1 | Decision Trees

> The words printed here are concepts.
> You must go through the experiences. – Carl Frederick

Learning Objectives:

- Explain the difference between memorization and generalization.
- Implement a decision tree classifier.
- Take a concrete task and cast it as a learning problem, with a formal notion of input space, features, output space, generating distribution and loss function.

Dependencies: None.

At a basic level, machine learning is about predicting the future based on the past. For instance, you might wish to predict how much a user Alice will like a movie that she hasn't seen, based on her ratings of movies that she has seen. This prediction could be based on many factors of the movies: their category (drama, documentary, etc.), the language, the director and actors, the production company, etc. In general, this means making informed guesses about some unobserved property of some object, based on observed properties of that object.

The first question we'll ask is: what does it mean to learn? In order to develop learning machines, we must know what learning actually means, and how to determine success (or failure). You'll see this question answered in a very limited learning setting, which will be progressively loosened and adapted throughout the rest of this book. For concreteness, our focus will be on a very simple model of learning called a **decision tree**.

1.1 What Does it Mean to Learn?

Alice has just begun taking a course on machine learning. She knows that at the end of the course, she will be expected to have "learned" all about this topic. A common way of gauging whether or not she has learned is for her teacher, Bob, to give her a exam. She has done well at learning if she does well on the exam.

But what makes a reasonable exam? If Bob spends the entire semester talking about machine learning, and then gives Alice an exam on History of Pottery, then Alice's performance on this exam will *not* be representative of her learning. On the other hand, if the exam only asks questions that Bob has answered exactly during lectures, then this is also a bad test of Alice's learning, especially if it's an "open notes" exam. What is desired is that Alice observes *specific* examples from the course, and then has to answer new, but related questions on the exam. This tests whether Alice has the ability to

generalize. Generalization is perhaps the most central concept in machine learning.

As a concrete example, consider a course recommendation system for undergraduate computer science students. We have a collection of students and a collection of courses. Each student has taken, and evaluated, a subset of the courses. The evaluation is simply a score from -2 (terrible) to $+2$ (awesome). The job of the recommender system is to **predict** how much a particular student (say, Alice) will like a particular course (say, Algorithms).

Given historical data from course ratings (i.e., the past) we are trying to predict unseen ratings (i.e., the future). Now, we could be unfair to this system as well. We could ask it whether Alice is likely to enjoy the History of Pottery course. This is unfair because the system has no idea what History of Pottery even is, and has no prior experience with this course. On the other hand, we could ask it how much Alice will like Artificial Intelligence, which she took last year and rated as $+2$ (awesome). We would expect the system to predict that she would really like it, but this isn't demonstrating that the system has learned: it's simply recalling its past experience. In the former case, we're expecting the system to generalize *beyond* its experience, which is unfair. In the latter case, we're not expecting it to generalize at all.

This general set up of predicting the future based on the past is at the core of most machine learning. The objects that our algorithm will make predictions about are **examples**. In the recommender system setting, an example would be some particular Student/Course pair (such as Alice/Algorithms). The desired prediction would be the rating that Alice would give to Algorithms.

To make this concrete, Figure 1.1 shows the general framework of **induction**. We are given **training data** on which our algorithm is expected to learn. This training data is the examples that Alice observes in her machine learning course, or the historical ratings data for the recommender system. Based on this training data, our learning algorithm induces a function f that will map a new example to a corresponding prediction. For example, our function might guess that $f(\text{Alice/Machine Learning})$ might be high because our training data said that Alice liked Artificial Intelligence. We want our algorithm to be able to make lots of predictions, so we refer to the collection of examples on which we will evaluate our algorithm as the **test set**. The test set is a closely guarded secret: it is the final exam on which our learning algorithm is being tested. If our algorithm gets to peek at it ahead of time, it's going to cheat and do better than it should.

The goal of inductive machine learning is to take some training data and use it to induce a function f. This function f will be evalu-

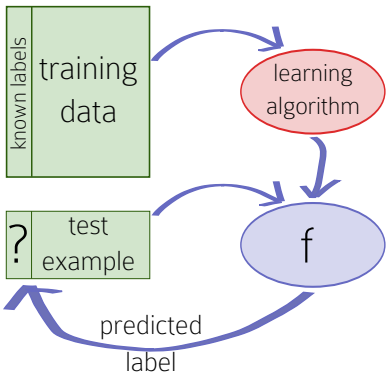

Figure 1.1: The general supervised approach to machine learning: a learning algorithm reads in training data and computes a learned function f. This function can then automatically label future text examples.

? Why is it bad if the learning algorithm gets to peek at the test data?

ated on the test data. The machine learning algorithm has succeeded if its performance on the test data is high.

1.2 Some Canonical Learning Problems

There are a large number of typical inductive learning problems. The primary difference between them is in what type of *thing* they're trying to predict. Here are some examples:

Regression: trying to predict a real value. For instance, predict the value of a stock tomorrow given its past performance. Or predict Alice's score on the machine learning final exam based on her homework scores.

Binary Classification: trying to predict a simple yes/no response. For instance, predict whether Alice will enjoy a course or not. Or predict whether a user review of the newest Apple product is positive or negative about the product.

Multiclass Classification: trying to put an example into one of a number of classes. For instance, predict whether a news story is about entertainment, sports, politics, religion, etc. Or predict whether a CS course is Systems, Theory, AI or Other.

Ranking: trying to put a set of objects in order of relevance. For instance, predicting what order to put web pages in, in response to a user query. Or predict Alice's ranked preferences over courses she hasn't taken.

? For each of these types of canonical machine learning problems, come up with one or two concrete examples.

The reason that it is convenient to break machine learning problems down by the type of object that they're trying to predict has to do with measuring error. Recall that our goal is to build a system that can make "good predictions." This begs the question: what does it mean for a prediction to be "good?" The different types of learning problems differ in how they define goodness. For instance, in regression, predicting a stock price that is off by \$0.05 is perhaps much better than being off by \$200.00. The same does not hold of multiclass classification. There, accidentally predicting "entertainment" instead of "sports" is no better or worse than predicting "politics."

1.3 The Decision Tree Model of Learning

The **decision tree** is a classic and natural model of learning. It is closely related to the fundamental computer science notion of "divide and conquer." Although decision trees can be applied to many

learning problems, we will begin with the simplest case: binary classification.

Suppose that your goal is to predict whether some unknown user will enjoy some unknown course. You must simply answer "yes" or "no." In order to make a guess, you're allowed to ask binary questions about the user/course under consideration. For example:

You: Is the course under consideration in Systems?
Me: Yes
You: Has this student taken any other Systems courses?
Me: Yes
You: Has this student liked most previous Systems courses?
Me: No
You: *I predict this student will not like this course.*

The goal in learning is to figure out what questions to ask, in what order to ask them, and what answer to predict once you have asked enough questions.

The decision tree is so-called because we can write our set of questions and guesses in a tree format, such as that in Figure 1.2. In this figure, the questions are written in the internal tree nodes (rectangles) and the guesses are written in the leaves (ovals). Each non-terminal node has two children: the left child specifies what to do if the answer to the question is "no" and the right child specifies what to do if it is "yes."

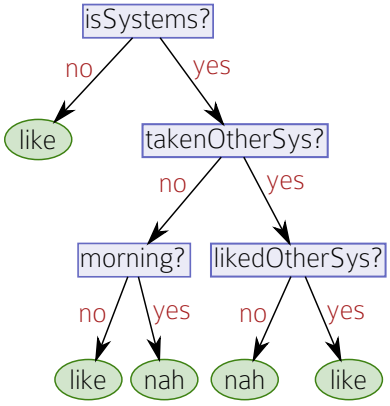

Figure 1.2: A decision tree for a course recommender system, from which the in-text "dialog" is drawn.

In order to learn, I will give you training data. This data consists of a set of user/course examples, paired with the correct answer for these examples (did the given user enjoy the given course?). From this, you must construct your questions. For concreteness, there is a small data set in Table 1 in the Appendix of this book. This training data consists of 20 course rating examples, with course ratings and answers to questions that you might ask about this pair. We will interpret ratings of 0, +1 and +2 as "liked" and ratings of −2 and −1 as "hated."

In what follows, we will refer to the questions that you can ask as **features** and the responses to these questions as **feature values**. The rating is called the **label**. An example is just a set of feature values. And our training data is a set of examples, paired with labels.

There are a lot of logically possible trees that you could build, even over just this small number of features (the number is in the millions). It is computationally infeasible to consider all of these to try to choose the "best" one. Instead, we will build our decision tree *greedily*. We will begin by asking:

If I could only ask one question, what question would I ask?

You want to find a feature that is *most useful* in helping you guess whether this student will enjoy this course. A useful way to think

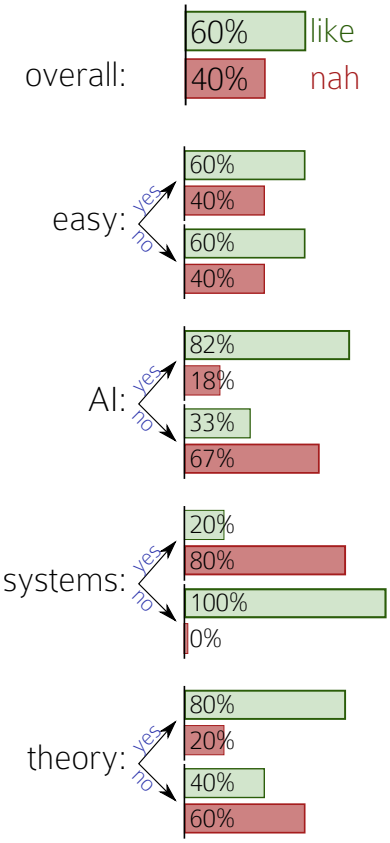

Figure 1.3: A histogram of labels for (a) the entire data set; (b-e) the examples in the data set for each value of the first four features.

about this is to look at the **histogram** of labels for each feature. [1] This is shown for the first four features in Figure 1.3. Each histogram shows the frequency of "like"/"hate" labels for each possible value of an associated feature. From this figure, you can see that asking the first feature is not useful: if the value is "no" then it's hard to guess the label; similarly if the answer is "yes." On the other hand, asking the second feature *is* useful: if the value is "no," you can be pretty confident that this student will hate this course; if the answer is "yes," you can be pretty confident that this student will like this course.

More formally, you will consider each feature in turn. You might consider the feature "Is this a System's course?" This feature has two possible value: no and yes. Some of the training examples have an answer of "no" – let's call that the "NO" set. Some of the training examples have an answer of "yes" – let's call that the "YES" set. For each set (NO and YES) we will build a histogram over the labels. This is the second histogram in Figure 1.3. Now, suppose you were to ask this question on a random example and observe a value of "no." Further suppose that you must *immediately* guess the label for this example. You will guess "like," because that's the more prevalent label in the NO set (actually, it's the *only* label in the NO set). Alternatively, if you receive an answer of "yes," you will guess "hate" because that is more prevalent in the YES set.

So, for this single feature, you know what you *would* guess if you had to. Now you can ask yourself: if I made that guess on the *training data*, how well would I have done? In particular, how many examples would I classify *correctly?* In the NO set (where you guessed "like") you would classify all 10 of them correctly. In the YES set (where you guessed "hate") you would classify 8 (out of 10) of them correctly. So overall you would classify 18 (out of 20) correctly. Thus, we'll say that the *score* of the "Is this a System's course?" question is 18/20.

? How many training examples would you classify correctly for each of the other three features from Figure 1.3?

You will then repeat this computation for each of the available features to us, compute the scores for each of them. When you must choose which feature consider first, you will want to choose the one with the highest score.

But this only lets you choose the *first* feature to ask about. This is the feature that goes at the *root* of the decision tree. How do we choose subsequent features? This is where the notion of divide and conquer comes in. You've already decided on your first feature: "Is this a Systems course?" You can now *partition* the data into two parts: the NO part and the YES part. The NO part is the subset of the data on which value for this feature is "no"; the YES half is the rest. This is the *divide* step.

[1] A colleague related the story of getting his 8-year old nephew to guess a number between 1 and 100. His nephew's first four questions were: Is it bigger than 20? (YES) Is it even? (YES) Does it have a 7 in it? (NO) Is it 80? (NO). It took 20 more questions to get it, even though 10 should have been sufficient. At 8, the nephew hadn't quite figured out how to divide and conquer. http://blog.computationalcomplexity.org/2007/04/getting-8-year-old-interested-in.html.

Algorithm 1 DECISIONTREETRAIN(*data*, *remaining features*)

```
guess ← most frequent answer in data                    // default answer for this data
if the labels in data are unambiguous then
    return LEAF(guess)                                  // base case: no need to split further
else if remaining features is empty then
    return LEAF(guess)                                  // base case: cannot split further
else                                                    // we need to query more features
    for all f ∈ remaining features do
        NO ← the subset of data on which f=no
        YES ← the subset of data on which f=yes
        score[f] ← # of majority vote answers in NO
                   + # of majority vote answers in YES
                                                        // the accuracy we would get if we only queried on f
    end for
    f ← the feature with maximal score(f)
    NO ← the subset of data on which f=no
    YES ← the subset of data on which f=yes
    left ← DECISIONTREETRAIN(NO, remaining features \ {f})
    right ← DECISIONTREETRAIN(YES, remaining features \ {f})
    return NODE(f, left, right)
end if
```

Algorithm 2 DECISIONTREETEST(*tree*, *test point*)

```
if tree is of the form LEAF(guess) then
    return guess
else if tree is of the form NODE(f, left, right) then
    if f = no in test point then
        return DECISIONTREETEST(left, test point)
    else
        return DECISIONTREETEST(right, test point)
    end if
end if
```

The *conquer* step is to recurse, and run the *same* routine (choosing the feature with the highest score) on the NO set (to get the left half of the tree) and then separately on the YES set (to get the right half of the tree).

At some point it will become useless to query on additional features. For instance, once you know that this is a Systems course, you *know* that everyone will hate it. So you can immediately predict "hate" without asking any additional questions. Similarly, at some point you might have already queried every available feature and still not whittled down to a single answer. In both cases, you will need to create a leaf node and guess the most prevalent answer in the current piece of the training data that you are looking at.

Putting this all together, we arrive at the algorithm shown in Algorithm 1.3.[2] This function, DECISIONTREETRAIN takes two argu-

[2] There are more nuanced algorithms for building decision trees, some of which are discussed in later chapters of this book. They primarily differ in how they compute the *score* function.

ments: our data, and the set of as-yet unused features. It has two base cases: either the data is unambiguous, or there are no remaining features. In either case, it returns a LEAF node containing the most likely guess at this point. Otherwise, it loops over all remaining features to find the one with the highest score. It then partitions the data into a NO/YES split based on the best feature. It constructs its left and right subtrees by recursing on itself. In each recursive call, it uses one of the partitions of the data, and removes the just-selected feature from consideration.

The corresponding *prediction* algorithm is shown in Algorithm 1.3. This function recurses down the decision tree, following the edges specified by the feature values in some *test point*. When it reaches a leaf, it returns the guess associated with that leaf.

? Is Algorithm 1.3 guaranteed to terminate?

1.4 Formalizing the Learning Problem

As you've seen, there are several issues that we must take into account when formalizing the notion of learning.

- The performance of the learning algorithm should be measured on unseen "test" data.
- The way in which we measure performance should depend on the problem we are trying to solve.
- There should be a strong relationship between the data that our algorithm sees at training time and the data it sees at test time.

In order to accomplish this, let's assume that someone gives us a **loss function**, $\ell(\cdot, \cdot)$, of two arguments. The job of ℓ is to tell us how "bad" a system's prediction is in comparison to the truth. In particular, if y is the truth and $\hat{y}$ is the system's prediction, then $\ell(y, \hat{y})$ is a measure of error.

For three of the canonical tasks discussed above, we might use the following loss functions:

Regression: **squared loss** $\ell(y, \hat{y}) = (y - \hat{y})^2$
or **absolute loss** $\ell(y, \hat{y}) = |y - \hat{y}|$.

Binary Classification: **zero/one loss** $\ell(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{otherwise} \end{cases}$

This notation means that the loss is zero if the prediction is correct and is one otherwise.

Multiclass Classification: also zero/one loss.

? Why might it be a bad idea to use zero/one loss to measure performance for a regression problem?

Note that the loss function is something that *you* must decide on based on the goals of learning.

Now that we have defined our loss function, we need to consider where the data (training *and* test) comes from. The model that we

MATH REVIEW | EXPECTATED VALUES

We write $\mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(y, f(\boldsymbol{x}))]$ for the expected loss. Expectation means "average." This is saying "if you drew a bunch of (x, y) pairs independently at random from $\mathcal{D}$, what would your *average* loss be?More formally, if $\mathcal{D}$ is a discrete probability distribution, then this expectation can be expanded as:

$$\mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}[\ell(y, f(\boldsymbol{x}))] = \sum_{(\boldsymbol{x},y)\in\mathcal{D}} [\mathcal{D}(\boldsymbol{x}, y)\ell(y, f(\boldsymbol{x}))] \tag{1.1}$$

This is *exactly* the weighted average loss over the all $(\boldsymbol{x}, y)$ pairs in $\mathcal{D}$, weighted by their probability, $\mathcal{D}(\boldsymbol{x}, y)$. If D is a *finite discrete distribution*, for instance defined by a finite data set $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)$ that puts equal weight on each example (probability $1/N$), then we get:

$$\mathbb{E}_{(\boldsymbol{x},y)\sim D}[\ell(y, f(\boldsymbol{x}))] = \sum_{(\boldsymbol{x},y)\in D} [D(\boldsymbol{x}, y)\ell(y, f(\boldsymbol{x}))] \quad \text{definition of expectation} \tag{1.2}$$

$$= \sum_{n=1}^{N} [D(\boldsymbol{x}_n, y_n)\ell(y_n, f(\boldsymbol{x}_n))] \quad D \text{ is discrete and finite} \tag{1.3}$$

$$= \sum_{n=1}^{N} [\frac{1}{N}\ell(y_n, f(\boldsymbol{x}_n))] \quad \text{definition of } D \tag{1.4}$$

$$= \frac{1}{N}\sum_{n=1}^{N} [\ell(y_n, f(\boldsymbol{x}_n))] \quad \text{rearranging terms} \tag{1.5}$$

Which is exactly the average loss on that dataset.

The most important thing to remember is that there are two equivalent ways to think about expections: (1) The expectation of some function g is the *weighted average value of* g, where the weights are given by the underlying probability distribution. (2) The expectation of some function g is your *best guess of the value of* g if you were to draw a single item from the underlying probability distribution.

Figure 1.4:

will use is the *probabilistic* model of learning. Namely, there is a probability distribution $\mathcal{D}$ over input/output pairs. This is often called the **data generating distribution**. If we write $\boldsymbol{x}$ for the input (the user/course pair) and y for the output (the rating), then $\mathcal{D}$ is a distribution over $(\boldsymbol{x}, y)$ pairs.

A useful way to think about $\mathcal{D}$ is that it gives *high probability* to reasonable $(\boldsymbol{x}, y)$ pairs, and *low probability* to unreasonable $(\boldsymbol{x}, y)$ pairs. A $(\boldsymbol{x}, y)$ pair can be unreasonable in two ways. First, $\boldsymbol{x}$ might be an unusual input. For example, a $\boldsymbol{x}$ related to an "Intro to Java" course might be highly probable; a $\boldsymbol{x}$ related to a "Geometric and Solid Modeling" course might be less probable. Second, y might be an unusual rating for the paired $\boldsymbol{x}$. For instance, if Alice were to take AI 100 times (without remembering that she took it before!), she would give the course a $+2$ almost every time. Perhaps some

semesters she might give a slightly lower score, but it would be unlikely to see $\boldsymbol{x}$ =Alice/AI paired with $y = -2$.

It is important to remember that we are not making *any* assumptions about what the distribution $\mathcal{D}$ looks like. (For instance, we're not assuming it looks like a Gaussian or some other, common distribution.) We are also not assuming that we know what $\mathcal{D}$ is. In fact, if you know *a priori* what your data generating distribution is, your learning problem becomes significantly easier. Perhaps the hardest thing about machine learning is that we *don't* know what $\mathcal{D}$ is: all we get is a random sample from it. This random sample is our training data.

Our learning problem, then, is defined by two quantities:

1. The loss function ℓ, which captures our notion of what is *important* to learn.
2. The data generating distribution $\mathcal{D}$, which defines what sort of data we expect to see.

? Consider the following prediction task. Given a paragraph written about a course, we have to predict whether the paragraph is a *positive* or *negative* review of the course. (This is the sentiment analysis problem.) What is a reasonable loss function? How would you define the data generating distribution?

We are given access to **training data**, which is a random sample of input/output pairs drawn from $\mathcal{D}$. Based on this training data, we need to **induce** a function f that maps new inputs $\hat{\boldsymbol{x}}$ to corresponding prediction $\hat{y}$. The key property that f should obey is that it should do well (as measured by ℓ) on future examples that are *also* drawn from $\mathcal{D}$. Formally, it's **expected loss** ϵ over $\mathcal{D}$ with repsect to ℓ should be as small as possible:

$$\epsilon \triangleq \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}\big[\ell(y, f(\boldsymbol{x}))\big] = \sum_{(\boldsymbol{x},y)} \mathcal{D}(\boldsymbol{x}, y)\ell(y, f(\boldsymbol{x})) \tag{1.6}$$

The difficulty in minimizing our **expected loss** from Eq (1.6) is that we *don't know what $\mathcal{D}$ is!* All we have access to is some training data sampled from it! Suppose that we denote our training data set by D. The training data consists of N-many input/output pairs, $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_N, y_N)$. Given a learned function f, we can compute our **training error**, $\hat{\epsilon}$:

$$\hat{\epsilon} \triangleq \frac{1}{N}\sum_{n=1}^{N} \ell(y_n, f(\boldsymbol{x}_n)) \tag{1.7}$$

That is, our training error is simply our *average error* over the training data.

? Verify by calculation that we can write our training error as $\mathbb{E}_{(\boldsymbol{x},y)\sim D}\big[\ell(y, f(\boldsymbol{x}))\big]$, by thinking of D as a distribution that places probability $1/N$ to each example in D and probability 0 on everything else.

Of course, we can drive $\hat{\epsilon}$ to zero by simply memorizing our training data. But as Alice might find in memorizing past exams, this might not generalize well to a new exam!

This is the fundamental difficulty in machine learning: the thing we have access to is our training error, $\hat{\epsilon}$. But the thing we care about

minimizing is our expected error ϵ. In order to get the expected error down, our learned function needs to **generalize** beyond the training data to some future data that it might not have seen yet!

So, putting it all together, we get a formal definition of induction machine learning: **Given (i) a loss function ℓ and (ii) a sample D from some unknown distribution $\mathcal{D}$, you must compute a function f that has low expected error ϵ over $\mathcal{D}$ with respect to ℓ.**

A very important comment is that we should *never* expect a machine learning algorithm to generalize beyond the data distribution it has seen at training time. In a famous—if posssibly apocryphal—example from the 1970s, the US Government wanted to train a classifier to distinguish between US tanks and Russian tanks. They collected a training and test set, and managed to build a classifier with nearly 100% accuracy on that data. But when this classifier was run in the "real world", it failed miserably. It had not, in fact, learned to distinguish between US tanks and Russian tanks, but rather just between clear photos and blurry photos. In this case, there was a *bias* in the training data (due to how the training data was collected) that caused the learning algorithm to learn something other than what we were hoping for. We will return to this issue in Chapter 6; for now, simply remember that the distribution D for training data *must match* the distribution D for the test data.

1.5 Chapter Summary and Outlook

At this point, you should be able to use decision trees to do machine learning. Someone will give you data. You'll split it into training, development and test portions. Using the training and development data, you'll find a good value for maximum depth that trades off between underfitting and overfitting. You'll then run the resulting decision tree model on the test data to get an estimate of how well you are likely to do in the future.

You might think: why should I read the rest of this book? Aside from the fact that machine learning is just an awesome fun field to learn about, there's a lot left to cover. In the next two chapters, you'll learn about two models that have very different inductive biases than decision trees. You'll also get to see a very useful way of thinking about learning: the geometric view of data. This will guide much of what follows. After that, you'll learn how to solve problems more complicated that simple binary classification. (Machine learning people like binary classification a lot because it's one of the simplest non-trivial problems that we can work on.) After that, things will diverge: you'll learn about ways to think about learning as a formal optimization problem, ways to speed up learning, ways to learn

without labeled data (or with very little labeled data) and all sorts of other fun topics.

But throughout, we will focus on the view of machine learning that you've seen here. You select a model (and its associated inductive biases). You use data to find parameters of that model that work well on the training data. You use development data to avoid underfitting and overfitting. And you use test data (which you'll never look at or touch, right?) to estimate future model performance. Then you conquer the world.

1.6 Further Reading

In our discussion of decision trees, we used *misclassification rate* for selecting features. While simple and intuitive, misclassification rate has problems. There has been a significant amount of work that considers more advanced splitting criteria; the most popular is ID3, based on the mutual information quantity from information theory. We have also only considered a very simple mechanism for controlling inductive bias: limiting the depth of the decision tree. Again, there are more advanced "tree pruning" techniques that typically operate by growing deep trees and then pruning back some of the branches. These approaches have the advantage that different branches can have different depths, accounting for the fact that the amount of data that gets passed down each branch might differ dramatically[3].

[3] Quinlan 1986