MANNING

Reactive Machine Learning Systems

MEAP

Jeff Smith

MEAP Edition
Manning Early Access Program
Reactive Machine Learning Systems
Version 2

Chapter 1

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

brief contents

1 Learning reactive machine learning

This chapter covers:

- The components of machine learning systems
- The reactive systems design paradigm
- The reactive approach to building machine learning systems

This book is all about how to build machine learning systems, which are sets of software components capable of learning from data and making predictions about the future. This chapter will discuss the challenges of building machine learning systems and some approaches to overcoming those challenges. The example we'll look at is of a startup that tries to build a machine learning system from the ground up and finds it very, very hard.

If you've never built a machine learning system before, you might find it challenging and a bit confusing. My goal is to take some of the pain and mystery out of this process. I won't be able to teach you everything there is to know about the techniques of machine learning; that would take a mountain of books. Instead, we'll focus on how to build a system that can put the power of machine learning to use.

I'll introduce you to a fundamentally new and better way of building machine learning systems called *reactive machine learning*. Reactive machine learning represents the marriage of ideas from reactive systems and the unique challenges of machine learning. By understanding the principles that govern these systems, you'll see how to build systems that are more capable, both as software and as

predictive systems. This chapter will introduce you to the motivating ideas behind this approach, laying a foundation for the techniques you'll learn in the rest of the book.

1.1 An example machine learning system

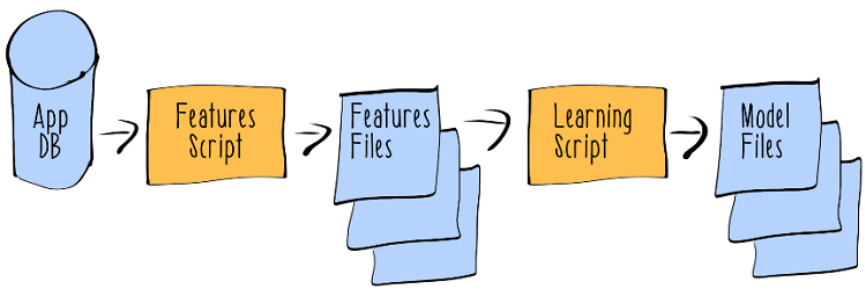
Consider the following scenario. Sniffable is the Facebook for dogs. It's a startup based out of a dog-filled loft in New York. Using the Sniffable app, dog owners post pictures of their dogs, and other dog owners like, share, and comment on those pictures. The network was growing well, but the team felt there might be a meteoric opportunity here. But if Sniffable was really going to take off, it was pretty clear that they'd have to build more than just the standard social networking features.

1.1.1 Building a prototype system

Sniffable users, called *sniffers*, are all about promoting their specific dog. Many sniffers hope that their dog will achieve canine celebrity status. The team had an idea that what sniffers really wanted were tools to help make their posts, called *pupdates*, more viral. Their initial concept for the new feature was a sort of competitive intelligence tool for the canine equivalent of stage moms, internally known as *den mothers*. The belief was that den mothers were taking many pictures of their dogs, and were trying to figure out which picture would get the biggest response on Sniffable. The team intended the new tool to predict the number of likes a given pupdate might get, based on the hashtags used. They named the tool *Pooch Predictor*. It was their hope that it would engage the den mothers, help them create viral content, and grow the Sniffable network as a whole.

The team turned to their lone data scientist to get this product off the ground. The initial spec for the minimal viable product was pretty fuzzy, and the data scientist was already a pretty busy guy, since he was the entire data science department. So, over the course of several weeks, he stitched together a system that looked something like figure 1.1.

Figure 1.1. Pooch Predictor 1.0 Architecture



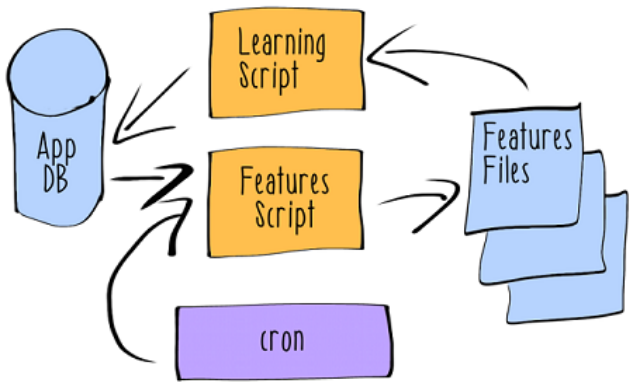
The app already sent all raw user interaction data to the application's relational database, so he decided to start building his model with that data. He wrote a simple script that dumped the data he wanted to flat files. Then he processed that interaction data using a different script to produce derived representations of the data, the features and the concepts. This script produced a structured representation of a pupdate, the number of likes it got, and other relevant data like the hashtags associated with the post. Again, this script just dumps its output to flat files. Then he ran his model learning algorithm over his files to produce a model that predicted likes on posts, given the hashtags and other data about the post.

The team was thoroughly amazed by this prototype of a predictive product, and they pushed it through the engineering roadmap to get it out the door as soon as possible. They assigned a junior engineer to the job of taking the data scientist's prototype and getting it running as a part of the overall system. He decided to embed the data scientist's model directly into the app's post creation code. This made it easy to display the predicted number of likes in the app.

A few weeks after Pooch Predictor went live, the data scientist happened to notice that the predictions weren't changing much, so he asked the engineer about the retraining frequency of the modeling pipeline. The engineer had no idea what the data scientist was talking about. They eventually figured out that the data scientist intended his scripts to be run on a daily basis over the latest data from the system. Every day there should be a new model in the system to replace the old one.

These new requirements changed how the system needed to be constructed, resulting in the architecture seen in figure 1.2.

Figure 1.2. Pooch Predictor 1.1 Architecture



In this version of Pooch Predictor, the scripts were run on a nightly basis, scheduled by cron. They still dumped their intermediate results to files, but now they needed to insert their models into the application's database. Now the backend server was responsible for producing the predictions displayed in the app. It would

pull the model out of the database and use it to provide predictions to the app's users.

This new system was definitely better than the first version, but in its first several months of operation, the team discovered several pain points with it.

First of all, Pooch Predictor wasn't very reliable. Often something would change in the database, and one of the queries would fail. Other times, there would be high load on the server, and the modeling job would fail. This was happening more and more as both the size of the social network and the size of the dataset used by the modeling system increased. One time, the server that was supposed to be running the data processing job failed, and all of the relevant data was lost. These sorts of failures were hard to detect without building up more sophisticated monitoring and alerting infrastructure. But even if someone did detect a failure in the system, there wasn't much that could be done other than kick off the job again and hope that it succeeded this time.

Other than these big system-level failures, the data scientist started to find other problems in Pooch Predictor. Once he got at the data, he realized that some of the features weren't being correctly extracted from the raw data. It was also really hard to understand how a change to the features that were being extracted would impact modeling performance, so he felt a little blocked from making improvements to the system.

There was also a major issue that ended up involving the entire team. For a period of a couple of weeks, the team saw their interaction rates steadily trend down with no real explanation. Then someone noticed a problem with Pooch Predictor while testing on the live version of the app. For users who were based outside of the US, Pooch Predictor would always predict a negative number of likes for their pupdates. In forums around the internet, disgruntled users were voicing their rage at having the adorableness of their particular dog insulted by the Pooch Predictor feature. Once the Sniffable team detected the problem, they were able to quickly figure out that it was a problem with the modeling system's location-based features. The data scientist and engineer paired on a fix, and the problem went away, but not after seriously hurting their credibility with sniffers based abroad.

Shortly after that issue, Pooch Predictor ran into more problems. It started with the data scientist implementing more feature-extraction functionality to hopefully improve modeling performance. To do that, he got the engineer's help to send more data from the user app back to the application database. On the day that the new functionality rolled out, the team saw immediate issues. The app slowed down dramatically. Posting was now a very laborious process, since each button tap seemed to take several seconds. Sniffers were seriously irritated with these issues. Things went from bad to worse when Pooch Predictor started to cause yet more problems with posting. It turned out that the new functionality caused exceptions to be thrown on the server, which led to pupdates being dropped.

At this point it was all hands on deck in a furious effort to put out this fire. They realized that there were two major issues with the new functionality:

- Sending the data from the app back to the server required a transaction. When the data scientist and engineer added more data to the total amount of data being collected for modeling, this transaction took way too long to maintain reasonable responsiveness within the app.
- The prediction functionality within the server that supported the app didn't handle the new features properly. The server would throw an exception any time the prediction functionality saw any of the new features that had been added in another part of the application.

After understanding where things had gone wrong, the team quickly rolled back all of the new functionality and restored the app to a normal operational state.

1.1.2 Building a better system

Everyone on the team agreed that something was wrong with the way that they were building their machine learning system. They held a retrospective to help the team figure out what went wrong and how they were going to do better in the future. The outcome was the following vision for what a Pooch Predictor replacement needed to look like:

Operational characteristics

- The Sniffable app must remain responsive, regardless of any other problems with the predictive system.
- The predictive system needs to be much less tightly coupled to the rest of the systems.
- The predictive system needs to behave predictably regardless of high load or errors in the system itself.

Development process

- It should be easier for different developers to make changes to the predictive system without breaking things.
- The code needs to use different programming idioms that ensure better performance when used consistently.

Predictive functionality

- The predictive system must measure its modeling performance better.
- The predictive system should support evolution and change.
- The predictive system should support online experimentation.
- It should be easy for humans to supervise the predictive system and rapidly correct any rogue behavior.

1.2 Reactive machine learning

In the previous example, it really seems like the Sniffable team missed something big, right? They built what initially looked like a useful machine learning system that added value to their core product. But all the issues they had in getting there obviously had a cost. Production issues with their machine learning system frequently pulled the team away from work on improvements to the capability of the system. Even though they had a bunch of smart people in the room thinking hard about how to predict the dynamics of dog-based social networking, their system repeatedly failed at its mission.

1.2.1 Machine learning

Building machine learning systems that do what they're supposed to do is *hard*, but it's not impossible. In the previous example, the data scientist knew how to *do* machine learning. Pooch Predictor totally worked on his laptop; it made predictions from data. But the data scientist was not thinking of machine learning as an *application*—he only understood machine learning as a *technique*. Pooch Predictor did not consistently produce trustable, accurate predictions. It was a failure both as a predictive system and as a piece of software.

In this book, I will show you how to build machine learning systems that are just as awesome as the best web and mobile applications. But understanding how to build these systems will require you to think of machine learning as an application, and not merely as a technique. The systems that we'll build won't fail at their missions.

In the next section, we'll get into the reactive approach to building machine learning systems. But first I want to clarify what a machine learning system is and how it differs from merely using machine learning as a technique. To do so, I'll have to introduce some terminology. If you have experience with machine learning, some of this might seem basic, but bear with me. Terms related to machine learning can be pretty inconsistently defined and used, so I want to be explicit about what we're talking about.

Functionality vs. implementation

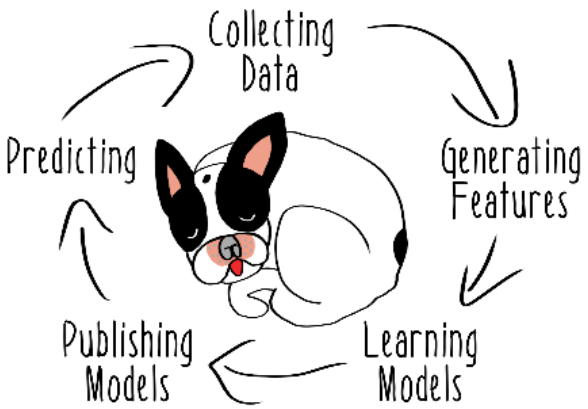
Note that this brief introduction is only focused on ensuring that you're sufficiently oriented in terms of the *functionality* of a machine learning system. This book is focused on the *implementation* of machine learning systems, not on the fundamentals of machine learning itself. Should you find yourself needing a better introduction to the techniques and algorithms used in machine learning, I recommend reading *Real World Machine Learning* by Henrik Brink, Joseph W. Richards, and Mark Fetherolf (Manning, 2016).

At its simplest, machine learning is a technique for learning from and making predictions on data. At a minimum, to *do* machine learning, you must take some data, learn a model, and use that model to make predictions. Using this definition, we can imagine an even cruder form of the Pooch Predictor example. It could be a

program that queries the application database for the most popular breed of dog (French Bulldogs, it turns out) and tells the app to say that all posts containing a French Bulldog will get a lot of likes.

That minimal definition of machine learning leaves out a lot of relevant detail. Most real-world machine learning systems need to do a lot more than just that. They will usually need to have all of the components shown in Figure 1.4.

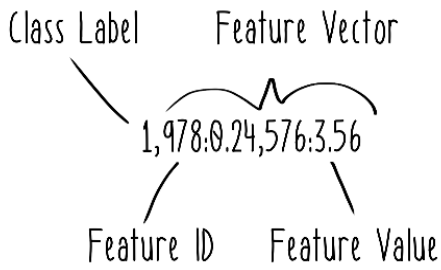
Figure 1.3. Components of a Machine Learning System



Starting from the beginning, a machine learning system must collect data from the outside world. In the Pooch Predictor example, the team was trying to skip this concern by using the data that their application already had. Obviously, this approach was quick, but it tightly couples the Sniffable application data model to the Pooch Predictor data model. How to collect and persist data for a machine learning system is a large and important topic, so I'll spend all of Chapter 3 showing you how to set your system up for success.

Once the system has data in it, that data is rarely ready to send off to a machine learning algorithm. Most machine learning algorithms are applied to derived representations of the raw data called*instances*. Figure 1.5 shows the parts of an instance in a common syntax (LibSVM).

Figure 1.4. The Structure of an Instance



There are many syntaxes that can be used to express instances, so we're not going to worry too much about the specifics of any syntax. However they are expressed, instances are always made up of the same components.

Features are meaningful data points derived from raw data related to the entity being predicted on, at the time you're trying to make a prediction. A Sniffable example of a feature would be the number of friends a given dog has. In Figure 1.4, features are expressed using a unique ID field and feature value. Feature number 978, which might represent the sniffer's proportion of friends that are male dogs, has a value of 0.24. Typically, a machine learning system will extract many features from the raw data available to it. The feature values for a given instance are collectively called a*feature vector*.

A *concept* is the thing that the system is trying to predict. In the context of Pooch Predictor, a concept would be the number of likes a given post receives. When a concept is discrete (i.e. not continuous), it can be called a *class label*, and you'll often see just the word *label* used in the relevant parts of machine learning libraries, such as MLlib, which we'll use in this book.

Only some sorts of machine learning problems involve having concepts available in the form of class labels. This sort of machine learning context is known as *supervised learning*, and most of the material in this book will be focused on this type of machine learning problem, although reactive machine learning could be applied to unsupervised learning problems as well.

Defining and implementing the best features and concepts to represent the problem you're trying to solve is an enormous portion of the work of real-world machine learning. From an application perspective, these tasks are the beginning of your data pipeline. Constructing pipelines that do this job reliably, consistently, and scalably requires a principled approach to application architecture and programming style. I'll spend Chapter 4 discussing the reactive approach to this part of machine learning systems under the banner of feature generation.

Using the data prepared in the previous steps, you're now ready to learn a model. You can think of a *model* as a program that maps from features to predicted

concepts, as shown in the simple Scala implementation in Listing 1.1.

Listing 1.1. A Simple Model

```
def genericModel(f: FeatureVector[RawData]): Prediction[Concept] = ???
```

Learning models occurs during the latter half of the data pipeline. A model produced by Pooch Predictor would be a program that takes as input the feature representation of the hashtag data and returns the predicted number of likes that a given pupdate might receive. This model is shown in Listing 1.2.

Listing 1.2. A Pooch Predictor Model

```
def poochPredictorModel(f: FeatureVector[Hashtag]): Prediction[Like] = ???
```

During this same phase of the pipeline, you'll need to begin to address several different types of uncertainty that crop up in building models. As a result, the model learning phase of the pipeline is concerned with more than just learning models. In Chapter 5, I'll discuss the various concerns that you'll need to consider in the model learning subsystem of a machine learning system.

Next, you'll need to take this model and make it useful by publishing it. *Model publishing* means making the model program available outside of the context it was learned in, so that it can make predictions on data it hasn't seen before. It's easy to gloss over the difficulties that come up in this part of a machine learning system, and the Sniffable team largely skipped it in their original implementation. They didn't even set up their system to retrain the model on a regular basis. Their next approach at implementing model retraining also ran into difficulty, causing their models to be out of sync with their feature extractors. There are ways of doing this better (hint: think about immutability), and I'll discuss them in Chapter 6.

Finally, you'll need to implement functionality for your learned model to be used in *predicting* concepts from new instances. This is ultimately where the rubber meets the road in a machine learning system, and in the Pooch Predictor system it was frequently where the car burst into flames. Given that team Sniffable had never really built a machine learning system like this before, it's not surprising that there were some pain points where their ideas met harsh reality. Some of their problems stemmed from treating their predictive system like a transaction business application that needed to record a purchase. An approach that relies upon strong consistency guarantees just doesn't work for modern distributed systems, and it's simply out of synch with the pervasive and intrinsic uncertainty in a machine learning system. Other problems the Sniffable team experienced had to do with not thinking about their system in dynamic terms. Machine learning systems must evolve, and they must support parallel tracks for that evolution through

experimentation capabilities. In Chapter 7, I'll show you how to build much better prediction functionality using very different ideas than the naive approach in the Pooch Predictor example.

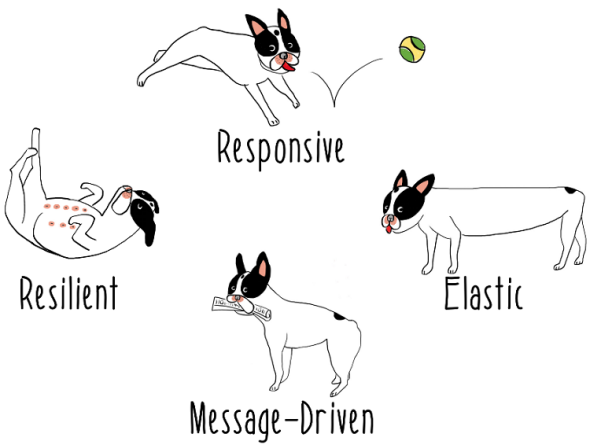
1.2.2 Reactive systems

Now that you understand a bit more about what a machine learning system is, I want to give you an overview of some of the ideas and approaches that we'll use to build successful ones. We'll begin with the reactive systems paradigm. Reactive systems are defined by four traits and three strategies. The paradigm as a whole is a way of codifying an approach to building systems that can serve modern user expectations for things like interactivity and availability.

TRAITS OF REACTIVE SYSTEMS

Reactive systems privilege four traits (see Figure 1.6).

Figure 1.5. The Traits of Reactive Systems



First and most importantly, reactive systems are *responsive*, meaning that they consistently return timely responses to users.

Responsiveness is the crucial foundation upon which all of future development efforts are going to be built. If a system doesn't respond to its users, then it's useless. Think of the Sniffable team causing a massive slowdown in the Sniffable app due to the responsiveness of their machine learning system.

Whether the cause is hardware, human error, or design flaws, software always breaks, as the Sniffable team has discovered. Providing some sort of acceptable response even when things don't go as planned is a key part of ensuring that users view a system as being responsive. It doesn't matter that an app is very fast when it's not broken if it's broken half the time.

Reactive systems must also be *elastic*; they need to remain responsive despite varying load.

The idea of elasticity isn't exactly the same as scalability, although the two are similar. Elastic systems should respond to increases *or* decreases in load. The Sniffable team saw this when their traffic ramped up and the Pooch Predictor system couldn't keep up with the load. That is exactly what a lack of elasticity looks like.

The idea of elasticity isn't exactly the same as scalability, although the two are similar. Elastic systems should respond to increases *or* decreases in load. The Sniffable team saw this when their traffic ramped up and the Pooch Predictor system couldn't keep up with the load. That is exactly what a lack of elasticity looks like.

Finally, reactive systems are *message-driven*; they communicate via asynchronous, non-blocking message passing.

The message-passing approach is in contrast with direct intra-process communication or other forms of tight coupling. It's easy to understand how a more explicit approach to ensuring loose coupling might solve some of the issues in the Sniffable example. A loosely coupled system organized around message passing can make it easier to detect failure or issues with load. Moreover, a design with this trait helps contain any of the effects of errors to just messages about bad news, rather than flaming production issues that needed to be immediately addressed, as they were in Pooch Predictor.

The reactive approach could certainly be applied to the problems the Sniffable team was having with their machine learning system. The four principles represent a coherent and complete approach to system design that make for fundamentally better systems. Such systems fulfill their requirements better than naively designed systems, and they are more fun to work on. After all, who wants to fight fires when you could be shipping awesome new machine learning functionality to loyal sniffers?

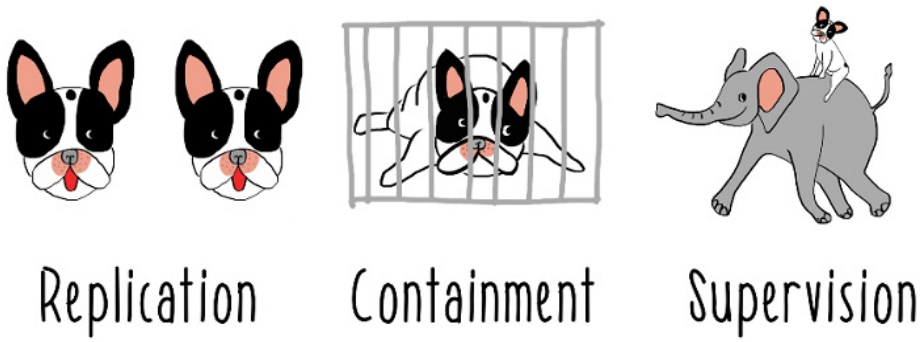
These traits certainly sound nice, but they're not much of a plan. How do you build a system that actually has these traits? Message passing is part of the answer, but it's not the whole story. Also, we're talking about machine learning systems, which

as you've seen, can be difficult to get right. They have unique challenges that will likely need unique solutions that don't come up in traditional business applications.

REACTIVE STRATEGIES

A key part of how we'll build a reactive machine learning system in this book is the reactive strategies (Figure 1.11).

Figure 1.10. Reactive Strategies



First, reactive systems use *replication*. They have the same component executing in more than one place at the same time.

More generally, this means that data, whether at rest or in motion, should be redundantly stored or processed.

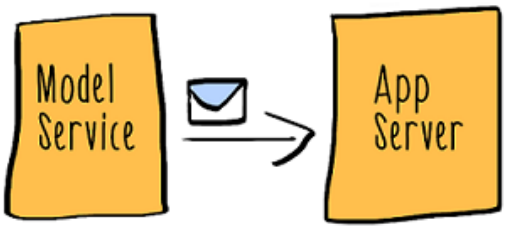
In the Sniffable example, there was a time when the server that ran the model-learning job failed, and no model was learned. Clearly, replication could have helped here. Had there been two or more model-learning jobs, the failure of one job would have had less impact. That sounds pretty wasteful, but it's actually the beginning of a solution. As you'll see in Chapters 4 and 5, you can build replication into your modeling pipelines using Spark. Rather than requiring you to always have two pipelines executing, Spark gives you automatic, fine-grained replication so that the system can recover from failure. This book will focus on the use of higher-level tools like Spark to manage the challenges of distributed systems. By relying on these tools, you can easily use replication in every component of your machine learning system.

Next, reactive systems use *containment* to prevent the failure of any single component of the system from affecting any other component.

The term *containment* might get you thinking about specific technologies like Docker or rkt, but this strategy isn't about any one implementation. Containment can be implemented using many different systems, including homegrown ones. The point is to prevent the sort of cascading failure we saw in Pooch Predictor, and to do so at a structural level.

Consider the issue with Pooch Predictor where the model and the features were out of sync, resulting in exceptions during model serving. This was only a problem because the model-serving functionality wasn't sufficiently contained. Had the model been deployed as a contained service communicating with the Sniffable application server via message passing, there would have been no way for this failure to have propagated as it did. Figure 1.14 shows an example of this architecture.

Figure 1.13. A Contained Model-Serving Architecture

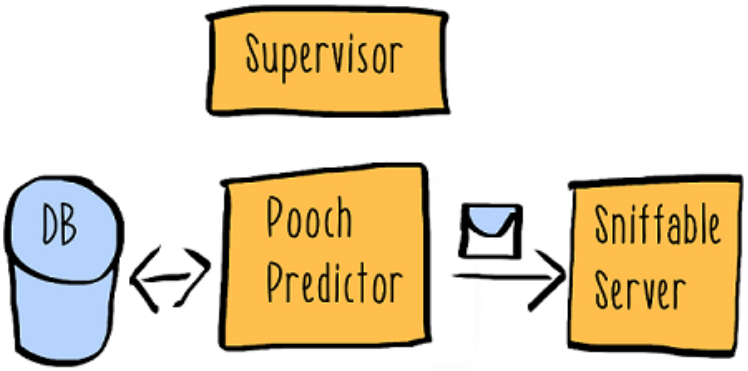


Lastly, reactive systems rely on the strategy of *supervision* to organize components.

When implementing systems using this strategy, you explicitly identify the components that could fail and make sure that some other component is responsible for their lifecycles. The strategy of supervision gives you a point of control, where you can ensure that the reactive traits are being achieved by the actual runtime behavior of your system.

The Pooch Predictor system had no system-level supervision. This unfortunate omission left the Sniffable team scrambling whenever something went wrong with the system. A better approach would have been to build supervision directly into the system itself, along the lines of Figure 1.15.

Figure 1.15. A Supervisory Architecture



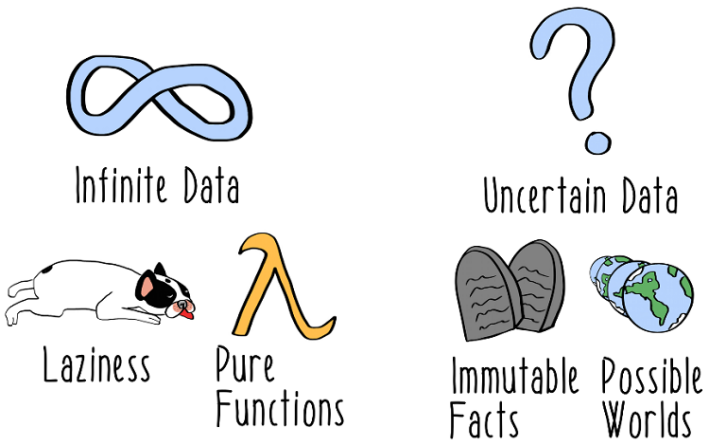
In this structure, the published models are observed by the model supervisor. Should their behavior ever deviate from acceptable bounds, the supervisor would stop sending them messages requesting predictions. In fact, the model supervisor could even completely destroy a model it knows to be bad, allowing the system to be self-healing. I'll begin discussing how you can implement model supervision in Chapters 6 and 7, and we'll continue exploring powerful applications of the strategy of supervision throughout the remainder of the book.

1.2.3 Making machine learning systems reactive

Now that we understand reactive systems, we can begin to discuss how we can apply these ideas to machine learning systems. In a reactive machine learning system, we still want our system to have all of the same traits as a reactive system, and we can use all of the same strategies. But we can do more to address the unique characteristics of a machine learning system. So far, I've discussed a lot of infrastructural concerns, but I haven't yet shown you how this enables new *predictive* capabilities. Ultimately, a reactive machine learning system gives you the ability to deliver value through ever better predictions. That's why reactive machine learning is worth understanding and applying.

The reactive machine learning approach is based on two key insights into the characteristics of data in a machine learning system: that it is uncertain and that it is effectively infinite. From those two insights, four strategies emerge that will help us build a reactive machine learning system (see Figure 1.17).

Figure 1.16. Reactive Machine Learning Data and Strategies



To begin, let's think about how much data the Pooch Predictor system might need to process. Ideally, with its new machine learning capabilities, Sniffable is going to take off and see tons of traffic. Even if that doesn't happen, there's still no way of knowing how many possible pupdates users might want to consider and thus send to the Pooch Predictor system. Imagine having to predict every possible post that a sniffer might make on Sniffable. Some posts would have big dogs, others small ones. Some posts would use filters, and others would be more natural. Some would be rich in hashtags, and some wouldn't have any annotations. Once you consider the impact of arbitrary parameters on feature values, the range of possible data representations becomes literally *infinite*.

It doesn't matter precisely how much raw data the Pooch Predictor ingests. We'll always assume that the amount of data is too much for one thread or one server. But rather than give up in the face of this unbounded scope, reactive machine learning employs two strategies to manage infinite data.

First, it relies on *laziness*, also known as *delay of execution*, to separate the composition of functions to execute from their actual execution. Rather than being a bad habit, laziness is a powerful evaluation strategy that can greatly improve the design of data-intensive applications.

By using laziness in the implementation of your machine learning system, you'll find that it's much easier to conceive of the data flow in terms of infinite streams instead of finite batches. This switch can have huge benefits for the responsiveness and utility of your system. I'll show how laziness can be used to build machine learning pipelines in Chapter 4.

Similarly, reactive machine learning systems deal with infinite data by expressing transformations as pure *functions*.

What does it mean for a function to be pure? First, evaluating the function must not result in some sort of side effect, such as changing the state of some variable or performing some I/O. Additionally, the function must always return the same value when given the same arguments. This latter property is referred to as *referential transparency*. Writing machine learning code that maintains this property can make implementations of mathematical transformations look and behave quite similarly to their expression in math.

Pure functions are a foundational concept in a style of programming known as *functional programming*, which I'll use throughout this book. At its heart, functional programming is all about computing with functions. In functional code, functions can be passed to other functions as arguments. Such functions are called *higher-order functions*, and we'll use this idiom throughout the code examples in this book. Functional programming idioms like higher-order functions are a key part of what makes reactive tools like Scala and Spark so powerful.

The emphasis on the use of functional programming in this book is not merely stylistic. Functional programming is one of the most powerful tools for taming complicated systems that need to reason about data, especially infinite data. The recent increase in the popularity of functional programming has been largely driven by its application to building big data infrastructure. Using the techniques of functional programming, we will be able to get our system right *and* scale it to the next level. As I'll discuss in Chapters 4 and 6, pure functions can offer real solutions to the problems of implementing feature extraction and prediction functionality.

Next, let's consider what Pooch Predictor knew about what was going on with Sniffable and its users. It had records of sniffers creating, viewing, and liking pupdates. This knowledge came from the main application database. Of course, as we saw, the app would sometimes lose sniffers' efforts to like a particular pupdate, due to operational issues, and this loss of data changed the concept that Pooch Predictor was built to learn. Similarly, Pooch Predictor's view of what feature values were seen at a given time was often impeded by bugs in its code or in the main app's code. This is all because *uncertainty* is intrinsic and pervasive in a machine learning system.

Machine learning models and the predictions they make are always approximate and only useful in the aggregate. It wasn't like Pooch Predictor knew *exactly* how many likes a given pupdate might get. Even before making a prediction, a machine learning system must deal with the uncertainty of the real world outside of the machine learning system. For example, do sniffers using the hashtag #adorabull

mean the same thing as sniffers using the hashtag #adorable, or should those be viewed as different features? A truly reactive machine learning system incorporates this uncertainty into the design of the system and uses two strategies to manage it: immutable facts and possible worlds.

It might sound strange to use facts to manage uncertainty, but that's exactly what we're going to do. Consider the location that a sniffer is posting a pupdate from. One way of recording this location data for later use in geographic features is to record the exact location reported by the app, as in Table 1.1.

Table 1.1. Pupdate Location Data Model

pupdate_id	location
123	Washington Square Park

But the location determined by the app at the time of the pupdate was uncertain; it was just the result of a sensor reading on a phone, which has a very coarse level of precision. The sniffer may or may not have been in Washington Square Park. Moreover, if a future feature tries to capture the distinct differences between East and West Greenwich Village, this data model will give a precise but potentially inaccurate view of how far to the east or west this pupdate came from.

A richer, more accurate way of recording this data is to use the raw location reading and the expected radius of uncertainty, as shown in Table 1.2.

Table 1.2. Revised Pupdate Location Data Model

pupdate_id	latitude	longitude	radius
123	40.730811	-73.997472	1.0

This revised data model can now represent immutable *facts*. This data can simply be written once and never modified; it is written in stone.

The use of immutable facts will allow us to reason about uncertain views of the world at specific points in time. This is crucial for creating accurate instances and many other important data transformations in a machine learning system. Having a complete record of all facts that occurred over the lifetime of the system will also enable important machine learning, like model experimentation and automatic model validation.

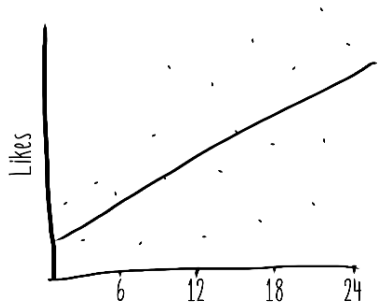
The use of immutable facts will allow us to reason about uncertain views of the world at specific points in time. This is crucial for creating accurate instances and many other important data transformations in a machine learning system. Having a complete record of all facts that occurred over the lifetime of the system will also

enable important machine learning, like model experimentation and automatic model validation.

To understand the other strategy for dealing with uncertainty, let's consider a fairly simple question: how many likes will pupdates about French Bulldogs get in the next hour? To answer this question, let's break it down into pieces.

First, how many pupdates will be submitted in the next hour? There are multiple ways of answering this question. We could just take the historical average rate, say 6,500. But the number of pupdates submitted varies over time, so we could also fit a line to the data that looks something like Figure 1.23.

Figure 1.22. Model of Likes by Hour



Using this model, we might expect 7,250 pupdates in the next hour.

Beyond that, we need to know how many likes these pupdates will receive. Again, we could take a historical average, which would give us 23 likes per pupdate in this case. Or we could use a model. That model would have to be applied to some recent sample of data to get an idea of the likes that recent traffic has been getting. The result of this model is that the average pupdate will receive 28 likes.

Now, we need to combine this information in some way. Table 1.3 shows the predictions that we could use in our final prediction.

Table 1.3. Possible Prediction Values

Model Type	Pupdates	Likes/Pupdate
Historical	6,500	23
Machine-Learned	7,250	28

We could decide to answer that the expected number of likes in the next hour is `6,500 * 23 = 149,500` using the historical values. Or we could decide to use the machine-learned model and get a value of `7,250 * 28 = 203,300`. We could even decide to combine the historical number of pupdates with the model-based

prediction of likes per pupdate to get `6,500 * 28 = 182,000`. These different views of our uncertain data can be thought of as *possible worlds*.

We don't know which of these worlds we will ultimately find ourselves in during the next hour of traffic on Sniffable, but we can make decisions with this information, like ensuring that the servers are prepared to handle more than 200,000 likes in the next hour. Possible worlds will form the basis for the queries we will make of all of the uncertain data that is present in our machine learning system. There are limits to the applicability of this strategy, because infinite data can produce infinite possible worlds. But by building our data models and queries with the concept of possible alternative worlds, we'll be able to more effectively reason about the real range of potential outcomes in our system.

We don't know which of these worlds we will ultimately find ourselves in during the next hour of traffic on Sniffable, but we can make decisions with this information, like ensuring that the servers are prepared to handle more than 200,000 likes in the next hour. Possible worlds will form the basis for the queries we will make of all of the uncertain data that is present in our machine learning system. There are limits to the applicability of this strategy, because infinite data can produce infinite possible worlds. But by building our data models and queries with the concept of possible alternative worlds, we'll be able to more effectively reason about the real range of potential outcomes in our system.

Using all of the strategies that we've discussed, it's easy to imagine the Sniffable team refactoring the Pooch Predictor system into something much more powerful. The reactive machine learning approach makes it possible to build a machine learning system that has fewer problems and allows for evolution and improvement. It's definitely a different approach than we saw in the original Pooch Predictor example, and this approach is grounded on a firmer footing. Reactive machine learning unites ideas from distributed systems, functional programming, uncertain data, and other fields in a coherent, pragmatic approach to building real world machine learning systems.

WHEN NOT TO USE REACTIVE MACHINE LEARNING

It's fair to ask whether all machine learning systems should be built using the reactive approach. The answer is no.

During the design and implementation of a machine learning system, it's beneficial to consider the principles of reactive machine learning. Machine learning problems definitionally have to do with reasoning about uncertainty. Thinking in terms of immutable facts and pure functions is a useful perspective for implementing any sort of application.

But the approach discussed in this book is a way to easily build sophisticated systems, and some machine learning systems don't need to be sophisticated. Some

systems won't benefit from using a message-passing semantic that assumes several independently executing processes. Research prototypes are a perfect example of a machine learning system that doesn't need the powerful capabilities of a reactive machine learning system. When you're building a temporary system, I recommend bending or breaking all of the rules that I lay out in this book. The prudent approach to building potentially disposable machine learning systems is to make far more extreme compromises than the in reactive approach. If you're building such a temporary system, see my guide to building machine learning systems at hackathons: https://medium.com/data-engineering/modeling-madly-8b2c72eb52be.

1.3 Summary

In this chapter, I've introduced you to machine learning systems and the challenges of building them. I've also laid the groundwork for the reactive machine learning approach that we'll use to surmount those challenges in the rest of the book.

- Even simple machine learning systems can fail.
- Machine learning can be viewed as an application, and not just as a technique.
- A machine learning system is composed of five different components.
  - The data collection component is responsible for ingesting data from the outside world into the machine learning system.
  - The data transformation component is responsible for transforming raw data into useful derived representations of that data called features and concepts.
  - The model learning component is responsible for learning models from the features and concepts.
  - The model publishing component is responsible for making a model available to make predictions.
  - The model serving component is responsible for connecting models to requests for predictions.
- The reactive systems design paradigm is a coherent approach to building better systems.
  - Reactive systems are responsive, resilient, elastic, and message-driven.
  - Reactive systems use the strategies of replication, containment, and supervision as concrete approaches for maintaining the reactive traits.
- Reactive machine learning is an extension of the reactive systems approach that addresses the specific challenges of building machine learning systems.
  - Data in a machine-learning system is effectively infinite.
    - Laziness, or delay of execution, is a way of conceiving of infinite flows of data, rather than finite batches.
    - Pure functions without side effects help us manage infinite data by ensuring that functions behave predictably, regardless of context.

- Uncertainty is intrinsic and pervasive in the data of a machine-learning system.
  - Writing all data in the form of immutable facts makes it easier to reason about views of uncertain data at points in time.
  - Different views of uncertain data can be thought of as possible worlds that can be queried across.

In the next chapter, I'll introduce some of the technologies and techniques we'll be using to build reactive machine learning systems. You'll see how reactive programming techniques will allow us to deal with complex system dynamics without complex code. I'll also introduce two powerful frameworks, Akka and Spark, that will allow us to build incredibly sophisticated reactive systems easily and quickly.