

The bigram model: your first real predictor

A table of counts can generate text. It does not need attention, a hidden layer, or a training cluster. It needs examples of which token followed which other token. This chapter builds that table, turns it into a predictor, and tests whether it learned anything useful. Then we identify a limitation that no amount of extra counting can remove.

The reading-and-lab budget above is a rough study allowance, not a measured completion time. Take longer if you need to work through the probabilities yourself. The executable lab is deliberately small; most of the learning comes from explaining its behavior.

Why this matters

In the tokens chapter, text became integer IDs. In the embeddings chapter, an ID selected a row of numbers. Neither operation alone tells us what should come next. Today we give a row a specific job: represent a probability distribution over the next token. The input token chooses the row, and the row tells us how likely each output token is.

This is the smallest useful version of the prediction loop inside an autoregressive language model. We can inspect every count, every probability, and every decision. When later chapters replace the table with a transformer, the interface will remain recognizable: context in, next-token scores out, then a choice about how to use them.

A simple model also gives you a baseline. Suppose a much more complicated training pipeline produces attractive text. Did the model learn predictive structure, or did you select a lucky sample? Today we answer with a held-out score and simpler competing predictors. That habit transfers directly to AI engineering: demonstrate an improvement against an appropriate baseline rather than trusting a polished demonstration.

There is a useful caution for agent work, too. Text that resembles familiar language is not necessarily a correct answer, and a predictive score is not an authorization decision. Our generated sample will contain recognizable fragments while failing to maintain meaning. That is a concrete reason to separate generation quality from task correctness. We are not building an agent or a security test here. We are learning how to recognize what a model has and has not demonstrated.

The idea in plain words

Imagine reading the string `abacaba` and making a tally every time a letter has a following letter. After `a`, you see `b` twice and `c` once. After `b`, you see `a` twice. After `c`, you see `a` once. The final `a` has no following letter, so it adds no pair.

Ask what should follow a new `a`. A reasonable first estimate is to reproduce the observed proportions: `b` gets two out of three, and `c` gets one out of three. This is not a rule somebody wrote saying that a particular letter must follow another. It is an estimate fitted from examples. Counting is already a form of learning.

Now name the object: a bigram is a pair of adjacent tokens. A bigram language model estimates the next token using only the immediately preceding token. In this lab, tokens are characters, so the current character selects a row of possible next characters. The same design could use words or subword tokens instead. Character tokenization is a teaching choice, not part of the definition of a bigram.

There is an obvious danger in the raw estimate. We never observed `a` followed by `a`, but that does not prove it is impossible. We will add a small positive count to every possibility before normalizing. With add-one smoothing, the row for `a` becomes counts `[1, 3, 2]`, with total `6`. The probabilities are therefore `[0.166667, 0.5, 0.333333]` after rounding, exactly as the lab prints.

To generate text, start with a character, choose the next character according to its row, then use that new character to select the next row. Repeat. Prediction gives the distribution; sampling chooses one outcome. They are different operations, and neither changes the fitted counts.

How it actually works

Part 1: define the training events

Our vocabulary is sorted, so the toy IDs are `a = 0`, `b = 1`, and `c = 2`. An ID is an address, not a numeric meaning. If we consistently renamed every ID, the model's behavior in text space would be unchanged.

Walk the toy string with overlapping pairs: `ab`, `ba`, `ac`, `ca`, `ab`, `ba`. The lab confirms `7` tokens and `6` transitions. Every position except the last supplies an input with an observed target. Do not pair the final token with the first unless you deliberately want a circular training sequence. Our text is not circular.

Let `C[i, j]` mean the count of occasions where token `i` was followed by token `j`. The first index selects the current-token row. The second selects the next-token column. The resulting matrix is:

```
counts after a: [0, 2, 1]
counts after b: [2, 0, 0]
counts after c: [1, 0, 0]
```

For independent documents, counting must respect boundaries. Our reusable function accepts a list of sequences and counts within each one. It does not silently create a pair from the end of one document to the start of another. A different legitimate design would insert explicit beginning and ending tokens and learn those transitions. Neither design should be accidental.

The Shakespeare portion treats each selected text slice as one continuous sequence. Newlines, punctuation, and spaces are ordinary characters. A period is not an automatic stop instruction: stopping is controlled by the generator's requested length. This distinction matters because data boundaries, vocabulary symbols, and generation stopping rules are separate choices.

Part 2: counts become conditional probabilities

For a row with observed successors, the unsmoothed estimate is:

$$P(j \mid i) = C[i, j] / \text{sum_k } C[i, k].$$

Read $P(j \mid i)$ as "the probability of next token `j`, given current token `i`." The vertical bar means "given," not division. The denominator adds the counts across all candidate next tokens `k` in the current row. It

does not add the entire matrix, and it does not normalize a column.

For the toy `a` row, the raw probability of `b` is `2 / 3`, printed as `0.666667`. Dividing by the total number of transitions would answer a different question: how common is this pair among all pairs? We need the distribution after this particular current token, so we divide only by that row's total.

Add-alpha smoothing changes the estimate to:

```
P_alpha(j | i) = (C[i, j] + alpha) / (sum_k C[i, k] + alpha * V).
```

Here `alpha` is a nonnegative amount added to every cell, and `V` is the vocabulary size, which is also the number of candidate successors. Adding to every cell means adding `alpha * V` to the denominator. Forgetting that extra denominator mass produces numbers that no longer sum to a valid distribution.

Work the add-one example by hand. The original `a` row has total `3`. The vocabulary has `3` entries. The denominator becomes `3 + 3 = 6`. The `b` numerator becomes its observed count plus the pseudocount, so its probability becomes `3 / 6 = 0.5`. The lab also prints the other rows: after `b`, probabilities are `[0.6, 0.2, 0.2]`; after `c`, they are `[0.5, 0.25, 0.25]`.

Smoothing does not create evidence that the missing pairs really occurred. It is a modeling choice that reserves probability for unseen possibilities. More smoothing pulls each row toward a uniform distribution. A completely unobserved context becomes uniform when alpha is positive. With no smoothing and an empty row, the denominator is zero, so our code raises a clear error instead of hiding an invalid model behind `NaN` values.

The name Laplace smoothing refers to the add-one case. Smaller positive values are often useful too. Jurafsky and Martin's language-model chapter explains why sparse observations require smoothing and why more sophisticated methods exist. We will compare a small set of smoothing choices on validation data rather than declare one value universally best.

Part 3: give the predictor an honest score

A generated sample is not a sufficient evaluation. Instead, give the model a sequence it did not fit and inspect the probabilities it assigned to the actual next tokens. A good predictor consistently puts more probability on the targets that occur.

Consider `aba`, with the initial `a` supplied. Its predicted suffix contains transitions from `a` to `b` and from `b` to `a`. Under the smoothed toy model, the conditional probability of that suffix is `0.500000 * 0.600000 = 0.300000`. We are not claiming a probability for the initial token because this model has no initial-token distribution.

Multiplying probabilities across long texts creates extremely small values. Logs turn that product into a sum. We use the negative natural logarithm so that more probable correct targets receive smaller positive penalties. Averaging makes scores comparable across evaluation sets with different numbers of predicted tokens, provided the tokenization and evaluation convention remain the same.

Write the score as `L = -(1 / M) * sum_t ln P(x_next | x_current)`. `L` is the mean loss, `M` is the number of scored transitions, `t` indexes those transitions, and `ln` is the natural logarithm. Each `x_current` is the observed input token, and each `x_next` is its actual successor. The minus sign reverses the sign of the logarithm. With natural logs, the units are nats per predicted token.

For `aba`, take the negative log of `0.300000` and divide by the number of transitions. The measured mean negative log likelihood is `0.601986`. Perplexity is $\exp(L)$, the exponential of the mean loss, here `1.825742`. It is a transformed predictive score, sometimes interpreted as an effective branching factor. It is not the fraction of wrong characters and does not literally count how many choices the model considered at every step.

If the model assigns zero probability to an observed target, its negative log penalty is infinite. The lab demonstrates that on `abc` without smoothing: `b` to `c` was never observed. Add-one smoothing makes the loss finite, `1.151293`. That score is worse than the toy uniform score `1.098612`, reminding us that a fitted model is not guaranteed to beat a simpler model on every short sequence.

Part 4: choose outcomes without changing the model

Sampling means drawing a token from the current row with the listed probabilities. In the toy `a` row, repeated independent draws should approach the row's proportions. The lab makes `20000` such draws and measures `[0.164, 0.5044, 0.3316]`. Those frequencies are close to, not exactly equal to, the probabilities. Finite random samples fluctuate.

Generating a sequence is different from repeatedly testing a fixed row. After each generated character, the state changes to that character. If the model picks `b`, the next draw comes from the `b` row, not the old `a` row. This is the entire autoregressive feedback loop at this scale.

Greedy decoding always chooses the largest probability in a row. The toy greedy output is `ababababababa`: it keeps taking the favorite transition and cycles. Sampling sometimes chooses less probable alternatives and produces a different path. Neither is "more trained." They are decoding policies applied to the same fitted model.

We use an explicit CPU random generator and seed so the run is repeatable in the installed environment. A seed is not a portable promise that every library version or device will emit identical characters. The behavioral invariant is the sampled distribution and reproducibility within the same execution setup, not the literary quality of one output string.

Part 5: separate fitting, tuning, and testing

The lab reads the local Tiny Shakespeare dataset. Counts and vocabulary come only from the first `200000` characters. The next `20000` form the validation slice; the final `20000` form the test slice. Their offsets are printed, and an assertion checks that validation and test do not overlap. No bigrams are counted across the split boundaries.

The training slice contains `62` distinct characters, so this model has `3844` table cells. That differs from a tokenizer built from the complete dataset: we deliberately do not inspect future text to decide our vocabulary. The existing character tokenizer raises an error for unknown characters. In this particular run, both evaluation slices are encodable with the training vocabulary. Smoothing helps unseen pairs among known tokens; it cannot invent an ID for a wholly unknown character.

Validation compares alpha values `0.01`, `0.1`, `1`, and `10`. Their losses are `2.550121`, `2.524522`, `2.510559`, and `2.568968`, respectively. We select alpha `1` because it has the lowest validation loss among these candidates. We then report the untouched test slice once. This is a small illustrative model-selection exercise, not an exhaustive search or an industrial benchmark.

The uniform baseline ignores everything and assigns equal probability to each vocabulary entry. The unigram baseline learns how common the target characters are but ignores the current character. For a fair training-event comparison, its counts come from the same target positions used by the bigram model, with add-one smoothing. Only the bigram conditions its distribution on the previous character.

Measured test losses are 4.127134 for uniform, 3.389097 for unigram, and 2.629084 for bigram. Their perplexities are 62.000000, 29.639168, and 13.861062. The comparison demonstrates that this table learned useful adjacent-character information on these slices. It does not establish performance on arbitrary language, code, or agent tasks.

Part 6: bridge back to embeddings

An embedding lookup simply retrieves a row. We can therefore store one row of next-token logits for each current-token ID. Unlike a general feature embedding, each column now has a defined meaning: it is a score for a particular next token. This table is vocabulary-by-vocabulary rather than vocabulary-by-a-separately-chosen-feature-width.

A logit is a real-valued score. Softmax exponentiates the scores and divides each by the sum of exponentials in the same row. If we set the logits to $\log(P)$ for a positive probability row, exponentiation recovers P , whose sum is already one. Softmax therefore recovers the original distribution. The lab verifies this numerically and gives the toy table shape (3, 3) and the prediction-batch shape (2, 3).

PyTorch cross entropy expects logits, not probabilities, as its prediction input. Giving it our log-probability rows returns 0.601986, matching the direct negative-log-likelihood calculation. Passing the probability rows themselves would instead apply softmax to those numbers as scores, changing the distribution and the loss.

We freeze this demonstration table. No optimizer runs and no gradient training happens in this chapter. A neural bigram could instead learn its logits by gradient descent. Changing the fitting method would not remove the context restriction: a row selected solely by the last token still ignores everything before it. Representation, optimization, and context length are separate design decisions.

Build it

Run this from the project root:

```
env -u PYTHONPATH .venv/bin/python chapters/llm-fundamentals/03-the-bigram-model/lab.py
```

The script locates `labs/` relative to its own file, so it also works from another working directory when called by absolute path. It uses CPU tensors, the installed PyTorch package, the existing character tokenizer, and the checked-in text data. There is no network request, model download, API key, or hosted generation.

The parts in the file match the numbered mechanism sections above. Start with Part 1's explicit loop:

```
manual_counts = torch.zeros_like(toy_counts)
for current, following in zip(toy_ids, toy_ids[1:]):
    manual_counts[current, following] += 1
```

The slice starts one position later than the original list. `zip` pairs corresponding positions and stops when the shorter list ends. Thus each input is paired with the next token, not itself. The matrix starts at zero and each event increments exactly one cell. The script checks this simple loop against the reusable implementation.

The shared `labs/tutorlab/bigram_counts.py` uses a vectorized counter for the longer text. A pair of indices is converted to one flat index by `current * vocab_size + following`. `torch.bincount` accumulates all occurrences, including repeated pairs, and the result is reshaped into the square table. This avoids a slow Python loop over the full corpus without changing what is counted.

Normalization is intentionally easy to inspect:

```
weights = counts + alpha
totals = weights.sum(dim=1, keepdim=True)
return weights / totals
```

`dim=1` sums across the successor columns. `keepdim=True` leaves a column of row totals, so broadcasting divides each row by its own denominator. Dropping that dimension would produce a vector aligned with the trailing dimension and could silently divide columns by unrelated totals. A plausible-looking matrix is not proof that you normalized the intended conditional distribution.

In scoring, `probs[ids[:-1], ids[1:]]` uses paired advanced indices. It selects one probability for each observed transition. Taking `log`, negating, concatenating across independent sequences, and averaging implements the formula above. Concatenation before averaging weights every transition equally rather than giving a short document the same weight as a long document.

The generation loop looks up a row, samples an integer ID with `torch.multinomial`, appends it, and repeats. The full module checks valid ranges, nonnegative finite probabilities, normalized rows, and requested output length. Those guards make a broken model fail explicitly rather than emit misleading samples.

Here is an exact excerpt from `build/lab-output.txt`:

```
selected alpha=1 using validation only
uniform train NLL=4.127134; test NLL=4.127134; test perplexity=62.000000
unigram train NLL=3.299755; test NLL=3.389097; test perplexity=29.639168
bigram train NLL=2.416625; test NLL=2.629084; test perplexity=13.861062
smoothed table row-sum max error=4.441e-16
```

The tiny row-sum error is floating-point rounding, not lost probability mass that matters to this demonstration. We use tolerant numerical comparisons rather than requiring every computed sum to equal a decimal ideal bit-for-bit.

Read the generated Shakespeare sample in the output file. You can recognize text fragments, line breaks, and occasional words, but the sequence does not maintain grammar or meaning. Do not select a nicer sample and call that an evaluation. The fixed held-out score is the evidence for prediction quality; the sample shows the character of the errors.

The final `check()` function tests the hand-counted matrix, transition totals, the smoothing example, normalized rows, the hand-derived loss, the infinite unsmoothed loss, seeded generation, sampling

frequencies, split separation, baseline ordering, and the cross-entropy equivalence. The dedicated tests also cover invalid IDs, malformed matrices, empty-context handling, and false cross-document pairs. Successful execution ends with `CHECK PASSED` only after the assertions run.

Try the predict-first changes at the bottom of the lab. Change smoothing and predict what happens to the favorite successor. Change only the random seed and predict whether the deterministic test loss changes. Compare separate sequences with one joined sequence and identify the extra boundary transition. If an assertion fails after an intentional experiment, identify which original hypothesis you changed; do not delete the check just to restore a green result.

Common mistakes and misconceptions

"A bigram sees the previous two tokens." It models a pair consisting of the current token and the next token. Its conditioning history contains only the current token. The lab checks that contexts `ab` and `cb` select the same final `b` row. The earlier character is unavailable to the predictor, even though it is still present in the Python output list.

"Counting is not learning." The fitted conditional distribution changes when the training data changes. That is learning from observations. Gradient descent is another way to fit parameters, not the definition of learning. In this simple model, direct counts make the fitted relationship especially transparent.

"Smoothing means missing pairs become equally likely to observed pairs." We add the same pseudocount to every cell; we do not erase the observed counts. Observed frequencies still affect the row. Extremely strong smoothing can drown out those differences, which is why the strength should be evaluated rather than assumed harmless.

"A valid probability row must make good predictions." Normalization is necessary for our probability interpretation, but uniform rows normalize perfectly and can be weak predictors. Numerical correctness and predictive quality require different tests. Our row-sum checks establish the former; held-out comparison establishes a bounded claim about the latter.

"The sampling function requires normalized input." PyTorch's `torch.multinomial` also accepts unnormalized nonnegative finite weights with nonzero row sums, according to its documentation. We normalize because we also want probabilities for scoring and interpretation. The specific API is not used by every language-model deployment; it is a concrete implementation of a broadly useful sampling operation.

"A lower training loss proves generalization." Training loss describes data that influenced the fit. Validation helps choose settings; test measures the selected system on held-out data. If you repeatedly adjust settings after inspecting test results, that test is no longer an untouched evaluation. The clean split is part of the experiment, not bookkeeping to add afterward.

"More data will give this model long memory." More data can improve estimates of the same conditional rows. It cannot make contexts ending in the same token produce different rows. To exploit earlier tokens, we must change the conditioning representation, not just increase the number of observations or rename the table a neural network.

In the real world

As of 2026-10-10, the linked PyTorch documentation confirms the APIs used here: `nn.Embedding` retrieves rows, `torch.multinomial` draws indices from weights, and cross entropy consumes logits with class-index targets. These are API facts, not claims about a specific frontier model's private implementation. We exercise the relevant behavior locally rather than relying only on descriptions.

Karpathy's makemore repository explicitly presents an autoregressive character-level progression from bigrams to more expressive models. Its bigram description is a lookup table of counts. That progression is useful because generation stays understandable while the mechanism producing the distribution changes. Our examples use a small artificial string and the project's local Shakespeare data, not a claim to reproduce his dataset scores.

The project already contains a later transformer implementation in `labs/tutorlab/gpt.py`. Its `generate` method obtains logits from a context window, selects the final position, converts scores into probabilities, draws a token, and appends it. That is the same outer loop we just built. The crucial difference is that the transformer can use a longer context to compute those logits. We do not train or benchmark that transformer in this chapter.

A dense bigram table also exposes a scaling cost: every current token needs a score for every possible successor. Its storage grows as vocabulary size squared. Our character vocabulary makes that manageable. A large subword vocabulary makes a dense table much less attractive, while still leaving the last-token-only restriction intact. Bigger token pieces do not automatically solve the need to reason across a sequence.

For engineering work, retain the evaluation pattern: freeze a fitting dataset, choose settings on validation data, compare sensible baselines, and make claims only about the measured scope. A language-model loss is useful evidence about prediction. It is not a substitute for testing factual answers, tool outcomes, or access controls in a deployed assistant.

Go deeper

The moments in `lectures.json` were verified on 2026-10-10 using the creators' published chapter markers because YouTube blocked transcript retrieval from this host. Their anchors are marker titles, not purported verbatim spoken quotations. The full gate checks those anchors again.

- Andrej Karpathy, "The spelled-out intro to language modeling: building makemore," at [12:45](#). The published marker is "counting bigrams in a 2D torch tensor (\"training the model\")." Watch how pair counting becomes a table, and compare the indexing with our explicit loop. Pause when a row is interpreted and explain its denominator before continuing.
- Andrej Karpathy, "Let's build GPT: from scratch, in code, spelled out.," at [22:11](#). The marker is "simplest baseline: bigram language model, loss, generation." Watch the connection between table lookup, logits, loss, and generation. Notice which pieces remain when the model later gains context.
- Jurafsky and Martin, [N-gram Language Models](#), gives the formal treatment of conditional probabilities, evaluation, and smoothing. Read it after the hand calculation so the notation labels a mechanism you already understand.

A productive extension is not immediately to make the table larger. First construct contexts sharing the same final token but requiring different continuations. Explain why the present representation must

collapse them. That question prepares you for the next chapter's attention mechanism without pretending attention is just a larger count matrix.

Explain it back

Explain to someone new to modeling how the string `abacaba` becomes a next-character predictor, why unseen pairs need probability mass, how held-out loss differs from a generated sample, and why two contexts ending in the same character receive the same prediction. Use the row for `a` as your concrete example. Dictation is fine; say the mechanism in your own words rather than reading the formulas.

Your study partner will check one gap at a time, then give you a new application. Reading, running the lab, or receiving this chapter is not evidence of mastery. Mastery needs both an accurate explanation and a correct application to a fresh case.

Check yourself

`quiz.json` contains the answer key, explanations, and explain-back rubrics. Predict before opening it. The questions are:

1. What information conditions a bigram's next-token prediction?
2. Which denominator normalizes a smoothed row?
3. What is the consequence of a zero-probability observed target?
4. Which split should select the smoothing strength?
5. Why can the same fitted model produce different samples?
6. What probabilities result from add-one smoothing the toy `a` row?
7. What pair is accidentally introduced by joining independent sequences?
8. What happens to the evaluation score if only the sampling seed changes?
9. Why can contexts with different prefixes receive identical predictions?
10. How do you score a fresh transition under a changed count row, and what does that score measure?

Sources

- KrypteiaSec Academy, chapter coverage baseline, read 2026-10-10: <https://krypteiasec.com/academy/llm-fundamentals/ch3>
- Prior-chapter continuity, Academy embeddings page, read 2026-10-10 because the local preceding chapter was not yet available: <https://krypteiasec.com/academy/llm-fundamentals/ch2>
- Karpathy, makemore, count-based bigram and autoregressive model progression: <https://github.com/karpathy/makemore>
- Karpathy, makemore lecture and creator-published timeline: <https://www.youtube.com/watch?v=PaCmpygFfXo>
- Karpathy, GPT lecture and creator-published timeline: <https://www.youtube.com/watch?v=kCc8FmEb1nY>
- Jurafsky and Martin, Speech and Language Processing, chapter 3, N-gram Language Models, retrieved 2026-10-10: <https://web.stanford.edu/~jurafsky/slp3/3.pdf>
- PyTorch, multinomial sampling input constraints, verified 2026-10-10: <https://docs.pytorch.org/docs/2.14/generated/torch.multinomial.html>

- PyTorch, embedding lookup and pretrained weights, verified 2026-10-10:
<https://docs.pytorch.org/docs/2.14/generated/torch.nn.Embedding.html>
- PyTorch, cross entropy, logits and class-index targets, verified 2026-10-10:
<https://docs.pytorch.org/docs/2.14/generated/torch.nn.CrossEntropyLoss.html>
- Tiny Shakespeare provenance, Karpathy's char-rnn repository: <https://github.com/karpathy/char-rnn>
- Local executable evidence: `lab.py`, `build/lab-output.txt`, `labs/tutorlab/bigram_counts.py`, and `labs/tests/test_bigram_counts.py`. Every measured result quoted above comes from this lab run. The later-model code comparison refers specifically to the local `labs/tutorlab/gpt.py`, not an unverified external deployment.