Modeling Langauge

(Course notes for 6.461: Natural Language Processing)

Last updated: September 10, 2025

Chapter 1

Introduction

1.1 A guessing game

1.1.1 Gambling with Andrey

Imagine that, one day, your friend (let's call him Andrey) asks you to play a strange new gambling game. In each round of the game, Andrey begins by paying you a dollar. Then another friend (call her Karen) selects a random page from a random book in her extensive library extensive library. Karen reveals to both of you the first few words of the page, perhaps:

> *after the Massachusetts Instute of*

Next, Andrey guesses a possible next word, perhaps:

> *Fire*

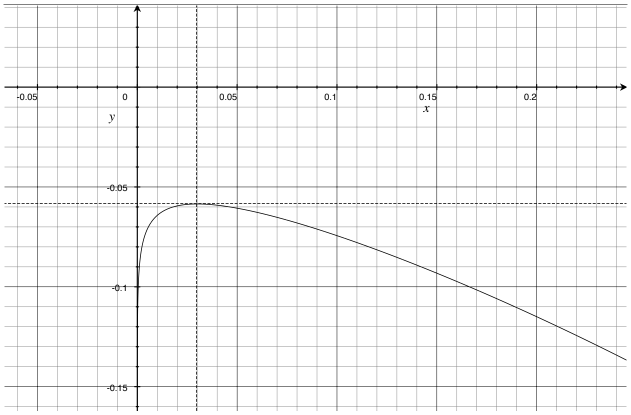
Finally, you must guess a number between 0 and 1. Call this number q. If Andrey's guess is correct (the next word in the book is *Fire*), you have to pay him back $-\log q$ dollars; otherwise you have to pay him back $-\log(1-q)$ dollars.

How should you set the value of q? If you set it to 0.1, do you think you'll make money? What if you set it to 0.000001? (Does learning that there is a real organization called the "Massachusetts Institute of Fire Department Instructors" change your answer to either of the questions above?)

We can begin to answer this question if we know something about the *probability* of words appearing next to each other in Karen's library. In particular, your expected return from the bet is:

$$p(\text{next word is } \textit{Fire}) \cdot \log(q) \text{ dollars} + (1 - p(\text{next word is } \textit{Fire})) \cdot \log(1-q) \text{ dollars}$$

If the true value of $p(\text{next word is } \textit{Fire})$ is 0.03, then your expected winnings (as a function of x) look like this graph



This graph is maximized at $x = 0.03$. In fact, we'll show later that it's *always* best to let x be the probability that the next word appears. As a corollary, if you're playing this game rationally, Andrey can infer—from the value you give for q—what you think the probability of *Fire* is in this context.

1.1.2 Playing to win

So if you want to win (or at least minimize your losses) in this game, what do you need to know? Consider the following snippets from Karen's library, and two possible guesses from Andrey:

> *(1) The beaver standing on top of the pile of books loudly [sing, sings]*
>
> *(2) Macondo is a fictional city perhaps located in the South American nation of [Colombia, Panama]*
>
> *(3) Instructor: "Please write a five-paragraph essay on the causes of the First World War." Student: "While the immediate cause of the war was the [assassination, abdication]*
>
> *(4) Determine all positive integers n for which there exist positive integers a, b, and c satisfying* $2a^n + 3b^n = 4c^n$*. Answer: [1, any]*
>
> *(5) After perusing the codebase, I was able to attribute the vulnerability to the following buffer overrun:*
> `char username[8] = ""; bool is_admin = 0; [strcpy, strlcpy]`

As we can see from these examples, knowing which bets to take in the guessing game can, in principle, require many different kinds of knowledge: linguistic knowledge (does the verb after *loudly* more likely to be *sing* or *sings*?), world knowledge (what are the names of the different countries in South America?), the ability to follow instructions, solve math problems, and even reason about non-linguistic text like poorly written code. And in all cases, the *probabilistic* aspect of this knowledge is essential—we need to know not just that *sings* is more plausible than *sings*, but the rate at which people confuse them, and how much more likely *Colombia* is than *Suriname* as the home of a city named *Macondo*.

So again—if we could simply build a system that was good at playing the guessing game, we could use it for all kinds of things: identifying grammatical mistakes, answering factual questions about the world, solving hard math problems and helping writing code. And building such a system ultimately just requires modeling the probability that one word will follow another in any of the books in Karen's library.

1.1.3 Playing to write

After a few rounds, you discover that you're quite good at this game—when you play with Andrey, you usually come out ahead. But then the procedure by which Karen selects books from her library seems to change.

Initially, Andrey says, the context is no context at all—he wants you to guess the very first word in a document. He proposes many next words, gets your probabilities for all of them, and eventually (after paying you lots of money) determines that you think the most likely first word in this empty context is *The*.

Entirely coincidentally, he says, the next context context for the game consists of the single word *The*, and he again makes a very large series of bets from which he eventually concludes that the word you consider most probable is *big*. And now, even more surprisingly, the next context that comes out of the library consists of the words *The big*. At this point you're beginning to catch on, and keep playing along, eventually predicting that a period will come after *The big cat sat on the mat and looked around*. Thus, Andrey has used the guessing game to make you *generate* a sentence that you think is particularly probable.

1.1.4 Beyond the guessing game

From the dawn of human civilization, right up until just a few years ago, writing essays (example 3 above) and solving math problems (example 4) were considered almost quintesentially human activities—no other form of intelligence, whether produced by evolution or human engineering, could even begin to perform these tasks. Even grammatical error correction, basic factual recall, and static analysis of computer programs represented major open research challenges.

Today, we live in a world where anyone with an internet connection can access highly capable automated systems that can do all these things, and more. For readers who have grown up in this world, it may be difficult to understand just how surprising, and just how rapid, these changes were from the perspective of the history of artificial intelligence research. Today's AI systems are certainly not perfect—they are remarkably un-human-like, and prone to surprising mistakes; their failures are often difficult to predict. But it's undeniable that they're capable of performing wide variety of tasks, deploying a greater breadth (and sometimes depth) of knowledge than many expert humans possess.

And perhaps most surprisingly of all, the root of all of these capabilities is one very simple idea: building automated systems that can predict the probability that one word will follow another, by learning to play the guessing game.

1.2 Modeling language

Probabilistic models of text—the kind needed to evaluate $p(\text{Andrey is correct})$, and ultimately to determine which bets to take—are called **language models**. In a broad sense, this course is about nothing more than figuring out how to train a good language model: that is, a good automated player of the guessing game. By the end of the course, you'll understand how state-of-the-art language models are trained, how they're implemented, and a little about how they're deployed to the millions of people who use them every day.

As we'll see, building a language model that *works*—that can reliably solve problems like the ones above—touches on almost every question of interest to computer scientists, from the lowest-level questions about computer architecture to the highest-level questions about human–computer interaction. A single semester isn't long enough to cover, or even touch on, most of these questions, but there are a few topics that will be recurring themes throughout the course:

1.2.1 Data

When we first saw the guessing game, the "input text" used in each round of the game came from Karen's library. But what is the equivalent of this library in the real world—what's the actual distribution over texts that we're supposed to be modeling?

Language models are called *language* models for a reason—their most impressive abilities, as well as their usefulness, are a direct result of the fact that they're not just models of any distribution over strings, but of human language.

Language itself is one of the most remarkable features of the human experience. It's the result of tens (or perhaps hundreds) of thousands of years of cultural exchange, and rests atop tens of millions of years of evolution that provided both the cognitive and vocal apparatus that produce birdsong, prairie dog calls and dolphin whistles.[1] Today, more than seven thousand different languages are spoken throughout the world. The vast majority of these languages (about 95%) are spoken by fewer than a million people; and many (about 25%) are spoken by fewer than a thousand![2] Many of these languages have no written form; even those that do may have little readily accessible text in digital form.

This is important because the real-world version of "Karen's library" is, to a first approximation, the internet—all (large-scale) language model training today begins by estimating the distribution of words in pre-existing and pre-digitized texts. What these LMs learn is not, as a result, a snapshot of "human language" as a whole, but rather language as used and shared by a relatively small fraction of the world's most connected people. At several points in this course, we'll return to the mechanisms underlying this data generation process, and their consequences for the language models that we train.

To further complicate matters, the real-world language models we interact with are not actually models of text-in-general, but rather *tools* that have been designed to serve specific functions (to engage in chat conversations about specific topics; to generate code using certain langauges and libraries). Thus, the real distribution over texts that they imitate is, to a large extent, *designed*, and the design of these distributions itself raises a number of interesting questions in machine learning and human-computer interaction.

1.2.2 Models

So far we have been talking abstractly about using an "automated predictor" to compute the probability of a word given a context. What do these predictors look like in practice?

Throughout this course, we'll focus on predictors that use machine learning techniques to estimate the probability of words in context from large datasets. We'll see several such predictors, ranging from very simple ones (that just count the number of times different words co-occur) to very sophisticated ones (based on artificial neural networks). Here our goal will be to introduce you to the standard kit of parts—mechanisms for representing words and phrases, for combining information from these representations, and so on—that underlie almost all of today's language models, and which are likely to serve as the building blocks of future models as well. In addition, we'll become familiar with suite of important **algorithms** that allow us to fit neural networks to these datasets, and to learn from other forms of interaction. Finally, we'll briefly explore problems related to **evaluation**—how to figure out what models have learned, and how effectively they are generalizing to new inputs.

The largest language models are very expensive to train (whether you measure in electricity, hardware, or dollars), and very expensive to deploy. They are also complicated to update or modify when they make mistakes or go out of date. So in addition to standard machine learning considerations about accuracy and generalization, some of our discussion of modeling will focus on ways of designing models so that they can be trained, deployed, and updated efficiently.

1.2.3 Applications

As we have noted, most interactions with real-world LMs don't involve asking them to predict the next word in natural text—instead, we use them as part of larger software applications, in which they are intended to serve specific functions and address specific use-cases.

[1] Bolhuis et al. Twitter evolution: converging mechanisms in birdsong and human speech.

[2] https://www.ethnologue.com/

But unlike ordinary pieces of software, neural-network-based language models are by their nature unpredictable. We can't (yet) anticipate what mistakes they will make, or explain to users or developers why those mistakes occurred, and we certainly can't write unit tests or conduct formal verification that will convince us that errors will never occur. A growing body of research on **interpretability** techniques for neural networks is beginning to help us understand better how these models work "under the hood"—how their predictions derive from human-understandable features, circuits, and learned algorithms. While this field is still in its infancy, interpretability techniques might offer first steps toward real bottom-up understanding (and verification) of language model behaviors. But for now, the process of incorporating language models into larger software systems forces us to think about ways we can use techniques from outside machine learning to build **safeguards and fallback mechanisms** that ensure that these systems behave reliably even when language models themselves don't.

One setting in which we must think most carefully about questions of reliability is in language models' **interactions with people**. As they become increasingly integrated into search engines, productivity software, and social networks, it is crucial to avoid situations offer bad advice, amplify misinformation, and adversely affect users' mental health. Answering these questions often involves identifying situations in which we should avoid using language models altogether. And even when they are useful or critical, the impact of language models on human beliefs and behavior (at the individual and societal level) raises important philosophical and technical questions—how do we ensure that models accurately express uncertainty, derive information from trustworthy sources, and refuse to perform tasks we consider harmful? Is this even the right set of desiderata?

Finally, the last few years have seen rapid changes in the **legal and policy** frameworks under which language models (and other AI systsems) operate. For example, what kinds of rights do the creators of models' training data have over models' eventual output? If we use a book as part of a model's training set, then give the model away, is it as if we had given away a copy of the book (which is generally prohibited by copyright laws)? Or is it as if a human creater had read the book, been inspired, and used that inspiration to create a new work (which is generally permitted)? When a mistake from a language model causes harm (say if a user is injured by bad medical advice generated from a language model), does the fault lie with the user, the organization that distributed the model, or the organization that trained it?

These questions become more pressing as LMs become more capable, and their internal computations become harder to understand. But before we tackle them, let's begin to develop some intuition by thinking about the simplest language models we can imagine.

Part I

Foundations of language modeling

Chapter 2

Preliminaries

When we described the guessing game, we phrased it in terms of guessing the next word, but saw that the basic task of assigning a probability to a one-word continuation in a context could be iterated to perform much more complicated tasks, like assessing the probability of entire sentences, or even generating whole stories from scratch. Let's formalize this a little bit.

2.1 Strings

As we know from our intro programming class, the basic computational representation of (written) language data, and other text like code itself, is the **string**. Concretely, a string is just a sequence of symbols strung together. In programming contexts we're used to thinking of them as sequences of characters or bytes (the string `"Hello␣world!"` corresponds to the sequence `[H, e, l, l, o, ␣, w, o, r, l, d, !]`. In this class it will often be useful to work at a higher level of abstraction—for example, by thinking of strings as sequences of words (`[Hello, ␣world!]`) or sequences of any other (larger or smaller) "atomic" unit of text (`[Hello, ␣wor, ld, !]`). Let's call these units "tokens". Formally,

Definition 1. *Let Σ denote a **vocabulary** consisting of a finite set of **tokens** (e.g. letters, words, or word pieces. A **string** $\mathbf{x}$ is a finite sequence of tokens. We will denote by $\Sigma^\star$ the set of all strings.*

In the example from the previous paragraphs, if tokens are individual characters, we might have $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \ldots\}$, and $\Sigma^\star = \{\texttt{a}, \texttt{aa}, \texttt{aba}, \ldots, \texttt{hello}, \ldots\}$.

2.2 Distributions

Almost all of the objects we call **language models** (or **LMs**) are designed to model a **next-token** distribution: given a sequence of tokens as input, they assign a probability to all possible next tokens. But in fact the term "language model" has beecome a bit overloaded over the years, and is often used to refer to models that predict other distributions (like the probability of some token given the tokens before *and after*—as though we have hidden a single word from a document and want to guess what it is given all the surrounding context). So when we're being precise, we'll use the following definition and terminology:

Definition 2. *An **autoregressive language model** is a conditional distribution $p_{\mathrm{LM}}(x_i \mid \mathbf{x}_{1:i-1} = x_1, x_2, \ldots, x_{i-1})$ that assigns probabilities to possible **next tokens** x_{i+1} given **context strings** $\mathbf{x}_{1:i}$.*

In the running example, an autoregressive LM might assign a probability to the token `o` given the context `Hello w`.

To avoid having to write all these subscripts (and in contexts where we don't need to refer to them explicitly), we'll also write this distribution as $p_{\mathrm{LM}}(y \mid \mathbf{x})$ (the probability of a next token y given a context $\mathbf{x}$). Like the conditional distributions you saw in introductory probability, an LM must be well-defined in all input contexts, must to assign a probability to every possible next token, and these probabilities must sum to one:

$$\sum_{x \in \Sigma} p_{\mathrm{LM}}(y \mid \mathbf{x}) = 1 \;\; \forall \mathbf{x} \tag{2.1}$$

Here we have defined autoregressive LMs as distributions over next tokens. But it is also possible to use these LMs to place distributions over *strings*. Let us first define a special token $\textsc{eos}$(for **E**nd **of** **S**tring), and assume that the distribution $p(y \mid \mathbf{x})$ defined by our LM also assigns some probability to $\textsc{eos}$. Then we will define the probability of a string $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ as:

$$p_{\mathrm{LM}}(\mathbf{x}) = p_{\mathrm{LM}}(x_1) \cdot p_{\mathrm{LM}}(x_2 \mid x_1) \ldots p_{\mathrm{LM}}(x_n \mid x_{1:n-1}) \cdot p_{\mathrm{LM}}(\textsc{eos} \mid x_1, \ldots, x_n) \tag{2.2}$$

We'll see at the end of this section why we need the final $\textsc{eos}$to get a proper distribution over strings, and some other subtleties that arise when turning a distribution over next tokens into a distribution over strings.

2.3 Tokens

We began this chapter by defining LMs as distributions over tokens, and noting that there were many ways of tokenizing a string. Which one should we choose, and how do we actually define a tokenization procedure?

2.3.1 Byte-level tokenization

Let's thing about the string `Hello␣world!` again. Before we can even imagine using it as training data or input for a language model, we need some way of representing it in computer memory. In most programming languages, this is as a series of **bytes**. In most encoding schemes for English and other languages with Latin alphabets, every **character** gets one byte, so we would represent `Hello␣world!` as the sequence `[H, e, l, l, o, ␣, w, o, r, l, d]`.[1] We may, in general, reasonably assume that these encoding schemes (and the alphabet used to write English!) will remain fixed during the lifetime of any language model we train. So, when implementing an LM as a next-token prediction system, we can always tokenize strings the way ordinary software does—as byte sequences. In this case our vocabulary Σ simply consists of this finite character set:

$$\Sigma = \{\mathsf{chr}(i) : i = 0..2^8 - 1\} \tag{2.3}$$

But what are the disadvantages of this approach? Think back to the end of Chapter 1, where we used the guessing game to generate a story word-by-word. A typical paragraph might have 1000 or 2000 characters—that's a lot of guesses! It will take a long time to generate text if we have to produce them one-by-one. And more importantly, most of those guesses are *boring*—once we've generated `Hello␣worl`, every single word in the dictionary has a `d` as its next character. Training a big expensive neural network to perform this computation, when we can get the same answer directly from the dictionary, seems quite wasteful.

[1] Which in turn is represented in memory as the sequence of bytes `[1001000, 1100101, 1101100, 1101100, 1101111, 100000, 1110111, 1101111, 1110010, 1101100, 1100100, 100001]`.

2.3.2 Word-level tokenization

In some sense, a dictionary (of the kind you use for looking up the meanings of words) encodes a different, human-designed tokenization scheme—a list of character sequences that are very likely to appear in human-generated text. Can we simply use *words* as tokens, and build LMs that generate text one word at a time? (This is how we posed the original version of the guessing game.) We might wish to handle punctuation and spaces separately, representing our original string as `[Hello, ␣, world, !]`.

This solves our length problem: to generate a typical paragraph, we've gone from thousands to hundreds of characters. But it also introduces several new, thornier ones. What qualifies a word for inclusion in our vocabulary? Is it any entry appearing in a previously published dictionary? (The dictionary on my laptop doesn't contain the word *autoregressive*.) Is it any sequence of characters separated by whitespace in any pre-existing collection of text documents?

We might some day wish for a system that can reason about the multiplication problem *What's two times 28668714930225?* But according to Google, this number does not appear anywhere on the internet, so it won't be in the vocabulary for any tokenizer we try to train. In many languages, even defining a word is complicated—in Chinese, words (*ci*) that span multiple characters are not delimited by spaces; in Turkish, expressions that would be entire sentences of English are formed by adding suffixes to nouns and verbs (*I can slow down* is *yavaşlayabilirim*). Thus, if we pick the wrong set of "words" ahead of time, we may find that there is text that is *impossible* to tokenize, when we try to present it to an LM as input, or generate it as output.

2.3.3 Optimal tokenization

In the course of trying to construct byte-level and word-level tokenizers, we've encountered the two main difficulties that arise when trying to build a tokenization scheme: we want a way of representing text that is *compact* (strings aren't any longer than they need to be) and *expressive* (we can represent *any* string, even those we didn't anticipate when we were designing the tokenizer). If we look beyond byte-level and word-level tokenization, is there a way to achieve expressivity and compactness?

Let's now be a little more formal about this—and about tokenizers more general.

Definition 3. *A **tokenizer** is a tuple* (Σ_0, Σ, f)*, where*

1. Σ_0 *is a (finite)* ***base vocabulary*** *describing the minimal units from which strings are built (*bytes / characters *in the examples above).*
2. Σ *is a (finite)* ***token vocabulary*** *that the tokenizer uses to represent outputs (we tried to fill this with* words *above).*
3. $f : \Sigma_0^\star \to \Sigma^\star$ *is a* ***tokenization function*** *that takes strings represented as sequences in the base vocabulary, and transforms them into tokens.*

Crucially, f must be a *total* function—it needs to be able to tokenize any string representable in the base vocabulary. We need both a vocabulary *and* a tokenization function to handle cases like $\Sigma = \texttt{He}, \texttt{llo}, \texttt{Hel}, \texttt{lo}, x = \texttt{Hello}$ for which the tokenization might be ambiguous.

With these definitions, we can now describe precisely what it would mean for a tokenization scheme to be both expressive and compact. Suppose we have a collection of strings $X = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}\}$, where each $\mathbf{x}^{(i)} \in \Sigma_0^\star$ that are representative of the distribution over strings we will eventually need to tokenize. Then we might try to define the *optimal* tokenizer X as the one that solves

$$\arg\min_{\Sigma, f} \sum_i |f(\mathbf{x}^{(i)})| \tag{2.4}$$

Here $|\mathbf{x}|$ denotes the length of the string $\mathbf{x}$. We have written the arg min over f and Σ to emphasize that we're optimizing *both* the set of tokens and the mapping from the base vocabulary to tokenized strings.

There's a problem with this definition—for any finite X, the optimal "tokenization" scheme simply assigns every $\mathbf{x}$ to a single token, so each $|\mathbf{x}_i| = 1$, but $|\Sigma| = |X|$ (and will get bigger and bigger as our vocabulary grows!) In practice, tokenizers need to satisfy a third requirement—that the size of the token vocabulary itself is not too large, so that models for next-token prediction are not too big to train, run, or store. One option is to fix a vocabulary size V ahead of time:

$$\arg\min_{\Sigma,f} \sum_i |f(\mathbf{x}^{(i)})| \qquad \text{such that } |\Sigma| = V \tag{2.5}$$

Another option is to minimize the *combined* size of the tokenized dataset and vocabulary:

$$\arg\min_{\Sigma,f} \big(\sum_i |f(\mathbf{x}^{(i)})|\big) + \lambda \cdot |\Sigma| \tag{2.6}$$

where λ controls the relative importance of short strings and a small vocabulary.

But alas! Both Eq. (2.5) and Eq. (2.6) are NP-complete problems, so we cannot expect to find optimal tokenization schemes for large datasets.

2.4 Byte pair encoding

Instead, almost all LMs that are trained today use an approximate, greedy algorithm to solve Eq. (2.5). That algorithm is called **byte pair encoding**. At a high level, it iteratively creates new tokens by iteratively finding the *pair* of tokens that, when merged, would immediately reduce the size of the dataset X by the largest possible amount.

For example, given the (single) input document `Hello␣mellow␣fellows!` and a base vocabulary Σ_0 consisting of all characters, we first represent the input as a list of characters:

`[H, e, l, l, o, ␣, m, e, l, l, o, w, ␣, f, e, l, l, o, w, s, !]`

and initialize $\Sigma = \Sigma_0$. We then count the number of times each *pair* of adjacent characters occurs:

Token pair	Count	Token pair	Count
`H, e`	1	`e, l`	3
`l, l`	3	`l, o`	3
`o, ␣`	1	`␣, y`	1
`m, e`	1	`o, w`	2
`w, ␣`	1	`␣, f`	1
`w, s`	1	`s, !`	1

Here, there are three most frequent pairs: `el`, `ll`, and `lo`. We create a *new token* from the first of these pairs, add this token to our vocabulary ($\Sigma = \Sigma_0 \cup \{$`el`$\}$), and re-write the data in terms of the new token:

`[H, el, l, o, ␣, m, el, l, o, w, ␣, f, el, l, o, w, s, !]`

Now, we simply repeat this process until the target vocabulary size is reached!

$\Sigma = \Sigma_0 \cup \{$`el,ell`$\}$
$X =$ `[H, ell, o, ␣, m, ell, o, w, ␣, f, ell, o, w, s, !]`

$\Sigma = \Sigma_0 \cup \{$`el,ell,ello`$\}$
$X =$ `[H, ello, ␣, m, ello, w, ␣, f, ello, w, s, !]`

and so on.

More formally, we can write the BPE algorithm as:

Algorithm 1 Dictionary learning for byte pair encoding

```
Input: Base vocabulary $\Sigma_0$, dataset $X$, target vocab size $V$
$\Sigma \leftarrow \Sigma_0$
$M \leftarrow []$
while $|\Sigma| < V$ do
    $(a^\star, b^\star) = \arg\max_{a,b \in \Sigma} \sum_{\mathbf{x} \in X} \sum_{i=1}^{|\mathbf{x}|-1} \mathbb{1}[x_i = a \wedge x_{i+1} = b]$
    $\texttt{new\_tok} = a^\star b^\star$
    $\Sigma \leftarrow \Sigma \cup \{\texttt{new\_tok}\}$
    $M \leftarrow M + [\texttt{new\_tok}, a^\star, b^\star]$
    for $\mathbf{x} \in X$ do
        $\mathbf{x} \leftarrow \texttt{replace}([a^\star, b^\star], [\texttt{new\_tok}], \mathbf{x})$
    end for
end while
return $\Sigma, M$
```

Here $\mathbb{1}$ denotes the **indicator** function that evalutes to 1 if its argument is true and 0 otherwise; $\texttt{replace}(a, b, c)$ replaces the sequence a with b wherever it occurs in c.

Note that so far we have described a way to compute the vocabulary Σ, but not the tokenization function f, which tokenizes *new* strings (not in X). Here, a simple option is to remember the sequence of merges that produced new tokens in Algorithm 1 (stored in the list M) and tokenize new strings by applying the same set of merges in order.

Algorithm 2 Tokenization for byte pair encoding

```
Input: Merge history $M$, string $\mathbf{x}$
for $\texttt{tok}, a, b \in \Sigma$ do
    $\mathbf{x} \leftarrow \texttt{replace}([a, b], \texttt{tok}, \mathbf{x})$
end for
return $\mathbf{x}$
```

Tokenization remains an active area of research, and other strategies are sometimes used. At the end of the course, we'll see *long-context* LMs specifically designed to process very long input sequences efficiently; in the future, these models may even enable use of byte-level tokenization by default. But in any case, we will assume in the remainder of this course that we have access to *some* pre-existing tokenization scheme (embodied by Σ_0, Σ, and f). With this in hand, we can begin to ask: how do we *predict* the next token?

Chapter 3

Count-based models

3.1 Unigram models

Now that we have a way of representing strings, let's start trying to assign probabilities to them—giving us our first language model.

Recall that, in general, a language model is a distribution $p(x_{i+1} \mid \mathbf{x}_{1:i})$ that places a distribution over next tokens x_i given a context $\mathbf{x}_{1:i}$. In other words, a language model specifies a complete conditional distribution over tokens for *every* context that we might encounter. In previous classes, you've probably written down conditional distributions like this as tables, but here we're not going to be able to write down a table of conditional probabilities for every context we might encounter—after all, strings can be arbitrarily long, so there are infinitely many contexts we might encounter! And as we'll see soon, we'd like our LMs to be able to share information and re-use computation between closely related contexts. So every language model we see in this class is going to make some **simplifying assumption** about the process by which contexts are turned into next words. Different simplifying assumptions will give us different language models that make use of data and computational resources in different ways.

So let's begin our language modeling journey with the simplest LM we can possibly imagine writing down—one that ignores its context altogether, and always predicts the *same* distribution over tokens. In other words,

$$p_{\mathrm{LM}}(x_i \mid \mathbf{x}_{1:i-1}) \approx p_{\mathrm{LM}}(x_{i+1}) \, . \tag{3.1}$$

A model that makes the assumption in Eq. (3.1) is called a **unigram** model (because it generates text by computing the probabilities of single tokens in isolation). Note that unigram models are not totally trivial—to be reliable, they still need to capture some information about natural text, like the fact that *orange* is a more likely next token than *prestidigitation* in a randomly chosen context. But they are still so simple that we *can* write them down as finite-sized tables—with one number for every word in the vocabulary.

In Chapter 2, we saw that tokenizers output an ordered list of tokens (`[a, b, c, . . . , el, ell, . . . ]`). Since a unigram model just needs to assign a probability to each of these tokens, we can represent it as a vector of real-valued **parameters**,

$$\theta = [\theta_{\mathtt{a}}, \theta_{\mathtt{b}}, \ldots, \theta_{\mathtt{el}}, \theta_{\mathtt{ell}}, \ldots] \, , \tag{3.2}$$

subject to the constraint that $\sum_i \theta_i = 1$ (so that we have a proper probability distribution). Then, our final unigram model is simply defined by:

$$p_{\mathrm{LM}}(x_i \mid \mathbf{x}_{1:i-1}) \approx \theta_{x_i} \, . \tag{3.3}$$

Throughout the rest of these notes, we will sometimes write the LM's predictive distribution as $p_{\mathrm{LM}}(y \mid \mathbf{x}; \theta)$ when we need to make the dependence on θ explicit. But even when written like the above, LM predictions will always depend implicitly on these parameters.

3.2 Maximum likelihood estimation

How should we choose the parameters θ for our unigram model? Suppose we have access to some large **corpus** of human-generated text. We can even think of this text as one very long string $\mathbf{x}$ in which whatever books, web pages, etc. we have are simply concatenated together. One reasonable principle for designing a language model is that it should be "as close to" the empirical distribution of tokens in this corpus as possible—if people use a word frequently, our LM should assign it high probability, and if people use a word rarely, our LM should assign it low probability. (This is not the only principle for designing an LM! But it's a very important one, and one we'll stick with for most of the course.)

One way of formalizing this intuition—that an LM should be as close to the data as possible—is known as the **principle of maximum likelihood**. It says that, from all of the different θ we could choose, we should pick the one that *maximizes the probability that the language model assigns to the data*.

$$\theta^\star = \arg\max_\theta \; p_{\text{LM}}(\mathbf{x};\theta) \tag{3.4}$$

The quantity $p_{\text{LM}}(\mathbf{x};\theta) = \prod_i p_{\text{LM}}(x_i \mid \mathbf{x}_{1:i-1};\theta)$ is known as the **likelihood function**.[1]

For computational reasons, it's often easier to work with log-probabilities than probabilities (in fact, we'll see an example of this in a moment, when we try to solve for θ). And because log is monotonic, the θ that maximizes $p_{\text{LM}}(\mathbf{x})$ will also maximize $\log p_{\text{LM}}(\mathbf{x})$. So we more often write:

$$\begin{aligned}
\theta^\star &= \arg\max_\theta \; \log p_{\text{LM}}(\mathbf{x};\theta) && (3.5)\\
&= \arg\max_\theta \; \log\Big(\prod_i p_{\text{LM}}(x_i \mid \mathbf{x}_{1:i-1};\theta)\Big) && (3.6)\\
&= \arg\max_\theta \; \sum_i \log p_{\text{LM}}(x_i \mid \mathbf{x}_{1:i-1};\theta) && (3.7)\\
&= \arg\max_\theta \; \sum_i \log \theta_{x_i} && (3.8)
\end{aligned}$$

where θ_{x_i} denotes the entry in the vector θ associated with the token x_i (i.e. the ith token in $\mathbf{x}$). In this case we call $\theta^\star$ the **maximum likelihood estimate (MLE)**.

In this case we can simplify further. Suppose the string $\mathbf{x}$ is `[the, cat, in, the, hat]`. Then

$$\begin{aligned}
\sum_i \log\theta_{x_i} &= \log\theta_{\texttt{the}} + \log\theta_{\texttt{cat}} + \log\theta_{\texttt{in}} + \log\theta_{\texttt{the}} + \log\theta_{\texttt{hat}} && (3.9)\\
&= 2\cdot\log\theta_{\texttt{the}} + \log\theta_{\texttt{cat}} + \ldots && (3.10)
\end{aligned}$$

Because `the` occurs twice, we simply wind up with two copies of $\log\theta_{\texttt{the}}$ in the objective. More generally, if we rearrange terms associated with copies of the same word, we get:

$$\arg\max_\theta \; \sum_{x\in\Sigma} \#(x)\log\theta_x \tag{3.11}$$

where $\#(x)$ denotes the **number of times** the token x occurs in $\mathbf{x}$.

Given some specific corpus $\mathbf{x}$, how should we actually compute the maximum likelihood estimate? Importantly, this is a *constrained* optimization problem—because our unigram language model ultimately needs to be a probability distribution, we can't pick any old θ—its entries must sum to 1.

[1] Pedantically, it's the "likelihood of θ"—that is, the likelihood function is a function of θ—but you may often hear people refer to the "likelihood of $\mathbf{x}$" as the probability of $\mathbf{x}$ under some model.

Recall from your calculus class that we can solve optimization problems like this one using the method of **Lagrange multipliers**. We write down both our original optimization problem and a weighted constraint ($\lambda \sum_i \theta_i = 1$)

$$L(\theta, \lambda) = \sum_{x \in \Sigma} \#(x) \log \theta_x + \lambda \Big(1 - \sum_{x \in \Sigma} \theta_x\Big) \tag{3.12}$$

Next, we solve for both θ and λ by setting gradients equal to 0:

$$\frac{\partial}{\partial \theta_x} L(\theta, \lambda) = \frac{\#(x)}{\theta_x} - \lambda = 0 \tag{3.13}$$

$$\#(x) = \lambda \theta_x \tag{3.14}$$

$$\theta_x = \frac{\#(x)}{\lambda} \tag{3.15}$$

Adding these equations together across all x, we have:

$$\sum_x \#(x) = \sum_x \lambda \theta_x = \lambda \sum_x \theta_x = \lambda \tag{3.16}$$

Notice that $\lambda = \sum_{x \in \Sigma} \#(x)$ is simply the number of tokens in the data. Plugging this back into Eq. (3.14), we have:

$$\theta_x = \frac{\#(x)}{\sum_{x'} \#(x')} \tag{3.17}$$

(If we were being rigorous here, we would also need to prove that this is a maximum, and not a minimum or saddle point of the loss function.) So the value we assign to each θ_x is simply the number of times x appears in the data, divided by the total number of tokens in the data—the **empirical probability** of x. So this gives us a very simple procedure for estimating unigram models—for each symbol in the vocabulary, we just count how often that token shows up, and normalize these counts to obtain a probability distribution.

Text sampled from a unigram model looks like:[2]

To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have

Not very good! But this is what happens when we sample uniformly at random from the vocabulary (assuming word-level tokenization):

poor gentle twelve How heel husband dark or swallowed stay makes Montague Anselmo delight shoemaker April brother Nay Gives accustomd

so even a few parameters go a long way toward more natural looking text.

Aside: the guessing game revisited In our initial presentation of the guessing game, Andrey would give you a possible next word and a context, you assigned the word a probability q, and received a payout of $\log q$ if you were right and $\log(1 - q)$ if you were wrong. Notice that payout corresponds exactly to the maximum likelihood objective we wrote down in Eq. (3.7) (but now with your guess spread over all the tokens in the vocabulary, rather than just a binary choice).

3.3 Sparsity

Suppose there's some token in the vocabulary that *never* appears in the data $\mathbf{x}$. According to the rule above, we should assign this token zero probability. Is this the right thing to do?

[2] Example from Jurafsky and Martin, *Speech and Language Processing*, 3rd edition.

Unigram models aren't going to be useful for much, but analogous situations can arise even in more complex models. Suppose we're trying to use probabilities from an LM to decide which of two sentences is better to include in an article, where our choices are:

Cats are a member of the family Felidae

Cats is a member of the family Felidae

and suppose further that there's a token for *Felidae* but it was never observed. Both of these sentences get assigned a probability of 0, and we can't distinguish between them, even though one is clearly better than the other. What can we do?

A standard approach is *smoothing*—modify the parameter vector so that impossible (or very rare) events seem more probable, in a way that makes the model behave more robustly in response to unusual events. One very simple approach is what's called *Laplace smoothing*—in place of the MLE, we set

$$\theta_x = \frac{\lambda + \#(x)}{\lambda|\Sigma| + \sum_{x'} \#(x')} \tag{3.18}$$

In other words, we pretend that we've seen each token λ more times than we actually did (where λ can be any positive number, but is often chosen to be between 0 and 1). This actually has a nice probabilistic interpretation (it's the result of optimizing $\log p_{\text{LM}}(\mathbf{x}) + \log p(\theta)$, where $p(\theta)$ is parameterized by a Dirichlet distribution). And when count-based models were widely in use, a number of more sophisticated smoothing schemes were developed. But most of the models we'll see in this class use a different set of tools for handling rare events, and Laplace smoothing works quite well when we do want to use an n-gram model.

3.4 n-gram models

The simplifying assumption we made to construct unigram models was that the next-token distribution doesn't depend *at all* on the context. This indeed gives a simple model, but one that's not useful for much—we can't control it, or use it to generate coherent looking text.

But a relatively simple extension of these models works surprisingly well for some applications, especially in low-data regimes. Instead of assuming that model predictions don't depend on *any* previous tokens, we assume that they depend on only a *finite number n* (say two or three). That is:

$$p_{\text{LM}}(x_i \mid \mathbf{x}_{1:i-1}) \approx p_{\text{LM}}(x_i \mid \mathbf{x}_{i-(n-1):i-1}) \tag{3.19}$$

Such models are called "n-gram" models. Notice that an n-gram model has a *context* consisting of $n-1$ tokens and predicts the nth. Unigram models are the special case where $n = 1$ and the context is empty.

Unlike unigram models, for moderately large values of n (and moderate amounts of training data), these models produce text that is looks quite reasonable.[3]

For $n = 2$: *Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow.*

For $n = 3$: *Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done.*

For $n = 4$: *King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch.*

An important lesson here is that generating (at least locally) coherent text often requires only short context windows! And for a long time, maintaining local coherence was pretty much the only thing anyone needed language models for—in their earliest applications, LMs were primarily used to *rerank* outputs from other, more specialized models for summarization, translation, etc.

[3] Examples again from Jurafsky and Martin, *Speech and Language Processing*, 3rd edition.

As in the special case of unigram models, it's very easy to perform parameter estimation for *n*-gram models—try this as an exercise. And while we saw that data sparsity could occasionally be an issue with unigram models, it's much greater in *n*-gram models—one consequence of the expressivity and flexibility of language is that most reasonable-looking sequences of four or five words have never appeared in any document, and get assigned a probability of 0 by *n*-gram models without appropriate smoothing. For larger values of *n*, there are a number of tricks that involve *backing off* to shorter contexts, or using more sophisticated smoothing schemes.

As we saw in Chapter 1, we can also summarization and translation as language modeling problems with appropriately structured contexts—for summarization, contexts like like *Here's an article: [article text] Here's a summary:*; for translation, contexts like: *Here's an English sentence: [sentence]. Here's the French translation:*. But trying to do this with n-gram models quickly runs into a few problems. First, to translate a sentence or write a summary, we need to look all the way back to the beginning of the input—not just the last few words. Any model that relies on a fixed-sized context won't be able o handle inputs longer than that context, and sparsity problems get worse and worse as contexts get longer. Second, once we've learned that *blue* and *cerulean* refer to similar colors, we should be able to infer that *a clear cerulean sky* is more probable than *a clear green sky*. But an *n*-gram model can't encode any notion of *similarity* between words—*n*-gram models must learn *independent* parameters for *every* usage of *blue* and *cerulean*. In the next few chapters, we'll see models that can overcome these limitations by learning shared representations of words, overcoming the sparsity problem and offering a first step toward longer contexts.

Chapter 4

Log-linear models

4.1 Parameterization

In Chapter 3, we saw that in an n-gram model, *every* sequence $x_1 \dots x_n$ of n tokens had to be associated with its own parameter $\theta_{x_1,\dots,x_n}$. But often, even when we're dealing with fixed-length contexts, it's possible to guess something about what word is likely to come next even if we consider only a subset of words in the context. For example, suppose we observe the snippet *executive* ■ ■ where each ■ denotes an unobserved word. Even if we can't guess exactly what's going to come next, we can infer that it's likely to have something to do with business or finance. Similarly, if we observe the snippet ■ ■ *very*, we can infer that the next word is likely to be an adjective or an adverb, even if we don't know what meaning it's likely to convey. Thus, even if we've never seen the snippet *executive compensation very* in our training data, we should be able to infer that the next word is probably a business-related adjective. Indeed, there are other features of documents that might not involve the identities of individual tokens at all (like the total number of words seen so far, or the average sentence length). n-gram models can't make use of any information of this kind—how can we build a model that can?

As before, we'll begin by writing down an approximation to the next-token distribution p_{LM}, and use data to estimate the parameters for this approximation. In n-gram models, we made the assumption that $p_{\text{LM}}(x_i \mid \mathbf{x}_{1:i-1}) \approx p_{\text{LM}}(x_i \mid \mathbf{x}_{i-(n-1):i-1})$. But if we want to be able to make use of many disparate (but potentially correlated) sources of information in the input, we'll need to make a different approximation.

To do so, let's start by defining a **feature function** $\phi(\mathbf{x})$ that produces a vector of all the different pieces of information we can extract from a context. For example, we might have:

$$\begin{aligned}
\phi_1(\mathbf{x}) &= \mathbb{1}[\text{last word in } \mathbf{x} \text{ is } \textit{very}] \\
\phi_2(\mathbf{x}) &= \mathbb{1}[\text{last word in } \mathbf{x} \text{ is } \textit{something}] \\
&\vdots \\
\phi_{2294}(\mathbf{x}) &= \mathbb{1}[\text{third-from-last word in } \mathbf{x} \text{ is } \textit{executive}] \\
&\vdots \\
\phi_{598251}(\mathbf{x}) &= \mathbb{1}[\text{\# of words in } \mathbf{x}]
\end{aligned}$$

Then $\phi(\textit{executive compensation very}) = [1, 0, \dots, 1, \dots, 3]$.

A feature function translates a context (a string) into a numerical representation (a vector). There are a few important things to note about this translation. First, because the output of ϕ needs to be a (real-valued) vector, many of our individual **features** ϕ_i are **indicator** features—they return 0 or 1 depending on whether the input has some attribute. Second, this means that to capture real text distributions, we'll need very large numbers of features—at least one for every token, and many more if we want to capture interactions between tokens (e.g. to test whether the last *two* tokens in the input are *before yesterday*). But this is fine—feature functions are typically defined in code, and we can construct large numbers of features automatically.

Having defined our feature function, we can now specify the approximation we'll use to train our language model. In a **log-linear model**, we have:

$$p_{\text{LM}}(y \mid \mathbf{x}) \approx \frac{\exp(\theta_y^\top \phi(\mathbf{x}))}{\sum_{y' \in \Sigma} \exp(\theta_{y'}^\top \phi(\mathbf{x}))} \tag{4.1}$$

Note that this gives us a proper probability distribution: each token y in the vocabulary gets a (positive) score $\exp(\theta_y^\top \phi(\mathbf{x}))$, and we normalize these scores to obtain next-word probabilities. In n-gram models, we had one real-valued parameter for each sequence of n tokens; in log-linear models, we have one parameter for every (input feature, output word) combination (and one parameter *vector* for each output word).

If you've seen multinomial (or "multiclass") logistic regression models in earlier classes, you'll notice that Eq. (4.1) is exactly such a model—we've reduced the problem of langauge modeling to the (extremely well studied) problem of logistic regression! These models are also called "log-linear" models because:

$$\log p_{\text{LM}}(y \mid \mathbf{x}) = \theta_y^\top \phi(\mathbf{x}) - Z(\mathbf{x}) \tag{4.2}$$

where

$$Z(\mathbf{x}) = \log \sum_{y \in \Sigma} \exp(\theta_{y'}^\top \phi(\mathbf{x})) \tag{4.3}$$

is an input-dependent normalizing constant. In other words, the (unnormalized) log probability the model assigns to each next token is a linear function of the input features.

4.2 Estimation

As in Section 3.2, if we have access to some training text $\mathbf{x}$, we can set the parameters θ via maximum-likelihood estimation: $\arg\max_\theta \sum_i \log p_{\text{LM}}(x_i \mid \mathbf{x}_{1:i-1})$. But unlike n-gram models, there's no closed-form expression that maximizes Eq. (4.1) in θ. Instead, we need to optimize it numerically. Today, most of the widely used optimizers for training machine learning models are based on **stochastic gradient descent** (SGD).[1] Intuitively, SGD iterates over training data, and for each datapoint nudges our current model parameters in a direction that increases the probability the model assigns to that datapoint. More specifically, we pick some initial parameters $\theta^{(0)}$ (typically by sampling them randomly from some pre-specified distribution). Then, for each (input, context) pair $(\mathbf{x}, y)$ in our data, we compute:

$$\theta^{(t+1)} = \theta^{(t)} + \frac{\partial}{\partial \theta} \log p_{\text{LM}}(y \mid \mathbf{x}) \, . \tag{4.4}$$

For complex language models, like the ones we'll see later in the course, this derivative can be messy to calculate, and is typically done with automated tools. But in the case of log-linear models, it actually has a very nice form. Recall that θ is a set of parameter vectors θ_y, one for each possible next token y. Let's consider what this update does to a single θ_a for some token a:

$$\frac{\partial}{\partial \theta_a} \log p_{\text{LM}}(y \mid \mathbf{x}) = \frac{\partial}{\partial \theta_a} \theta_y^\top \phi(\mathbf{x}) - \frac{\partial}{\partial \theta_a} Z(\theta) \tag{4.5}$$

[1] For historical reasons, optimization algorithms are typically framed in terms of *minimization* problems. Thus maximum likelihood estimation is sometimes presented as minimizing the negative log likeihood.

Let's start by computing the second term:

$$\begin{aligned}
\frac{\partial}{\partial \theta_a} Z(\mathbf{x}) &= \frac{\partial}{\partial \theta_a} \log \sum_{y' \in \Sigma} \exp(\theta_{y'}^\top \phi(\mathbf{x})) && (4.6)\\
&= \frac{\frac{\partial}{\partial \theta_a} \sum_{y' \in \Sigma} \exp(\theta_{y'}^\top \phi(\mathbf{x}))}{\sum_{y' \in \Sigma} \exp(\theta_{y'}^\top \phi(\mathbf{x}))} && (4.7)\\
&= \frac{\frac{\partial}{\partial \theta_a} \exp(\theta_a^\top \phi(\mathbf{x}))}{\sum_{y' \in \Sigma} \exp(\theta_{y'}^\top \phi(\mathbf{x}))} && (4.8)\\
&= \frac{\phi(\mathbf{x}) \exp(\theta_a^\top \phi(\mathbf{x}))}{\sum_{y' \in \Sigma} \exp(\theta_{y'}^\top \phi(\mathbf{x}))} && (4.9)\\
&= \phi(\mathbf{x}) p_{\mathrm{LM}}(y \mid \mathbf{x}) && (4.10)
\end{aligned}$$

Now let's compute the first term. Here there are two possibilities: if $a = y$ (i.e. the parameter we're computing the update for is the actual next token), then:

$$\frac{\partial}{\partial \theta_a} \theta_y^\top \phi(\mathbf{x}) = \phi(\mathbf{x}) \tag{4.11}$$

If $a \neq y$,

$$\frac{\partial}{\partial \theta_a} = 0 \tag{4.12}$$

Putting these together, we have:

$$\frac{\partial}{\partial \theta_a} \log p_{\mathrm{LM}}(y \mid \mathbf{x}) = \begin{cases} (1 - p_{\mathrm{LM}}(y \mid \mathbf{x})) \cdot \phi(\mathbf{x}) & \text{if } a = y \\ -p_{\mathrm{LM}}(y \mid \mathbf{x}) \cdot \phi(\mathbf{x}) & \text{if } a \neq y \end{cases} \tag{4.13}$$

Intuitively, if a is the next token, we push θ_a *toward* the feature representation of the context; if a is not the next token, we push θ_a *away* from the representation of the context.

Log-linear models have another nice property: suppose $\mathbf{x}$ is followed by a in q% of documents, and by some other token in $(1 - q)$% of documents. Moreover, suppose that $p_{\mathrm{LM}}(a \mid \mathbf{x}) = q$ (i.e. we've correctly estimated the probability that a follows $\mathbf{x}$).

$$\begin{aligned}
&\frac{\partial}{\partial \theta_a} q \log p_{\mathrm{LM}}(a \mid \mathbf{x}) + \frac{\partial}{\partial \theta_a} (1 - q) \log p_{\mathrm{LM}}(b \neq a \mid \mathbf{x}) && (4.14)\\
&\quad = q \cdot (1 - p_{\mathrm{LM}}(a \mid \mathbf{x})) \cdot \phi(\mathbf{x}) + (1 - q) \cdot (-p_{\mathrm{LM}}(a \mid \mathbf{x})) \cdot \phi(\mathbf{x}) && (4.15)\\
&\quad = q \cdot (1 - q) \cdot \phi(\mathbf{x}) + (1 - q) \cdot (-q) \cdot \phi(\mathbf{x}) && (4.16)\\
&\quad = 0 && (4.17)
\end{aligned}$$

So once we estimate these conditional distributions correctly, we stop updating θ!

4.3 Regularization

In Chapter 3, we saw that naive training of n-gram models could assign zero probability to possible events, or artificially high probability to very rare ones. Log-linear models are subject to the same problem: if we have a token y that never appears in the training data (and assuming ϕ outputs only positive values), gradient descent will try to assign *arbitrarily large negative values* to each entry in θ_y. (To see why this is, notice that the update in Eq. (4.13) is always in the direction of $-\phi(\mathbf{x})$, and is never 0 (because $\exp(\theta_y^\top \phi(\mathbf{x}))$ is always positive for finite feature and parameter values). By a similar argument, if there's a feature ϕ_i that is only associated with one output token y, $\theta_{y,i}$ is going to wind up with an arbitrarily large *positive* value, making it impossible to predict any other token if that feature appears in future training data.

In n-gram models, we fixed this problem with smoothing—adding values to n-gram counts even when not observed in training data. For log-linear models, we can use a generalization of this technique called **regularization**. In particular, instead of maximizing the log-likelihood as in Eq. (4.2), we'll optimize a "regularized" likelihood:

$$\sum_{\mathbf{x},y} \left[\theta_y^\top \phi(\mathbf{x}) - Z(\mathbf{x})\right] - \frac{\lambda}{2} \sum_y \|\theta_y\|^2 \tag{4.18}$$

where $\|\theta\|$ denotes the ℓ_2 norm $\theta^\top \theta$.

There are a few ways of interpreting this. First, in the form presented above, it's clear that this new objective says "all else equal, we should prefer parameter settings with small values of θ"—no setting weights to ∞ or $-\infty$!

Second, if we write out the gradient of the new term, it's just:

$$\frac{\partial}{\partial \theta_y} - \frac{\lambda}{2} \|\theta_y\|^2 = -\lambda \theta_y \tag{4.19}$$

In other words, every gradient step "shrinks" each θ_y by a factor $(1 - \lambda)$. For this reason, it is sometimes referred to as **weight decay**.

Third, suppose that instead of trying to maximize $\log p_{\mathrm{LM}}(\mathbf{x}) = \sum_i \log p_{\mathrm{LM}}(x_i \mid \mathbf{x}_{1:i-1})$, we had written our original objective as:

$$\arg\max_\theta \; \log \Big(p_{\mathrm{LM}}(\mathbf{x} \mid \theta) \cdot p(\theta)\Big) \tag{4.20}$$

then chose to define $p(\theta) = \mathcal{N}(0, \frac{1}{\lambda})$ (i.e. a standard normal distribution with variance $1/\lambda$). Writing this out explicitly, we have:

$$\arg\max_\theta \; \log \sum_i p_{\mathrm{LM}}(x_i \mid \mathbf{x}_{1:i-1}) + \log \Big(\frac{1}{\sqrt{2\pi\sigma^2}} \exp\big(-\|\theta\|^2/2\sigma^2\big)\Big) \tag{4.21}$$

$$= \log \sum_i p_{\mathrm{LM}}(x_i \mid \mathbf{x}_{1:i-1}) - \frac{1}{2}\log(2\pi\sigma^2) - \|\theta\|^2/2\sigma^2 \tag{4.22}$$

Remembering that $\sigma^2 = 1/\lambda$, we have:

$$= \log \sum_i p_{\mathrm{LM}}(x_i \mid \mathbf{x}_{1:i-1}) - \frac{1}{2}\log(2\pi/\lambda) - \frac{\lambda}{2}\|\theta\|^2 \tag{4.23}$$

which is the same as Eq. (4.18) minus an extra constant factor. This says that, instead of choosing θ to maximize the probability of the training data, we want to maximize the *joint* probability of the training data and θ itself, under the assumption that θ is drawn from a normal distribution (an approach referred to as **maximum a posteriori estimation**).

This approach to regularization—and these three different ways of thinking about what it does and how it works—are central components of almost all modern machine learning techniques. We'll continue to use them as we move from log-linear models to more expressive ones, ultimately including neural nets.

Log-linear models overcome many of the limitations of n-gram models—they allow much more expressive representations of the context, and can transfer information across contexts that share features with each other. But they don't solve all the problems we originally identified—in particular, we still have no way of representing (or using the fact) that *blue* and *cerulean* have similar meanings, and therefore that *every* feature involving *blue* can also be generalized to *cerulean*. In the next chapter, we'll see how to do this by moving from learned feature weights to learned representations.