you work with an individual tool, deploy a specific algorithm, or complete a single task. It is more difficult to find guidelines for building a holistic system that supports the entire ML workflow. My aim is to help you build just such a system, so that you are free to focus on inquiry and discovery rather than struggling with infrastructure and process. I write this as a software developer who has, at one time or another, been on the wrong end of all the recommendations presented here, and hopes to save you from similar headaches. Many of the examples and design choices are drawn from my experiences at the Digital Public Library of America, where I have worked alongside a very talented team of developers. This is by no means an exhaustive text, but rather a bit of pragmatic advice and a jumping-off point for further research, designed to give you a clearer idea of which questions to ask throughout your practice.

This article reviews the basic machine learning workflow, discussing design considerations along the way. It offers recommendations for data storage, guidelines on selecting and working with ML algorithms, and questions to guide tool selection. Finally, it describes some challenges with scaling up. My hope is that the insight presented here, combined with your good judgement, will empower you to get started with the actual practice of designing and executing a machine learning project.

Algorithm selection

As you begin ingesting and preparing data, you'll want to explore possible machine learning algorithms to perform on your dataset. Choose an algorithm that fits your research question and data. If you're not sure which algorithm to choose and not constrained by time, experiment with several different options and see which one yields the best results. Start by determining what general type of learning algorithm you need, and proceed from there to research and select one that specifically addresses your research question.

In *supervised learning*, you train a model to predict an output condition based on given input conditions; for example, predicting whether or not a patient has some disease based on their symptoms, or the topic of a news article based on keywords in the text. In order for supervised learning to work, you need *labeled* training data, meaning data in which the outcome is already known. Examples include records of symptoms in patients who were known to have the disease (or not), or news articles that have already been assigned topics.

Classification and *regression* are both types of supervised learning. In a classification problem, you are predicting a discrete number of possible outcomes. For example, "based on what I know about this book, will it make the *New York Times* Best Seller list?" is a classification problem because there are two discrete outcomes: yes or no. Classification algorithms include naive Bayes, decision trees, and k-nearest neighbor. Regression problems try to predict an outcome from a continuum of possibilities, i.e., "based on what I know about this book, what will its retail price be?" Regression algorithms include linear regression and regression trees.

In *unsupervised learning*, the ML algorithm discovers a new pattern. The training data is unlabeled, meaning there is no indication of how the data should be organized at the outset. A common example is clustering, in which the algorithm groups items together based on features it finds mathematically significant. Perhaps you have a collection of news articles (with no existing topic labels), and you want to discover common themes or topics that appear throughout the collection. The algorithm will not tell you what the themes or topics are, but will show which articles group together. It is then up to the researcher to work out the common thread.

In addition to serving your research question, your algorithm should also be a good fit for your data. Specific considerations will vary for each dataset and algorithm, so make sure you know the strengths and weaknesses of your algorithm and how they relate to the unique qualities of your dataset. For example, algorithms differ in their abilities to handle datasets with a very large number of features, handle datasets with high variance, efficiently process very large datasets, and glean meaningful intelligence from very small datasets. Is it important that your algorithm be easy to explain? Some algorithms, such as neural nets, function as black boxes, and it is difficult to decipher how they arrive at their decisions. Other algorithms, such as decision trees, are easy to understand. Can you prepare your data for the algorithm with a reasonable amount of preprocessing? Can you find examples of success (or failure) from people using similar datasets with the same algorithm? Asking these sorts of questions will help you to choose an algorithm that works well for your data, and will also inform how you prepare your data for optimal use.

Finally, consider whether or not you are constrained by time, hardware, or available toolsets. Different algorithms require different amounts of time and memory to train and/or execute. Different ML tools offer implementations of different algorithms.

The machine learning pipeline

The metaphor of a pipeline is often used for a machine learning workflow. This metaphor captures the idea of data channeled through a series of sequential transformations. However, it is important to note that each stage in the process will need to be repeated and honed throughout the course of your project. Therefore, don't think of yourself as building a single intelligent model, such as a decision tree or clustering algorithm. Instead, build a pipeline with pieces that can be swapped in and out as needed. Data flows through the pipeline and outputs a version of a decision tree, clustering algorithm, or other intelligent model. Throughout your process, you will tweak your pipeline, making many intelligent models. Eventually you will select the best model for your use case. To use another metaphor, don't build a car, build an assembly line for making cars.

While the final output of a machine learning workflow is some sort of intelligent model, there are many factors that make repetition and iteration necessary. ML processes often involve subjective decisions, such as which data points to ignore, or which configurations to select for your algorithm. You will want to test different possibilities to see what works best. As you learn more about your dataset throughout the course of the project, you will go back and tweak parts of your process. You may discover biases in your data or algorithms that need to be addressed. If you are working collaboratively, you will be incorporating asynchronous feedback from members of your team. At some point, you may need to introduce new or revised data, or try a new tool or algorithm. It is also prudent to expect and plan for errors. Human errors are inevitable, and hardware errors, such as network timeouts or memory overloads, are common. For all of these reasons, you will be well-served by a pipeline composed of modular, repeatable steps, each with discrete and stable output.

A modular pipeline supports a batch processing workflow, in which whole datasets undergo a series of transformations. During each step of the process, a large amount of data (possibly the entire dataset) is transformed all at once and then incrementally stored. This can be contrasted with a real-time workflow, in which individual records are transformed instantaneously (e.g. a librarian updates a single record in library catalog); or a streaming workflow, in which a continuous flow of data is pushed through an entire pipeline, often without incremental storage along the way (e.g. performing analysis on a continuous stream of new tweets). Batch processing is common in the research and development phase of an ML project, and may also be a good choice for a production system.

When designing any step in the batch processing pipeline, assume that at some point you will need to repeat it either exactly as is, or with modifications. Documenting your process lets you compare the outputs of different variations and communicate the ways in which your choices impact the final results. If you're writing code, version control software can help. If you're doing more manual data manipulations, such as editing data in spreadsheets, you will need an intentional system of documenting exactly which transformations you are applying to your data. It is generally preferable to automate processes wherever possible so that you can repeat them with ease and consistency.

A concrete example from my own experience demonstrates the importance of a pipeline that supports repetition. In my first ever ML project, I worked with a set of XML library data converted to CSV. I did most of my data cleanup by hand using spreadsheet software, and was not careful about preserving the formulas for each step of the process; instead, I deleted and wrote over many important intermediate computations, saving only the final results. This whole pro-

cess took me countless hours, and when an updated dataset became available, there was no way to reproduce my painstaking cleanup process. I was stuck with outdated data, and my final output was doomed to grow more and more irrelevant as time wore on. Since then, I have always written repeatable scripts for all my data cleanup tasks.

Each decision you make will have an impact on the final results, so it is important to keep clear documentation and to verify your assumptions and hypotheses wherever possible. Sometimes there will be explicit tests to perform; at other times, you may just need to look at data—make a quick visualization, perform a simple calculation, or glance through a sample of records. Be cognizant of the potential to introduce error or bias. For example, you could remove a field that you don't think is important, but that would, in fact, have a meaningful impact on the final result. All of these precautions will strengthen confidence in your final outcomes and make them intelligible to your collaborators and other audiences.

The pipeline for a machine learning project generally comprises five stages: data acquisition, data preparation, model training and testing, evaluation and analysis, and application of results.

Data acquisition

The first step is to acquire the data that you will be using for your machine learning project. You may need to combine data from several different sources. There are many ways to acquire data, including downloading files, querying a database or API, or scraping web pages. Depending on the size of the source data and how it is made available, this can be a quick and simple step or the most challenging bottleneck in your pipeline. However you get your initial data, it is generally a good idea to save a copy in the rawest possible form and treat that copy as immutable, at least during the initial phase of testing different algorithms or configurations. Having a raw, immutable copy of your initial dataset (or datasets) ensures that you can always go back to the beginning of your ML process and start over with exactly the same input. It will also save you from the possibility that the source data will change from beneath you, thereby compromising your ability to compare the outputs of different operations (for more on this, see the section on data storage). If possible, it's often worthwhile to learn about how the original data was created, especially if you are getting data from multiple sources that differ in subtle ways.

Data preparation

Data preparation involves cleaning data and transforming it into an appropriate format for subsequent machine learning tasks. This is often the part of the process that requires the most work, and you should expect to iterate over your data preparations many times, even after you've started training and testing models.

The first step of data preparation is to parse your acquired data and transform it into a common, usable schema. Acquired data often comes in file formats that are good for data sharing, such as XML, JSON, or CSV. You can parse these files into whatever schema makes sense to manage the various transformations you want to perform, but it can help to have a sense of where you are headed. Your eventual choice of data format will likely be dictated by your ML algorithms; likely candidates include multidimensional arrays, tensors, matrices, and DataFrames. Look ahead to specific functions in the specific libraries you plan to use, and see what type of input data is required. You don't have to use these same formats during your data preparations, though it can simplify the process.

Data cleanup and transformation is an art. Data is messy, and the messier the data, the harder it is to analyze and uncover underlying patterns. Yet, we are only human, and perfect data is far beyond our reach. To strike a workable balance, focus on those cleanup tasks that you know (or strongly suspect) will have a significant impact on the final product. Cleanup and transformation operations include removing punctuation or stopwords from textual data, standardizing date and number formats, replacing missing or dummy values with a meaningful default, and excluding data that is known to be erroneous or atypical. You will select relevant data points, and you may need to represent them in a new way: a birth date becomes age range; a place name becomes geo-coordinates; a text document becomes a word density vector. There are many possible normalizations to perform, depending on your dataset and which algorithm(s) you plan to use. It's not a bad idea to ensure that there's a genuinely unique identifier for each record (even if you don't see an immediate need for one). This is also a good time to reflect on any biases that might be inherent in your data, and whether or not you can adjust for them; even if you cannot, understanding how they might impact the ML process will help you conduct a more nuanced analysis and frame your final results. At the very least, you can record biases in the documentation so that future researchers will be aware of them and react accordingly. As you become more familiar with the data, you will likely hone your cleanup process and iterate through the steps multiple times.

The more you can learn about the data, the better your preparations will be. During the data preparation phase, practitioners often make use of visualizations and query frameworks to picture their data holistically, identify patterns, and find errors or outliers. Some ML tools support these features out-of-the-box, or are intentionally interoperable with external query and visualization tools. For a lightweight tool, consider spreadsheet or notebook software. Depending on your use case, it may be worthwhile to put your data into a temporary database or search index so that you can make use of a more sophisticated query interface.

Model testing and training

During the testing and training phase, you will build multiple models and determine which one gives you the best results. One of the main ways you will tune your model is by trying multiple combinations of *hyperparameters*. A hyperparameter is a value that you set before you run the learning process, which impacts how the learning process works. Hyperparameters control things like the number of learning cycles an algorithm will iterate through, the number of layers in a neural net, the characteristics of a cluster, or the number of decision trees in a forest. Often, you will also want to circle back to your data preparation steps to try different configurations, apply new enhancements, or address new problems and particularities that you've uncovered. The process is deceptively simple: try out different configurations until you get a good result. The challenge comes when you try to define what constitutes a good (or good-enough) result.

Measuring the quality of a machine learning model takes finesse. Start by asking: What would you expect to see if the model learned perfectly? Equally important, what would you expect to see if the model didn't learn anything at all? You can often utilize randomness as a stand-in for no learning, e.g. "if a result was selected at random, the probability of the desired outcome would be X". These two questions will help you to set benchmarks at both extremes of the realm of possible outcomes. Perfection is illusive, and the return on investment dwindles after a while, so be prepared to stop training once you've arrived at an acceptably good model.

In a supervised learning problem the dataset is split into training and testing datasets. The algorithm uses the training data to "learn" a set of rules that it can subsequently apply to new,

unseen data to predict the outcome. The testing dataset (also called a validation dataset) is used to test how well the model performs. Often, a third dataset is held out as well, reserved for final testing after the model has been trained. This third dataset provides an additional bulwark against bias and overfitting. Results are typically evaluated based on some statistical measurement that is directly relevant to your research question. In a classification problem, you might optimize for recall or precision. In a regression problem, you can use formulas such as the root-mean square deviation to measure how well the regression line matches the actual data points. How you choose to optimize your model will depend on your specific context and priorities.

Testing an unsupervised model is not as straightforward, since there is no preconceived notion of correct and incorrect categorization. You can sometimes rely on a known pattern in the underlying dataset that you would reasonably expect to be reflected in a successful model. There may also be characteristics of the final model that indicate success. For example, if you are working with a clustering algorithm, models with dense, well-defined clusters are probably better than sparse clusters with vague boundaries. In unsupervised learning, you may want to hold back some portion of your data to perform an independent validation of your results, or you may use the entire dataset to build the model—it depends on what type of testing you want to perform.

Application of results

As the final step of your workflow, you will use your intelligent model to perform some task. Perhaps you will use it for scholarly analysis of a dataset, or perhaps you will integrate it into a software product. If it is the former, consider how to export any final data and preserve the artifacts of your project. If it is the latter, consider how the model, its outputs, and its continued maintenance will fit into existing systems and workflows. Planning for interoperability may influence decisions from tool selection to data formats and storage.

Immutable data storage

Immutable data storage can benefit the batch-processing ML pipeline, especially during the initial research and development phase. This type of data storage supports iteration and allows you to compare the results of many different experiments. Treating data as immutable means that after each significant change or set of changes to your data, you save a new snapshot of the dataset that is never edited or changed. It also allows you to be flexible and adaptive with your data model. Immutable data storage has become a popular choice for data-intensive or "big data" applications as a way to easily assemble large quantities of data, often from multiple sources, without having to spend time upfront crafting a strict data model. You may have heard the term "data lake" to refer to such large, unstructured collections of data. This can be contrasted with a "data warehouse", which usually indicates a highly structured, centralized repository such as a relational database.

To demonstrate how immutable supports iteration and experimentation, consider the following scenario: You start with an input file `my_data.csv`, and then perform some cleanup operation over the data, such as converting all measurements in miles to kilometers, rounded to the nearest whole number. If you were treating your data as mutable, you might overwrite the original contents of `my_data.csv` with the transformed values. The problem with this approach comes if you want to test some alteration of your cleanup operation. Say, for example, you wanted to round all your conversions to the nearest tenth instead. Since you no longer have your original data, you would have to start the entire ML process from the top. If you instead

treated your data as immutable, you would keep `my_data.csv` in its original state, and save the output of your cleanup operation in a new file, say `my_clean_data.csv`. That way, you could return to `my_data.csv` as many times as you wished, try different operations on this data, and easily compare the results of these operations knowing the source data was exactly the same for each one. Think of each immutable dataset as a place in your process that you can safely reset to anytime you want to try something new or correct for some bias or failure.

To illustrate the benefits of a flexible data model, consider a mutable data store, such as a relational database. Before you put any data into the database, you would first need to design a system of tables with set fields and datatypes, and the relationships between those tables. This can feel like putting the cart before the horse, especially if you are starting with a dataset with which you are not yet intimately familiar, and you want the ability to experiment with different algorithms, all of which might require slightly different transformations on the original dataset. Revisiting the example in the previous paragraph, you might initially have defined your distance datatype as an integer (when you were rounding to the nearest whole number), and would later have to change it to a floating point number (when you were rounding to the nearest tenth). Making this change would mean altering the database schema and migrating all of the existing data to the new type, which is a nontrivial task—especially if you later decide to revert back to the original type. By contrast, if you were working with immutable CSV files, it would be much easier to write out two files, one with each data type, and keep whichever one ultimately proved most effective.

Throughout your ML process, you can create several incremental datasets that are essentially read-only. There's no one correct data storage format, but ideally you would use something simple and space-efficient with the capacity to interoperate with different tools, such as flat files (plain text files without extraneous markup, such as TXT, CSV, or Parquet). Even if your data is ultimately destined for a different kind of datastore, such as a relational database or triplestore, consider using simple, immutable storage as an intermediary to facilitate iteration and experimentation. If you're concerned about overwhelming your local drive, cloud storage is a good option, especially if you can read and write directly from your programs or software services.

One final benefit of immutable storage relates to scale. Batch processing workflows and immutable data storage work well with distributed data processing frameworks, such as MapReduce and Spark. Therefore, if you need to scale your ML project using distributed processing, the integration will be more seamless (for more, see the section on scaling up).

Organizing Immutable Data

Organizing immutable data stores can be a challenge, especially with multiple users. A little planning can save you from losing track of your experiments and results. A well-ordered directory structure, informative and consistent file names, liberal use of timestamps, and disciplined note-taking are simple but effective strategies. For example, say you were acquiring MARCXML records from an API feed, parsing out subject terms, and building a clustering algorithm around these terms. Let us explore one possible way that you could organize your data outputs through each step of the machine learning pipeline.

To enforce a naming convention, create a helper method that generates the output path for each run of a particular data process. This output path includes the date and timestamp of the run—that way you won't have to think about naming each individual file, and can avoid the phenomenon of a mess of files called `my_clean_data.csv`, `my_cleaner_data.csv`,

`my_final_cleanest_data.csv`, etc. Your file path for the acquired data might be in the format:

```
myProject/acquisitions/marc_YYYYMMDD_HHMMSS.xml
```

In this case, "YYMMDD" represents the date and "HHMMSS" represents the timestamp. Your file path for prepared and cleaned data might be:

```
myProject/clean_datasets/subjects_YYYYMMDD_HHMMSS.csv
```

Finally, each clustering model you build could be saved using the file path pattern:

```
myProject/models/cluster_YYYYMMDD_HHMMSS
```

Following this general pattern, you can organize all of the outputs for your entire project. Using date and timestamps in the file name also enables easy sorting and retrieval of the most recent output.

For each data output, you will want to maintain a record of the exact input, any special attributes of the process (e.g. "this time I rounded decimals to the nearest hundredth"), and metrics that will help you determine success or failure of the process. If you can generate this information automatically for each process, all the better for ensuring an accurate record. One strategy is to include a second helper method in your program that will generate and write out a companion file to each data output. The companion file contains information that will help evaluate results, detect errors, perform optimizations, and differentiate between any two data outputs.

In the example project, you could accompany the acquisition output with a text file detailing the exact API call used to fetch the data, the number of records acquired, and the runtime for the process. Keeping companion files as close as possible to their outputs helps prevent accidental separation, so save it to:

```
myProject/acquisition/marc_YYYYMMDD_HHMMSS.txt
```

In this case, the date and timestamp should exactly match that of its companion XML file. When running processes that test and train models, you can include information in your companion file about hyperparameters and whatever metrics you are using to evaluate the quality of the model. In our example, the companion file to each cluster model may contain the file path for the cleaned input data, the number of clusters, and a measure of cluster variance.

Working with machine learning algorithms

New technologies and software advances make machine learning more accessible to "lay" users, by which I mean those of us without advanced degrees in mathematics or data science. Yet, the algorithms are complex, and you need at least an intuitive understanding of how they work if you hope to implement them correctly. I use the following three questions as a guide for understanding an algorithm. Keep in mind that any one project will likely make use of several complex algorithms along the way. These questions help ensure that I have the information I truly need, and avoid getting bogged down with details best left to mathematicians.

- *What do the inputs and outputs of the algorithm mean?* There are two parts to answering this question. First is the data structure, e.g. "this is a vector with 300 integers." Second

is knowing what this data describes, e.g. "each vector represents a document, and each integer specifies the number of times a particular word appears in that document." You also need to be aware of specific implementation details—perhaps the input needs to be normalized in some way, perhaps the output has been smoothed (a technique that compensates for noisy data or outliers). This may seem straightforward, but it can be a lot to keep track of once you've gone through several layers of processing and abstraction.

- *What effect do different hyperparameters have on the algorithm?* Part of the machine learning process is tuning hyperparameters, or trying out multiple configurations until you get satisfying results. Part of the frustration is that you can't try every possible configuration, so you have to do some intelligent guesswork. Twiddling hyperparameters can feel enigmatic and unitutive, since it can be difficult to predict their impact on the final outcome. The better you understand hyperparameters and their roles in the ML process, the more likely you are to make reasonable guesses and adjustments—though you should always be prepared for a surprise.
- *Can you explain how this algorithm works to a lay person and why it's beneficial to the project?* There are two benefits to articulating a response to this question. First, it ensures that you really understand the algorithm yourself. And second, you will likely be called on to give this explanation to co-collaborators and other stakeholders. A good explanation will build excitement around the project, while a befuddling one could sow doubt or disinterest. It can be difficult to strike a balance between general summary and technical equations, since your stakeholders will likely include people with diverse backgrounds, so do your best and look for opportunities for people with different expertises to help refine your team's understanding of the algorithm.

Learning more about the underlying math can help you make better, more nuanced decisions about how to deploy the algorithm, and is fascinating in its own right—but in most cases I have found that the above three questions provide a solid foundation for machine learning research.

Tool selection

Tool selection is an important part of your process and should be approached thoughtfully. A good approach is to articulate and prioritize the needs of your team, and make selections that meet these needs. I've listed some possible questions for consideration below, many of which you will recognize as general concerns for any tool selection process.

- *What sorts of features and interfaces do they offer?* If you require a specific algorithm, the ability to make data visualizations, or query interfaces, you can find tools to meet these specific needs.
- *How well do tools interoperate with one another, or with other parts of your existing systems?* One of the advantages of a well-designed pipeline is that it will enable you to swap out software components if the need arises. For example, if your data is in a format that is interoperable with many systems, it frees you from being tied down to any specific tool.
- *How do the tools align with the skill sets and comfort levels of your team?* For example, consider what coding languages your collaborators know, and whether or not they have the

capacity to learn a new one. If you have someone who is already a wiz with a preferred spreadsheet program, see if you can export data into a compatible file format.

- *Are the tools stable, well-documented, and well-supported?* Machine learning is a fast-changing field, with new algorithms, services, and software features being developed all the time. Something new and exciting that hasn't yet been road-tested may not be worth the risk if there is a more dependable alternative. Furthermore, there tends to be more scholarship, documented use cases, and tutorials for older, more widely-adopted tools.
- *Are you concerned about speed and scale?* Don't get bogged down with these considerations if you're just trying to get a working pilot off the ground, but it can help to at least be aware of how problems are likely to manifest as your volume of data increases, or as you integrate into time-sensitive workflows.

You and your team can work through these questions and articulate additional requirements relevant to your specific context.

Scaling up

Scaling up in machine learning generally means that you need to work with a larger volume of data, or that you need processes to execute faster. Recent advances in hardware and software make the execution of complex computations magnitudes faster and more efficient than they were even a decade ago, and you can often achieve quite a bit by working on a personal computer. Yet, time is valuable, and it can be difficult to iterate and experiment effectively when individual processes take too long to execute.

There are many ML software packages that can help you make efficient use of whatever hardware you have, including your personal computer. Some examples at the time of writing are Apache Spark, TensorFlow, Scikit-learn, and Microsoft Cognitive Toolkit, each with their own strengths and applications. In addition to providing libraries for building and testing models, these software packages optimize algorithmic performance, memory resources, data throughputs, and/or parallel computations. They can make a remarkable difference in both processing speed and the amount of data you can comfortably handle. There are also services that allow you to submit executable code and data to the cloud for processing, such as Google AI Platform.

Managing your own hardware upgrades is not without challenge. You may be lucky enough to have access to a high-powered computer capable of accelerated processing. A common example is a computer with GPUs (graphics processing units), which break complex processes into many small tasks and run them in parallel. However, these powerful machines can be prohibitively expensive. Another scaling technique is distributed or cluster computing, in which complex processes are distributed across multiple computers, often in the cloud. A cloud cluster can bring significant cost savings, but managing one requires specialized knowledge and the learning curve can be rather steep. It is also important to note that different algorithms require different scaling techniques. Some clustering algorithms, for example, scale well with GPUs but not with distributed computing.

Even with the right hardware and software, scaling up can be a tricky business. ML processes tend to have dramatic spikes in memory or network use, which can tax your systems. Not all ML algorithms scale well, causing memory use or execution time to grow exponentially as more data is added. Sometimes you have to add additional, complexity-reducing steps to your pipeline to

handle data at scale. Some of the more common machine learning languages, such as Python and R, execute relatively slowly, putting the onus on developers to optimize operations for efficiency. In anticipation of these and other challenges, it is often a good idea to start with a scaled-down pilot or proof of concept, and not to underestimate the time and resources necessary to scale up from there.

Conclusion

New technologies make it possible for more researchers and developers to leverage the power of machine learning. Building an effective machine learning system means supporting the entire workflow, from data acquisition to final analysis. Practitioners must be mindful of how each implementation decision and subjective choice—from the way you structure and store your data to the algorithms you use to the ways you validate your results—will impact the efficiency of operations and the quality of learned intelligence. This article has offered some practical guidelines for building ML systems with modular, repeatable processes and intelligible, verifiable results. There are many resources available for further research, both online and in your libraries, and I encourage you to consult with subject specialists, data scientists, mathematicians, programmers, and data engineers. May your data be clean, your computations efficient, and your results profound.

Further Reading

I include here a few suggestions for further reading on key topics. I have also found that in the fast-changing world of machine learning technologies, blogs, internet communities, and online classes can be a great source of information that is current, introductory, and/or geared toward practitioners.

Tan, Pang-Ning, Michael Steinbach, and Vipin Kumar. 2005. *Introduction to Data Mining*. Boston: Pearson Addison Wesley. See chapter 2 for data preparation strategies. Later chapters introduce common classification and clustering algorithms.

Marz, Nathan and James Warren. 2015. *Big Data: Principles and best practices of scalable realtime data systems*. Shelter Island: Manning. "Part 1: Batch Layer" discusses immutable storage in depth.

Kleppmann, Martin. 2017. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. Boston: O'Reilly. "Chapter 10: Batch Processing" is especially relevant if you are interested in scaling up.