Chapter 3: Classical Language Models

Learning Objectives

After reading this chapter, the reader should be able to:

1. Build an n-gram language model from a text corpus by computing conditional probabilities from count statistics.
2. Apply smoothing techniques (add-k, Good-Turing, Kneser-Ney, interpolation) to handle unseen n-grams and explain the tradeoffs between them.
3. Evaluate a language model using held-out perplexity and interpret the results in terms of model quality.
4. Articulate the fundamental limitations of count-based models – sparsity, fixed context, and inability to generalize – that motivate neural approaches.

How would you predict the next word using nothing but word counts? Chapter 2 gave us the machinery: the chain rule decomposes the probability of a sentence into a product of conditional probabilities (Equation 2.1), maximum likelihood estimation turns counts into probabilities (Equation 2.2), and perplexity gives us a single number to judge the result (Equation 2.6). We have the toolkit. Now we build something with it.

The object we build in this chapter is the n-gram language model – the oldest, most transparent, and for decades the most successful approach to predicting the next word. The idea is almost embarrassingly simple. To estimate $P(\text{mat} \mid \text{the cat sat on the})$, we count how many times "the cat sat on the mat" appears in a training corpus, divide by how many times "the cat sat on the" appears, and call the result a probability. No neural networks, no optimization, no hidden representations. Just counting and dividing. The entire model fits in a lookup table.

That simplicity is exactly the point. n-gram models make every assumption visible, every computation auditable, and every failure diagnosable. When the model assigns zero probability to a perfectly reasonable sentence, we can look at the count table and see exactly which bigram or trigram was missing. When the model generates nonsensical text, we can trace the problem to an insufficient context window. This transparency makes n-grams the right starting point for understanding language modeling, even though modern systems have long since moved beyond them. We will build a complete bigram model from scratch, confront the zero-probability disaster that makes raw counting unusable, repair it with a progression of smoothing techniques, measure the result with perplexity, and then step back to understand why counting can never be enough – why the path forward requires the continuous representations and neural architectures of later chapters.

3.1 n-gram Language Models

3.1.1 The Markov Assumption Revisited

How much context does a language model really need to predict the next word?

Consider the sentence "the cat sat on the mat" and suppose we want to estimate $P(\text{mat} \mid \text{the cat sat on the})$. The chain rule from Chapter 2 (Equation 2.1) tells us this conditional probabil-

ity is perfectly well-defined: it is the probability of "mat" given the entire preceding history. But to estimate it from data, we need to count how many times the exact five-word sequence "the cat sat on the" appeared in our training corpus and how many of those times it was followed by "mat." In a corpus of a million words, this exact five-word sequence might appear once, or twice, or never. We simply do not have enough data to estimate conditional probabilities conditioned on long histories.

The Markov assumption is the compromise. Instead of conditioning on the complete history $w_1, w_2, \ldots, w_{t-1}$, we condition on only the most recent $n-1$ words. For a bigram model ($n = 2$), we approximate $P(w_t \mid w_1, \ldots, w_{t-1}) \approx P(w_t \mid w_{t-1})$. For a trigram model ($n = 3$), we use $P(w_t \mid w_{t-2}, w_{t-1})$. The general n-gram approximation is:

$$P(w_t \mid w_1, \ldots, w_{t-1}) \approx P(w_t \mid w_{t-n+1}, \ldots, w_{t-1}) = \frac{C(w_{t-n+1}, \ldots, w_t)}{C(w_{t-n+1}, \ldots, w_{t-1})} \tag{3.1}$$

where $C(\cdot)$ denotes the count of a sequence in the training corpus.

A common and consequential confusion: the Markov assumption does not claim that language actually works this way. English is emphatically not a second-order Markov process. Subject-verb agreement can span dozens of words ("the cats, which the dog chased through the yard and around the fence, *were* tired"), and discourse coherence spans paragraphs. The Markov assumption claims only that we will *pretend* language is Markov because we lack the data to estimate the full conditional. It is an approximation born of necessity, not a belief about how language works. Shannon [1948] introduced Markov chains as models of letter and word sequences in his foundational paper on information theory, fully aware that the approximation was crude – his interest was in characterizing the statistical properties of the source, not in claiming that English speakers make word choices by consulting only the previous two words. We carry that same pragmatic attitude through this chapter: the Markov approximation is a tool, and understanding where it breaks is as important as understanding where it works.

3.1.2 Unigrams, Bigrams, and Beyond

A unigram model ($n = 1$) conditions on nothing at all: $P(w_t) = C(w_t)/N$, where N is the total token count in the training corpus. It generates text by independently sampling each word according to its corpus frequency, ignoring all context. The result is word salad – "the the sat cat on mat the" – because frequent words dominate without any sequential coherence. A bigram model ($n = 2$) conditions on the immediately preceding word. It can produce "the cat sat" because $P(\text{cat} \mid \text{the})$ is reasonably high and $P(\text{sat} \mid \text{cat})$ captures a local collocation. A trigram model ($n = 3$) conditions on the previous two words and generates more recognizable phrases: "the cat sat on" emerges because the three-word window captures short prepositional patterns.

The quality jump from unigrams to bigrams is dramatic; the jump from bigrams to trigrams is noticeable but smaller; and beyond trigrams, improvements become marginal on typical training corpora because data sparsity begins to dominate. We will quantify this diminishing-returns pattern precisely in Section 3.3.2 with a perplexity plot.

To see this concretely, here is a code example that trains unigram, bigram, and trigram models on a small corpus and generates text from each:

```
import numpy as np
```

Figure 1: Figure 3.5 – Text generation comparison across n-gram orders

```
# Toy corpus with start/end markers
corpus = ["<s> the cat sat on the mat </s>", "<s> the dog sat on the rug </s>",
          "<s> the cat chased the dog </s>", "<s> the dog chased the cat </s>"]
tokens = [s.split() for s in corpus]

def build_bigram(sents):
    counts, ctx = {}, {}
    for s in sents:
        for i in range(1, len(s)):
            counts[(s[i-1], s[i])] = counts.get((s[i-1], s[i]), 0) + 1
            ctx[s[i-1]] = ctx.get(s[i-1], 0) + 1
    return counts, ctx

def generate(counts, ctx, seed="<s>", max_len=12):
    result, w = [], seed
    for _ in range(max_len):
        cands = [(w2, c) for (w1, w2), c in counts.items() if w1 == w]
        if not cands: break
        ws, cs = zip(*cands)
        probs = np.array(cs, dtype=float) / sum(cs)
        w = np.random.choice(ws, p=probs)
        if w == "</s>": break
        result.append(w)
    return " ".join(result)

bi_counts, bi_ctx = build_bigram(tokens)
np.random.seed(42)
```

```
for _ in range(3):
    print(f"Generated: {generate(bi_counts, bi_ctx)}")
```

3.1.3 Estimating n-gram Probabilities

What does it actually look like to build a probability table by hand?

We adopt a toy corpus that we will thread through the rest of this chapter – four sentences about cats and dogs:

> **Corpus:** "the cat sat on the mat . the dog sat on the rug . the cat chased the dog . the dog chased the cat ."

After adding start-of-sentence (<s>) and end-of-sentence (</s>) tokens, the corpus becomes four sequences. We count every bigram occurrence and organize the counts into a table. The bigram (the, cat) appears 4 times because "the cat" occurs once in each of the first, third, and fourth sentences, plus once more in the fourth sentence's ending. The bigram (the, dog) appears 3 times. The bigram (sat, on) appears 2 times, and so on.

$w_{t-1}\downarrow$	the	cat	dog	sat	chased	on	mat	rug	.	texttt</s>	Sigma
<s>	4	0	0	0	0	0	0	0	0	0	**4**
the	0	4	3	0	0	0	1	1	0	0	**10**
cat	0	0	0	1	1	0	0	0	1	0	**3**
dog	0	0	0	1	1	0	0	0	1	0	**3**
sat	0	0	0	0	0	2	0	0	0	0	**2**
chased	2	0	0	0	0	0	0	0	0	0	**2**
on	2	0	0	0	0	0	0	0	0	0	**2**
mat	0	0	0	0	0	0	0	0	1	0	**1**
rug	0	0	0	0	0	0	0	0	1	0	**1**
.	0	0	0	0	0	0	0	0	0	4	**4**

Shading intensity ∝ count. Zero cells (light gray) preview the sparsity problem.

Figure 2: Figure 3.1 – Bigram count table for the toy corpus

To convert counts to probabilities, we divide each cell by its row sum: $P(w_t \mid w_{t-1}) = C(w_{t-1}, w_t)/C(w_{t-1})$. For the context word "the," we have $C(\text{the}) = 10$ total occurrences as a bigram prefix (across all four sentences). Of those, 4 are followed by "cat," 3 by "dog," 1

by "mat," and 1 by "rug," plus 1 by the period. This gives us $P(\text{cat} \mid \text{the}) = 4/10 = 0.4$, $P(\text{dog} \mid \text{the}) = 3/10 = 0.3$, and so on. The probabilities in each row must sum to 1.0 – if they do not, we have miscounted, and that is the first sanity check to apply.

One detail the formulas hide: we need the start and end tokens. Without `<s>`, the model has no way to estimate the probability of a sentence-initial word. Without `</s>`, the model assigns no probability mass to stopping – it generates forever, or until it hits a word with no recorded successors. These boundary tokens are not decorative; they are structurally necessary for the model to define a proper probability distribution over sentences of varying length.

Here is the complete pipeline – from raw text to generated sentence to perplexity – in code:

```
import numpy as np

# Toy corpus with start/end tokens
corpus = [
    "<s> the cat sat on the mat </s>",
    "<s> the dog sat on the rug </s>",
    "<s> the cat chased the dog </s>",
    "<s> the dog chased the cat </s>",
]
sents = [s.split() for s in corpus]
vocab = sorted(set(w for s in sents for w in s))
w2i = {w: i for i, w in enumerate(vocab)}
V = len(vocab)

# Build bigram count matrix
C = np.zeros((V, V), dtype=int)
for s in sents:
    for a, b in zip(s, s[1:]):
        C[w2i[a], w2i[b]] += 1
P = C / C.sum(axis=1, keepdims=True).clip(1)  # normalize rows
print("Bigram P(cat|the) =", round(P[w2i["the"], w2i["cat"]], 3))

# Generate a sentence
np.random.seed(7)
w = "<s>"
sent = []
for _ in range(20):
    probs = P[w2i[w]]
    nxt = vocab[np.random.choice(V, p=probs)]
    if nxt == "</s>": break
    sent.append(nxt)
    w = nxt
print("Generated:", " ".join(sent))

# Perplexity on held-out sentence
test = "<s> the cat sat on the rug </s>".split()
log_prob = sum(np.log2(P[w2i[a], w2i[b]]) for a, b in zip(test, test[1:]))
```

```
pp = 2 ** (-log_prob / (len(test) - 1))
print(f"Perplexity: {pp:.1f}")
```

3.1.4 Higher-Order n-grams and the Bias-Variance Tradeoff

Increasing n always helps – until it does not.

This sounds like a contradiction, so let us be precise. A higher-order n-gram model has access to more context, which means it can make sharper, more informed predictions. A trigram model knows that "New York" is very likely to be followed by "City" or "Times," while a bigram model only sees "York" and must guess from a broader distribution. In the bias-variance framework, increasing n reduces the *bias* of the model: it captures more of the true conditional distribution by conditioning on more history. But it simultaneously increases the *variance*, because longer n-grams are observed far fewer times in the training corpus, making the count-based estimates noisy and unreliable. A 5-gram that appears exactly once in the training data gives us a maximum likelihood estimate of either 0 or 1 for the next word – an estimate driven entirely by the accident of which word happened to follow that particular 5-gram in the training text, not by any robust statistical signal.

The practical consequence is a sweet spot, typically around $n = 3$ for medium-sized corpora and $n = 5$ for very large ones, beyond which additional context hurts more than it helps because most of the longer n-grams simply have not been observed. This is the same bias-variance tradeoff that appears throughout statistics and machine learning, but in language modeling it has a particularly vivid character: the "variance" side of the tradeoff manifests as zero counts for most n-grams, which is not merely noisy estimation but complete absence of evidence. We explore this tension empirically in Section 3.3.2, and we return to it from a different angle in Section 3.4.1, where we compute just how many n-grams exist versus how many we can plausibly observe.

There is another effect of very high n that is worth noting, though it receives less attention in textbooks. When n is large enough that most n-grams in the training data are unique, the model effectively memorizes the training corpus. Text generated from such a model does not sound "fluent" in any creative sense – it reproduces verbatim stretches of the training data, because at each step there is only one recorded continuation. This is the language-modeling analogue of overfitting in supervised learning: the model has zero training perplexity (it assigns probability 1 to whatever the training data says) but poor test perplexity (it assigns probability 0 to anything it has not seen before).

3.2 Smoothing Techniques

3.2.1 The Zero-Frequency Problem

What happens when a language model encounters a word sequence it has never seen before?

Every n-gram model built by maximum likelihood is broken.

That is not rhetorical. Consider the held-out test sentence "the cat sat on the rug," evaluated under our bigram model from Section 3.1.3. Every bigram in this sentence was observed in the training corpus, so the model assigns a finite probability and we get a reasonable perplexity. But now consider the test sentence "the rug sat on the cat." The bigram (rug, sat) never appeared in training. Its count is zero. Its probability under MLE is zero. And because the probability of a sentence is the product of its bigram probabilities (by the chain rule), a single zero factor makes the

entire sentence probability zero. Zero probability means infinite perplexity. The model has declared, with absolute confidence, that this sentence is impossible.

This is absurd. The sentence is unusual, perhaps even unlikely, but it is not impossible. A rug cannot sit on a cat, but "the rug sat on the cat" is a grammatically well-formed English sentence that a creative writer might produce. Any useful language model must assign it some probability, however small. Yet MLE says zero, and zero multiplied by anything is still zero – there is no recovery within the MLE framework.

The problem is pervasive, not exotic. In Section 2.2.4, we noted that zero probabilities are a theoretical concern. Now we see why they are a practical catastrophe. Even for bigrams, a vocabulary of 10,000 words yields 100 million possible bigram types. A training corpus of one million tokens contains at most one million bigram instances, and due to Zipf's law (see sidebar below) most of those are repetitions of the same common bigrams. In practice, fewer than 1% of possible bigrams are ever observed in a typical corpus. For trigrams the situation is exponentially worse: $|V|^3 = 10^{12}$ possible types, of which we might observe a few million. The zero-frequency problem is not an edge case. It is the default state of affairs for most of the probability table.

Sidebar: Zipf's Law and the Long Tail

In 1935, the linguist George Kingsley Zipf documented a remarkably consistent pattern: if we rank the words in a language by frequency, the frequency of the r-th ranked word is approximately proportional to $1/r$. The most frequent English word ("the") appears roughly twice as often as the second most frequent ("of"), three times as often as the third ("and"), and so on. Formally, $f(r) \propto 1/r^s$ with $s \approx 1$. This power-law distribution, now called Zipf's law, has a striking consequence for language modeling. A small number of words – perhaps the top 100 – account for roughly half of all tokens in a typical English corpus. The remaining half is spread across tens of thousands of rare words, each appearing only a handful of times. This "long tail" of rare words is precisely where smoothing matters most: these are the words the model has seen too few times to estimate reliably, and they are also the words most likely to form unseen n-grams with their neighbors. Kneser-Ney smoothing, which we develop in Section 3.2.4, is particularly effective in the long tail because its continuation probability distinguishes between rare words that appear in many different contexts (genuinely versatile words, useful as backoff predictions) and rare words locked to a single context (like "Francisco," which almost exclusively follows "San"). Zipf's law also explains why increasing the training corpus size helps but never fully solves the sparsity problem: doubling the corpus roughly doubles our observations of common words but barely touches the rare ones, because the long tail grows as fast as the data [Zipf, 1935; Manning and Schutze, 1999].

3.2.2 Additive (Laplace) Smoothing

What is the simplest way to prevent a language model from assigning zero probability to unseen word sequences?

Pierre-Simon Laplace proposed the rule of succession in 1814 to address the problem of events not yet observed. If the sun has risen every day for 5,000 years, what is the probability it rises tomorrow? Laplace's answer: add one to both the observed and unobserved counts and re-normalize. Applied to bigrams, additive smoothing adds a constant k to every count before dividing:

$$P_{\text{add-k}}(w_t \mid w_{t-1}) = \frac{C(w_{t-1}, w_t) + k}{C(w_{t-1}) + k|V|} \tag{3.2}$$

When $k = 1$, this is Laplace smoothing; smaller values like $k = 0.01$ are sometimes called Lidstone smoothing. The formula guarantees that no bigram has zero probability – the minimum count is k rather than 0 – which eliminates the infinite-perplexity catastrophe from Section 3.2.1.

The trouble is that add-k smoothing does its job too aggressively for large vocabularies. Consider a vocabulary of $|V| = 50{,}000$ words. For a context word w_{t-1} that appeared $C(w_{t-1}) = 100$ times in training, the denominator under add-1 smoothing becomes $100 + 50{,}000 = 50{,}100$. The model has effectively diluted 100 observations across 50,000 phantom observations. A bigram that appeared 10 times in training, which under MLE would have probability $10/100 = 0.10$, now has probability $11/50{,}100 \approx 0.0002$ – a reduction by a factor of 500. Meanwhile, a bigram that was never observed gets $1/50{,}100 \approx 0.00002$, which is only ten times smaller. The model has taken a bigram that is genuinely common and made it almost as unlikely as one it has never seen. This is absurd in the opposite direction from MLE: where MLE was too confident (assigning zero to unseen events), add-1 is too egalitarian (treating frequent and never-seen events as nearly equally probable).

The Bayesian perspective makes the pathology transparent. Add-k smoothing is mathematically equivalent to the posterior mean of a multinomial distribution with a symmetric Dirichlet prior of concentration parameter $\alpha = k$. The "prior" says: before seeing any data, every next word is equally likely. That prior is reasonable when $|V|$ is small, but with a vocabulary of tens of thousands, it injects a massive amount of pseudo-evidence for nonsensical bigrams. In practice, add-k smoothing with $k = 1$ is almost never used in serious systems; it survives in textbooks (this one included) as the simplest illustration of how smoothing works, and as a cautionary example of how smoothing can go wrong. Chen and Goodman [1999] confirmed this empirically: add-1 smoothing consistently produced the highest (worst) perplexity among all methods tested, across all corpora and all n-gram orders.

Figure 3.2: Smoothing Comparison for Bigrams Following "the"

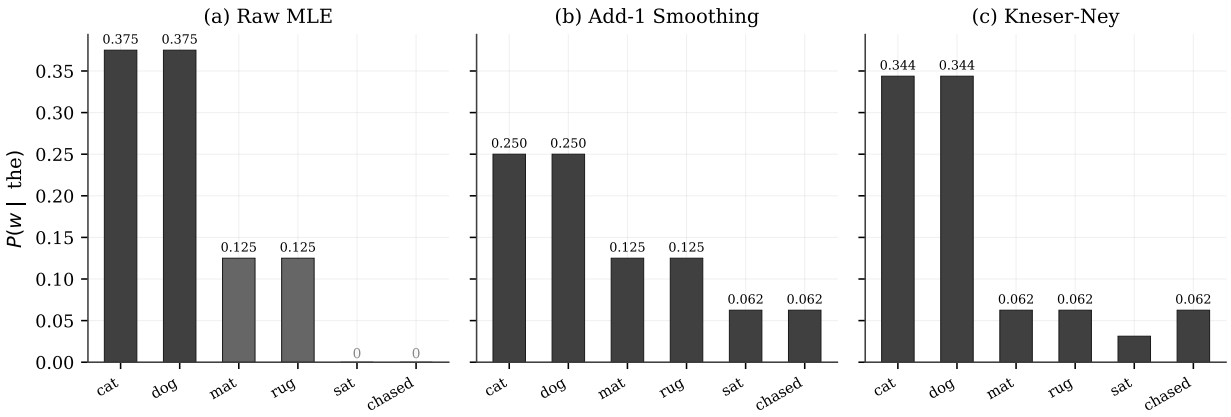

Figure 3: Figure 3.2 – Smoothing comparison

3.2.3 Good-Turing Estimation

The Good-Turing formula has an elegant motivation, but we confess we find the standard textbook explanation slightly misleading, so let us start from the limitation it addresses and work backward.

Add-k smoothing redistributes probability mass uniformly to all unseen events. But the total mass that *should* go to unseen events is not obvious. How much probability should we reserve for things we have never seen? Good-Turing estimation, proposed by Alan Turing and I.J. Good during World War II codebreaking work and published by Good [1953], answers this question using a clever statistical argument. The key idea is that the total probability mass of all events that occur exactly r times in training can be estimated from N_{r+1}, the number of event types that occur $r+1$ times. Specifically, the adjusted count for events seen r times is:

$$r^* = (r+1)\frac{N_{r+1}}{N_r}$$

where N_r is the number of distinct n-gram types observed exactly r times. The total probability reserved for unseen events (those with $r = 0$) is N_1/N, where N_1 is the number of n-gram types seen exactly once (the "hapax legomena") and N is the total number of n-gram tokens.

The intuition, which is more subtle than it first appears, is this: the n-grams we have seen exactly once are our best evidence about the world of n-grams we have not seen at all. If we re-collected the data from the same source, many of the once-seen n-grams would not reappear, and many currently-unseen n-grams would appear for the first time. The once-seen n-grams are, in a sense, proxies for the unseen ones. This proxy argument makes Good-Turing estimation remarkably effective in practice – better than add-k by a wide margin – though it has its own practical difficulties. The counts N_r can be zero for large r (there might be no n-grams seen exactly 47 times), requiring additional smoothing of the frequency-of-frequency counts themselves. Modern implementations typically smooth the N_r values with a regression before applying the formula, but this adds complexity that Kneser-Ney avoids. Whether Good-Turing deserves its continued prominence in NLP curricula, given that Kneser-Ney has superseded it in all practical applications, is a question we leave to the reader. The historical importance is beyond dispute; the contemporary relevance is debatable.

3.2.4 Kneser-Ney Smoothing

Suppose you are filling in the blank: "She ate a bowl of ____." Your bigram model backs off to the unigram distribution over single words. Under a standard unigram backoff, the word "Francisco" might rank fairly high because it appears 500 times in a corpus about California history. But "Francisco" almost exclusively follows "San" – it is not a versatile word that fits into many different contexts. "Rice," by contrast, might appear only 50 times but after dozens of different words: "ate rice," "cooked rice," "white rice," "brown rice," "fried rice." When we need a backoff prediction, "rice" is a far better candidate than "Francisco," despite being less frequent overall.

This observation is the core insight of Kneser-Ney smoothing [Kneser and Ney, 1995]: when backing off from a higher-order model, we should not use raw unigram frequency but instead use *continuation probability* – how many distinct contexts a word has appeared in. The continuation probability of a word w is:

$$P_{\text{continuation}}(w) = \frac{|\{w' : C(w', w) > 0\}|}{|\{(w', w'') : C(w', w'') > 0\}|}$$

The numerator counts the number of distinct word types that precede w in the training corpus. The denominator is the total number of distinct bigram types. "Francisco" might have $|\{w' : C(w', \text{Francisco}) > 0\}| = 1$ (only "San"), while "rice" has $|\{w' : C(w', \text{rice}) > 0\}| = 30$. Their continuation probabilities reflect this difference, giving "rice" a much higher backoff probability.

The full Kneser-Ney formula combines absolute discounting – subtracting a fixed discount d (typically around 0.75) from each observed count – with the continuation-probability backoff:

$$P_{\text{KN}}(w_t \mid w_{t-1}) = \frac{\max(C(w_{t-1}, w_t) - d, 0)}{C(w_{t-1})} + \lambda(w_{t-1})\, P_{\text{continuation}}(w_t) \tag{3.3}$$

where $\lambda(w_{t-1}) = \frac{d \cdot |\{w : C(w_{t-1}, w) > 0\}|}{C(w_{t-1})}$ is a normalization factor that distributes the discounted mass.

The subtraction of d from each observed count is the "absolute discounting" component: instead of the complex re-estimation of Good-Turing, we simply shave a fixed amount off every count and redistribute the freed mass via the continuation distribution. Chen and Goodman [1999] showed that interpolated modified Kneser-Ney – where the continuation probability is added to the discounted estimate rather than used only when the higher-order count is zero – consistently achieves the lowest perplexity among all classical smoothing methods, across all corpora and all n-gram orders they tested. That result has stood for over two decades.

Here is a code example comparing MLE, add-1, and simplified Kneser-Ney smoothing on our toy corpus:

```
import numpy as np

corpus = ["<s> the cat sat on the mat </s>", "<s> the dog sat on the rug </s>",
          "<s> the cat chased the dog </s>", "<s> the dog chased the cat </s>"]
sents = [s.split() for s in corpus]
vocab = sorted(set(w for s in sents for w in s))
V = len(vocab)
# Count bigrams and unigrams
bi, uni = {}, {}
for s in sents:
    for a, b in zip(s, s[1:]):
        bi[(a,b)] = bi.get((a,b), 0) + 1; uni[a] = uni.get(a, 0) + 1
# Continuation counts for Kneser-Ney (d=0.75)
pred = {}
for (a, b) in bi: pred.setdefault(b, set()).add(a)
N_types, d = len(bi), 0.75

test = "<s> the rug sat on the cat </s>".split()
for name, method in [("MLE","mle"), ("Add-1","add1"), ("KN","kn")]:
    lp = 0.0
    for a, b in zip(test, test[1:]):
        cab, ca = bi.get((a,b), 0), uni.get(a, 0)
        if method == "mle": p = cab / ca if ca > 0 else 0
        elif method == "add1": p = (cab + 1) / (ca + V)
        else:  # Kneser-Ney
```

```
            n_fol = len([w for w in vocab if bi.get((a,w),0)>0])
            p = max(cab-d,0)/ca + d*n_fol/ca * len(pred.get(b,set()))/N_types
        lp += np.log2(p) if p > 0 else float("-inf")
    pp = 2**(-lp/(len(test)-1)) if lp > float("-inf") else float("inf")
    print(f"{name}: perplexity = {pp:.1f}")
```

3.2.5 Interpolation and Backoff

The smoothing techniques discussed so far operate within a single n-gram order. Interpolation takes a different approach: it blends estimates from multiple orders simultaneously, using the reliable but coarse unigram estimate, the more specific bigram estimate, and the most specific trigram estimate together in a weighted sum:

$$P_{\text{interp}}(w_t \mid w_{t-2}, w_{t-1}) = \lambda_3 \, P_3(w_t \mid w_{t-2}, w_{t-1}) + \lambda_2 \, P_2(w_t \mid w_{t-1}) + \lambda_1 \, P_1(w_t) \tag{3.4}$$

where $\lambda_1 + \lambda_2 + \lambda_3 = 1$ and each $\lambda_i \geq 0$.

The rationale is straightforward: higher-order n-grams are more precise when they have sufficient data, but lower-order n-grams are more robust because they are estimated from larger sample sizes. The interpolation weights λ_i control the balance. If the trigram (w_{t-2}, w_{t-1}, w_t) was observed many times, we want λ_3 to be large; if the trigram context is rare, we want to lean more heavily on the bigram and unigram. In practice, the λ_i are tuned on a held-out validation set, either by grid search or by the expectation-maximization (EM) algorithm [Jelinek and Mercer, 1980].

A common misconception is that interpolation and backoff are interchangeable – the distinction is often confused but matters. Interpolation *always* mixes all orders, regardless of whether the higher-order n-gram was observed. Backoff, by contrast, uses the highest available order and falls back to a lower order only when the higher-order count is zero. Katz backoff [Katz, 1987] is the classical example: use the trigram probability if the trigram was seen in training; otherwise, use the (discounted) bigram probability; otherwise, use the unigram. The two strategies lead to different behaviors on rare n-grams. Interpolation's strength is that even when a trigram count is available, the unigram and bigram components act as regularizers, preventing the model from being overconfident in a trigram estimate based on very few observations. Backoff's strength is that it fully trusts the highest available order, which can be advantageous when that estimate is based on ample data.

Modified Kneser-Ney, the gold standard we described in Section 3.2.4, actually uses interpolation rather than pure backoff – the continuation-probability term is *added* to the discounted higher-order estimate, not used only when the higher-order count is zero. This is one of the refinements from Chen and Goodman [1999] that pushed Kneser-Ney to the top of every smoothing comparison. The combination of absolute discounting, continuation probability, and interpolation across orders produces a method that is difficult to beat with any count-based technique, and that served as the dominant language modeling approach from the mid-1990s until the advent of neural language models in the early 2010s.

3.3 Evaluating Language Models

3.3.1 Perplexity in Practice

How do we measure, in a single number, how well a language model predicts held-out text?

In Chapter 2 (Section 2.3), we introduced perplexity as PP $= 2^{H(P,Q)}$, where $H(P,Q)$ is the cross-entropy between the true distribution P and the model distribution Q. For n-gram models, this formula takes a particularly concrete form. Given a test sequence $W = w_1, w_2, \ldots, w_N$ and a model that assigns probability $P(w_t \mid w_{t-n+1}, \ldots, w_{t-1})$ to each token, the perplexity is:

$$\text{PP}(W) = \left(\prod_{t=1}^{N} \frac{1}{P(w_t \mid w_{t-n+1}, \ldots, w_{t-1})} \right)^{1/N} = 2^{-\frac{1}{N} \sum_{t=1}^{N} \log_2 P(w_t | w_{t-n+1}, \ldots, w_{t-1})} \tag{3.5}$$

The interpretation is intuitive once you internalize the formula: perplexity is the geometric mean of the inverse probabilities. If the model assigns probability 1.0 to every test word (a perfect oracle), the perplexity is 1.0 – the model is never "perplexed." If the model assigns equal probability $1/|V|$ to every word (a uniform model), the perplexity equals $|V|$ – the model is maximally confused, as if it were blindly guessing among all vocabulary items. A good model falls between these extremes, and lower perplexity is always better. Perplexity of 100 means the model's predictions are, on average, as uncertain as if it were choosing uniformly among 100 equally likely words at each step.

A practical note that trips up many students: the test tokens $w_1, \ldots, w_N$ include end-of-sentence tokens but typically exclude start-of-sentence tokens from the count N. The start token is part of the context (the model needs it to predict the first real word), but we do not "predict" it – it is given. Miscounting N by even one token changes the perplexity, sometimes dramatically for short test sets. Another common mistake is forgetting to handle out-of-vocabulary (OOV) words. If the test set contains a word not in the model's vocabulary, the model cannot assign it any probability. The standard practice is to replace all words below a frequency threshold with a special `<UNK>` token during training, and to replace any OOV words in the test set with the same token. This ensures that the model always has a probability estimate, but it also means that perplexity numbers are only comparable when computed with the same vocabulary and OOV treatment.

Domain mismatch amplifies these issues. A model trained on newswire text will have excellent perplexity on a newswire test set but terrible perplexity on medical records, because the conditional distributions are simply different. Reporting a perplexity number without specifying the training and test domains, the vocabulary size, and the OOV rate is like reporting a race time without specifying the distance – the number is meaningless in isolation. The Penn Treebank (PTB) corpus, with its standardized 10,000-word vocabulary and fixed train/valid/test splits, became the canonical benchmark for exactly this reason: it enabled apples-to-apples comparisons across years of language modeling research. On PTB, a well-tuned trigram with modified Kneser-Ney smoothing achieves a perplexity of roughly 141, which served as the baseline that neural language models had to beat [Chen and Goodman, 1999]. We note, with some irony, that this 141-perplexity baseline has outlived the corpus itself – PTB's 1989 Wall Street Journal text is so dated that modern models trained on it are essentially modeling the English of a vanished era. Yet the number persists in papers published in 2024.

3.3.2 Effect of n-gram Order on Perplexity

Does adding more context always help?

The answer depends on how much training data you have, and the relationship between context length and data size is the central empirical fact of n-gram language modeling. On a sufficiently large training corpus, moving from a unigram to a bigram typically cuts perplexity by 40–60%. Moving from bigram to trigram gives another 15–25% reduction. From trigram to 4-gram, the

improvement shrinks to 5–10%. And from 4-gram to 5-gram, the improvement is often negligible or even negative – perplexity actually *increases* because the model encounters so many unseen 5-grams that even good smoothing cannot fully compensate.

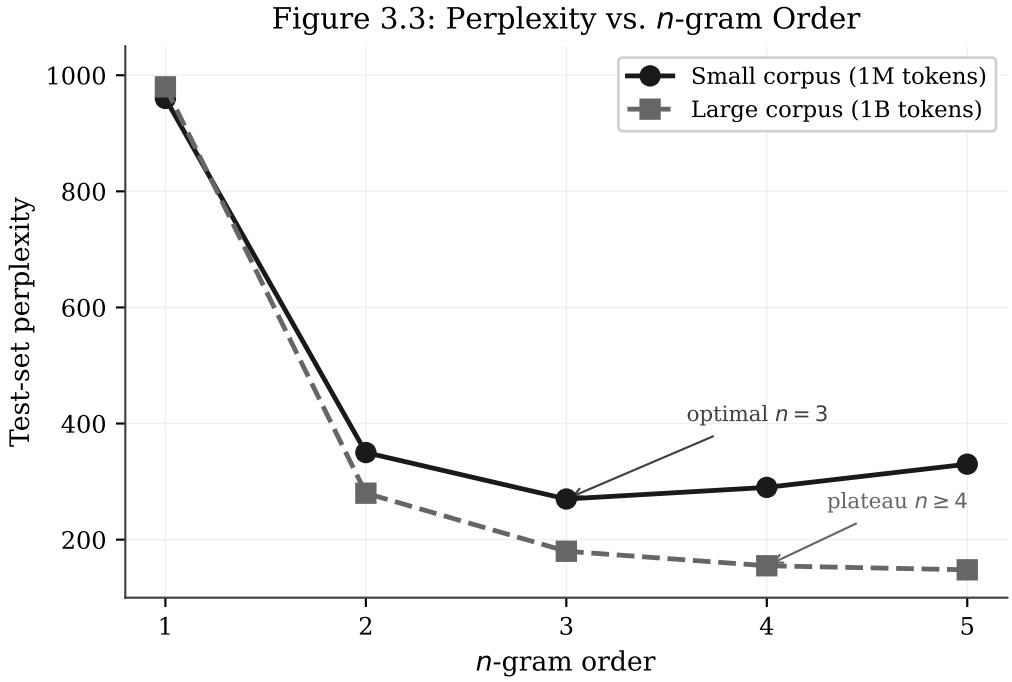

Figure 4: Figure 3.3 – Perplexity versus n-gram order on held-out data

The shape of this curve is the bias-variance tradeoff from Section 3.1.4 made visible. The left side of the curve (low n) is bias-dominated: the model does not have enough context to make good predictions, so adding context helps enormously. The right side (high n) is variance-dominated: the model has plenty of context in principle, but the estimates are unreliable because most high-order n-grams were never observed. The minimum of the curve – the optimal n – moves to the right as the training corpus grows, because more data means more n-gram observations, which delays the onset of sparsity. On the 1-million-word Penn Treebank, the optimal order is typically $n = 3$. On billion-word corpora, $n = 5$ models can improve over trigrams, though the gains are modest compared to the jump from unigrams to bigrams.

This pattern has a deeper implication that we will return to in Section 3.4: the diminishing returns of n-gram order suggest that counting, no matter how sophisticated the smoothing, cannot fully exploit long-range context. The information in words ten or twenty positions back is real – "the students who passed the difficult exam *were* celebrating" requires knowing the plural subject to predict the plural verb – but n-gram models cannot reach it without facing an explosion of unseen high-order n-grams. This is the architectural ceiling that recurrent neural networks (Chapter 5) and transformers (Chapter 8) were designed to break through.

3.4 Limitations and the Road to Neural Models

3.4.1 The Curse of Dimensionality

n-gram language models have a fatal scaling problem, and the numbers are worth computing explicitly.

A vocabulary of $|V| = 50{,}000$ words – modest by modern standards – produces $|V|^2 = 2.5 \times 10^9$ possible bigrams, $|V|^3 = 1.25 \times 10^{14}$ possible trigrams, and $|V|^5 = 3.125 \times 10^{23}$ possible 5-grams. The largest training corpora available to researchers in the n-gram era contained roughly 10^9 tokens (one billion words, as in the Google Web 1T corpus). That is enough to observe most common bigrams multiple times, a reasonable fraction of trigrams at least once, but only a vanishingly small sliver of 5-grams. Bengio et al. [2003] identified this exponential growth of the n-gram space as the "curse of dimensionality" for language models and argued that it is the fundamental barrier that count-based methods cannot overcome.

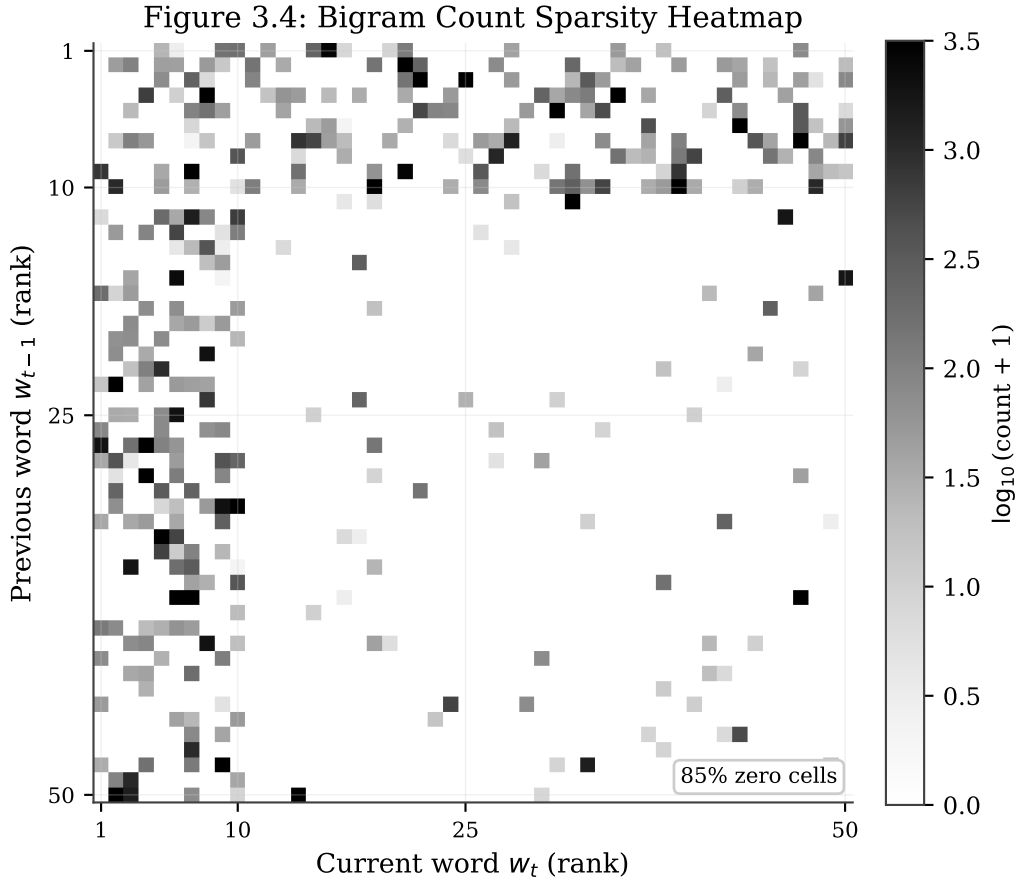

Figure 5: Figure 3.4 – Sparsity heatmap

The sparsity is not uniformly distributed. Common function words like "the," "of," "and" form dense rows in the bigram matrix because they combine freely with many content words. Content words like "serendipity" or "photosynthesis" form sparse rows because they appear in only a few specific contexts. Zipf's law guarantees that the number of very sparse rows vastly outnumbers the dense ones. Smoothing helps redistribute probability mass to unseen cells, but it does so blindly:

add-k smoothing spreads mass uniformly, and even Kneser-Ney, while smarter, cannot distinguish between the unseen bigram "cat photosynthesis" (nonsensical) and "dog played" (perfectly plausible but absent from the training data). The model has no mechanism for knowing that "dog" and "cat" are related, that "played" and "ran" describe similar activities, or that the plausible unseen bigrams are a tiny subset of all unseen bigrams. This brings us to the second limitation.

3.4.2 No Semantic Generalization

Try this thought experiment. You train a bigram model on a corpus that contains the sentence "the dog chased the cat" but not "the puppy chased the kitten." What has the model learned about puppies chasing kittens?

Nothing. Absolutely nothing.

In an n-gram model, every word is an atomic symbol with no internal structure. "Dog" is symbol number 4,731 and "puppy" is symbol number 28,509, and the model sees no more connection between them than between "dog" and "thermodynamics." If "puppy chased" never appeared in training, its count is zero and its MLE probability is zero, regardless of how many times "dog chased" appeared. Smoothing gives "puppy chased" a small non-zero probability, but it gives "thermodynamics chased" the same small probability. The model cannot distinguish a plausible unseen event from a nonsensical one because it has no representation of what words *mean*.

This limitation is worth dwelling on because it is so complete. It is not that n-gram models have *poor* semantic generalization – they have literally *none*. Seeing "we love Italian food" does not help predict "we love French cuisine" even slightly, because every word in the second sentence is a different symbol from every word in the first. A human reader generalizes instantly: "Italian" and "French" are nationalities, "food" and "cuisine" are near-synonyms, so the second sentence is obviously plausible given the first. But generalization requires a representation in which similar words occupy nearby points in some space, and count tables provide no such representation. Each word occupies its own isolated dimension, orthogonal to all others.

This is the problem that Chapter 4 (word embeddings) solves. By representing words as dense vectors in a continuous space – where "dog" and "puppy" are nearby, and "cat" and "kitten" are nearby – a model can automatically transfer knowledge from observed contexts to semantically related but unobserved ones. Bengio et al. [2003] articulated this argument with particular clarity: "a word sequence can inform the model about other word sequences that share similar features." The transition from discrete symbol tables to continuous vector spaces is, in retrospect, the single most important conceptual shift in the history of language modeling.

3.4.3 Fixed Context Windows

Five words of history. That is the boundary of the most aggressive n-gram model in common use.

The English sentence "The students who passed the extremely difficult final examination in advanced organic chemistry last semester *were* celebrating at the restaurant" requires knowing that the subject is "students" (plural) to predict the verb "were" (also plural) rather than "was." But the verb appears 14 words after the subject, far beyond the reach of any practical n-gram. A trigram model, seeing only "last semester ***," has no access to the subject at all. A 5-gram model sees "organic chemistry last semester*** ," which is barely better. The grammatically relevant information is simply out of reach, lost beyond the fixed context window.

Increasing n to 15 or 20 is not a viable solution. As we computed in Section 3.4.1, the number of possible n-grams grows as $|V|^n$. A 15-gram with a 50,000-word vocabulary produces $50{,}000^{15} \approx 3 \times 10^{70}$ possible types – a number that exceeds the estimated number of atoms in the observable universe. No training corpus can provide meaningful count estimates for more than a negligible fraction of these n-grams. The fixed context window is not merely a practical limitation that more data could solve. It is an architectural dead end, a consequence of the decision to model sequences with discrete count tables.

This is the limitation that recurrent neural networks (Chapter 5) address directly. An RNN maintains a hidden state vector $\mathbf{h}_t$ that is updated at every time step, accumulating information from the entire preceding sequence. In principle, $\mathbf{h}_t$ can carry information about the subject of a sentence across an arbitrarily long intervening clause. In practice, vanilla RNNs suffer from gradient vanishing that limits effective context to roughly 10–20 tokens, but LSTMs and GRUs extend this to hundreds of tokens – far beyond what any n-gram model can achieve. The Transformer architecture (Chapter 8) goes further by attending directly to every position in the input sequence, eliminating the sequential bottleneck entirely.

3.4.4 Looking Ahead: From Counts to Vectors

The three limitations we have diagnosed – sparsity, no semantic generalization, and fixed context – are not independent failures that happen to coexist. They are symptoms of a single architectural choice: representing words as discrete, atomic symbols and modeling their dependencies through count tables. Once we accept that choice, sparsity is inevitable (the count table is exponentially large), generalization is impossible (discrete symbols have no similarity structure), and context is fixed (extending the table to longer n-grams multiplies the sparsity exponentially).

The alternative, first articulated clearly by Bengio et al. [2003], is to replace the discrete count table with a function that maps word sequences to probability distributions, parameterized by continuous weights that are learned from data. In this framework, each word is represented not by a row number in a count table but by a dense vector of, say, 300 real numbers – a *word embedding* – learned so that words appearing in similar contexts end up with similar vectors. The probability $P(w_t \mid w_{t-n+1}, \ldots, w_{t-1})$ is computed not by looking up a count but by feeding the embedding vectors of the context words into a neural network that outputs a distribution over the vocabulary.

This shift from counts to vectors addresses all three limitations simultaneously. Sparsity is tamed because the neural network generalizes through its continuous parameters: even if the specific word sequence was never observed, the network can produce a reasonable estimate if the input embeddings are close to those of observed sequences. Semantic generalization becomes automatic: if "dog" and "puppy" have similar embeddings, then any pattern learned about "dog" transfers immediately to "puppy." And context windows can be extended without an exponential explosion of parameters, because the hidden state of a recurrent network or the attention mechanism of a transformer compresses context into a fixed-size vector rather than enumerating all possible n-grams.

The next chapter begins the shift. Instead of counting how often "dog" appears next to "chased," we ask a different question: can we represent "dog" and "puppy" as nearby points in a space, so that anything we learn about one transfers automatically to the other? The n-gram model, for all its limitations, remains the right starting point: it makes every computation transparent, it provides the baseline perplexity that neural models must beat, and its failures illuminate exactly what the neural models need to do differently. The road from counting to vectors is the central narrative arc of this book, and we have now taken the first step.

Sidebar: The IBM Language Model

In the early 1970s, Frederick Jelinek left his position at Cornell University to lead the Continuous Speech Recognition group at IBM's Thomas J. Watson Research Center. The group's mission was ambitious: build a system that could transcribe spoken English into text. Jelinek's crucial insight was to decompose the problem using Bayes' theorem – model the probability of a word sequence $P(W)$ separately from the probability of the acoustic signal given the words $P(A \mid W)$ – making the language model $P(W)$ an independent, reusable component. His team systematically developed n-gram estimation techniques, interpolation methods for combining models of different orders, and the perplexity metric for evaluating language models. Jelinek reportedly quipped, "Every time I fire a linguist, the performance of the speech recognizer goes up" – a provocative (and likely apocryphal, or at least exaggerated for effect) summary of the statistical revolution in NLP. The honest answer for why this quote resonated is partly historical: the rule-based linguistic approaches of the 1960s had hit a ceiling, and the IBM group's data-driven methods were genuinely producing better results. But the quote obscures the real lesson, which is that Jelinek's statistical models captured linguistic regularities *implicitly* through counts rather than *explicitly* through hand-written rules. Linguists were not wrong about language; the statistical approach was simply more scalable. Jelinek's group at IBM established the n-gram language model as the standard tool for speech recognition and machine translation, a position it held for three decades. The intellectual lineage from Jelinek's work leads directly to the neural language models of the 2010s, and the perplexity metric his group popularized remains the primary evaluation measure in the field today [Jelinek, 1976; Jelinek, 1997].

The limitations of count-based methods provide a natural transition to Chapter 4, where we explore how continuous vector representations of words overcome the sparsity and rigidity of discrete n-gram models.

Exercises

Exercise 3.1 (Theory – Basic). Using the toy corpus from Section 3.1.3 ("the cat sat on the mat / the dog sat on the rug / the cat chased the dog / the dog chased the cat"):

(a) List every unique bigram in the corpus with its count, including boundary tokens – each sentence begins with `<s>` and ends with `</s>`, so bigrams like (`<s>`, the) and (., `</s>`) must be counted. From these counts, compute the full bigram probability table $P(w_t \mid w_{t-1})$ for all observed context words.

(b) Verify that $\sum_{w_t} P(w_t \mid w_{t-1}) = 1$ for each context word w_{t-1}. If any row does not sum to 1.0, identify where the miscount occurred.

Exercise 3.2 (Theory – Intermediate). Prove that add-1 (Laplace) smoothing is equivalent to the posterior mean of a multinomial distribution with a symmetric Dirichlet prior with concentration parameter $\alpha = 1$. Specifically, show that if we place a Dirichlet$(\alpha, \alpha, \ldots, \alpha)$ prior on the multinomial parameters $\theta_1, \ldots, \theta_{|V|}$ and observe counts $C_1, \ldots, C_{|V|}$, the posterior mean is:

$$\mathbb{E}[\theta_i \mid C] = \frac{C_i + \alpha}{\sum_j C_j + |V|\alpha}$$

What does this imply about the prior assumptions of add-1 smoothing? Why are these assumptions problematic for large vocabularies?

Hint: Recall that the Dirichlet distribution is the conjugate prior for the multinomial. The posterior is Dirichlet$(C_1 + \alpha, \ldots, C_{|V|} + \alpha)$, and the mean of a Dirichlet is the normalized parameter vector.

Exercise 3.3 (Theory – Advanced). Explain, with a concrete numerical example, why the continuation probability $P_{\text{continuation}}(w) = |\{w' : C(w', w) > 0\}|/|\{(w', w'') : C(w', w'') > 0\}|$ is a better backoff distribution than the standard unigram $P(w) = C(w)/N$. Use the "San Francisco" scenario: suppose "Francisco" appears 500 times in a corpus but always after "San," while "rice" appears 50 times but after 30 different preceding words. Compute both the unigram probability and the continuation probability for each word, and discuss which gives better predictions when the model needs to back off from an unseen bigram context.

Hint: The continuation probability of "Francisco" should be very low (only 1 distinct predecessor) despite its high raw frequency, while "rice" should have a much higher continuation probability (30 distinct predecessors).

Exercise 3.4 (Programming – Basic). This exercise establishes the baseline that every subsequent model must beat. Implement a unigram language model in Python using NumPy. Train it on the toy corpus from Section 3.1.3 with add-1 smoothing. Compute the perplexity on the held-out sentence "the cat sat on the rug." Compare the perplexity to $|V|$ (the vocabulary size) and explain why unigram perplexity must fall between 1 and $|V|$.

Exercise 3.5 (Programming – Intermediate). Build a trigram model with add-1 smoothing on a larger text (use at least 20 sentences – you may use nursery rhymes, song lyrics, or a paragraph from a public-domain novel). Compute the perplexity on a held-out sentence for unigram, bigram, and trigram models. Plot perplexity versus n for $n = 1, 2, 3$. Does the curve decrease monotonically? If not, explain why.

Hint: For trigrams, you need to count sequences of three words. Use nested dictionaries or tuples as keys. Careful with the start-of-sentence context: a trigram model needs two `<s>` tokens at the beginning.

Exercise 3.6 (Programming – Intermediate). Implement Kneser-Ney smoothing for a bigram model. Use the toy corpus from Section 3.1.3 with discount $d = 0.75$. Compare the perplexity on the held-out sentence "the rug sat on the cat" under (a) MLE, (b) add-1 smoothing, and (c) Kneser-Ney smoothing. Report all three perplexity values and discuss which unseen bigrams benefit most from the continuation probability.

Hint: The normalization constant is $\lambda(w_{t-1}) = d \cdot |\{w : C(w_{t-1}, w) > 0\}|/C(w_{t-1})$. For the test bigram "rug sat," both "rug" and "sat" have non-trivial continuation counts, so Kneser-Ney will assign a higher probability than add-1.

Exercise 3.7 (Programming – Intermediate). Write a text generator that produces sentences from both a bigram and a trigram model trained on the same corpus. Generate 10 sentences from each model. Rate each sentence on a 1–5 fluency scale (1 = nonsensical, 5 = grammatically perfect and meaningful). Compute the average fluency for each model. Is the trigram model consistently more fluent?

Hint: Start from the `<s>` token (or `<s> <s>` for trigrams), sample the next word from $P(w_t \mid \text{context})$, and stop when you generate `</s>` or reach a maximum of 20 words. Use `numpy.random.choice` with the probability array.

Exercise 3.8 (Programming – Advanced). Visualize the bigram count matrix as a heatmap for the 40 most frequent words in a corpus of at least 50 sentences. Use a logarithmic color scale. Compute the fraction of zero cells in the matrix. Then fix a frequent word as w_{t-2} and visualize the resulting trigram count matrix $C(w_{t-2}, w_{t-1}, w_t)$ as a heatmap over the same 40 words. Compare the sparsity of the bigram and trigram matrices and discuss what this implies about the curse of dimensionality.

Hint: Use `matplotlib.pyplot.imshow` with `norm=matplotlib.colors.LogNorm()`. Most cells will be zero, especially in the trigram matrix. You may need to add 0.1 to zero cells before taking the log to avoid errors.

Exercise 3.9 (Programming – Advanced). Interpolated smoothing requires three data splits: train (to build n-gram counts), validation (to tune weights), and test (to report final perplexity). With small training corpora, the optimal weight distribution can shift dramatically – the trigram weight does not always dominate.

Implement interpolated smoothing by combining unigram, bigram, and trigram models with weights $\lambda_1, \lambda_2, \lambda_3$ (constrained so $\lambda_1 + \lambda_2 + \lambda_3 = 1$). Use a grid search in increments of 0.1 over the validation set. Then repeat with the training corpus reduced to half its original size. Fill in the following table with your results:

	λ_1	λ_2	λ_3	PP (validation)	PP (test)
Full training set					
Half training set					

Discuss: in which direction do the optimal weights shift when data is scarce, and why?