"Andriy's long-awaited sequel in his "The Hundred-Page" series of machine learning textbooks is a masterpiece of concision."

— **Bob van Luij**t, CEO and Co-Founder of Weaviate

"Andriy has this almost supernatural talent for shrinking epic AI concepts down to bite-sized, 'Ah, now I get it!' moments."

— **Jorge Torres**, CEO at MindsDB

"Andriy paints for us, in 100 marvelous strokes, the journey from linear algebra basics to the implementation of transformers."

— **Florian Douetteau**, Co-founder and CEO at Dataiku

"Andriy's book is an incredibly concise, clear, and accessible introduction to machine learning."

— **Andre Zayarni**, Co-founder and CEO at Qdrant

"This is one of the most comprehensive yet concise handbooks out there for truly understanding how LLMs work under the hood."

— **Jerry Liu**, Co-founder and CEO at LlamaIndex

Featuring a foreword by **Tomáš Mikolov** and back cover text by **Vint Cerf**

The Hundred-Page Language Models Book

Andriy Burkov

ISBN 978-1-7780427-2-0

Publisher: True Positive Inc.

To my family, with love

“Language is the source of misunderstandings.”
—Antoine de Saint-Exupéry, *The Little Prince*

“In mathematics you don't understand things. You just get used to them.”
—John von Neumann

“Computers are useless. They can only give you answers.”
— Pablo Picasso

The book is distributed on the “read first, buy later” principle

Contents

Chapter 2. Language Modeling Basics

Language modeling requires transforming text into numbers that computers can process. In this chapter, we'll explore how to convert words and documents into numerical formats, introduce the fundamentals of language modeling, and study count-based models as our first architecture. Finally, we'll cover techniques for measuring language model performance.

Let's begin with one of the oldest yet effective techniques for converting text into usable data for machine learning: bag of words.

2.1. Bag of Words

Suppose you have a collection of documents and want to predict the main topic of each one. When topics are defined in advance, this task is called **classification**. With only two possible topics, it's known as **binary classification**, as explained in Section 1.7. With more than two topics, we have **multiclass classification**.

In multiclass classification, the dataset consists of pairs $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, where $y_i \in \{1, \dots, C\}$, N represents the number of examples, and C denotes the number of possible classes. Each $\mathbf{x}_i$ could be a text document, with y_i being an integer indicating its topic—for example, 1 for "music," 2 for "science," or 3 for "cinema."

Machines don't process text like humans. To use machine learning on text, we first convert documents into numbers. Each document becomes a **feature vector**, where each feature is a **scalar**.

A common and effective approach to convert a collection of documents into feature vectors is the **bag of words** (**BoW**). Here's how it works for a collection of 10 simple documents:

ID	Text
1	Movies are fun for everyone.
2	Watching movies is great fun.
3	Enjoy a great movie today.
4	Research is interesting and important.
5	Learning math is very important.
6	Science discovery is interesting.
7	Rock is great to listen to.
8	Listen to music for fun.
9	Music is fun for everyone.
10	Listen to folk music!

A collection of text documents used in machine learning is called a **corpus**. The bag of words method applied to a corpus involves two key steps:

1. **Create a vocabulary**: List all unique words in the corpus to create the **vocabulary**.
2. **Vectorize documents**: Convert each document into a feature vector, where each dimension represents a word from the vocabulary. The value indicates the word's presence, absence, or frequency in the document.

For the 10-document corpus, the vocabulary is built by listing all unique words in alphabetical order. This involves removing punctuation, converting words to lowercase, and eliminating duplicates. After processing, we get:

```
vocabulary = ["a", "and", "are", "discovery", "enjoy", "everyone", "folk", "f
or", "fun", "great", "important", "interesting", "is", "learning", "listen",
"math", "movie", "movies", "music", "research", "rock", "science", "to", "tod
ay", "very", "watching"]
```

Splitting a document into small indivisible parts is called **tokenization**, and each part is a **token**. There are different ways to tokenize. We tokenized our 10-document corpus by words. Sometimes, it's useful to break words into smaller units, called **subwords,** to keep the vocabulary size manageable. For instance, instead of including "interesting" in the vocabulary, we might split it into "interest" and "-ing." One method for subword tokenization, which we'll cover in this chapter, is byte-pair encoding. The choice of tokenization method depends on the language, dataset, and model, and the best one is found experimentally.

> A count of all English word **surface forms**—like *do*, *does*, *doing*, and *did*—reveals several million possibilities. Languages with more complex morphology have even greater numbers. A Finnish noun alone can take 2,000–3,000 different forms to express various case and number combinations. Using subwords offers a practical solution, as storing every surface form in the vocabulary would consume excessive memory and computational resources.

Words are a type of token, so "token" and "word" are often used interchangeably as the smallest indivisible units of a document. In this book, when a distinction is important, context will make it clear. While the bag-of-words approach can handle both words and subwords, it was originally designed for words—hence the name.

Feature vectors can be organized into a **document-term matrix** (DTM). Here, rows represent documents, and columns represent tokens. Below is a partial document-term matrix for a 10-document corpus. It includes only a subset of tokens to fit within the page width:

Doc	a	and	...	fun	...	listen	math	...	science	...	watching
1	0	0	...	1	...	0	0	...	0	...	0
2	0	0	...	1	...	0	0	...	0	...	1
3	1	0	...	0	...	0	0	...	0	...	0
4	0	1	...	0	...	0	0	...	0	...	0
5	0	0	...	0	...	0	1	...	0	...	0
6	0	0	...	0	...	0	0	...	1	...	0
7	0	0	...	0	...	1	0	...	0	...	0
8	0	0	...	1	...	1	0	...	0	...	0
9	0	0	...	1	...	0	0	...	0	...	0
10	0	0	...	0	...	1	0	...	0	...	0

In the DTM above, 1 means the token appears in the document, while 0 means it does not. For instance, the feature vector $\mathbf{x}_2$ for document 2 (*"Watching movies is great fun."*) is:

$$\mathbf{x}_2 = [0,0,0,0,0,0,0,0,1,1,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1]^\top.$$

In natural languages, word frequencies follow **Zipf's Law,** stating that a word's frequency is inversely proportional to its rank in the frequency table—for instance, the second most frequent word appears half as often as the most frequent one. Consequently, document-term matrices are usually **sparse,** containing mostly zeros.

A neural network can be trained to predict a document's topic using these feature vectors. Let's do that. The first step is to assign labels to the documents, a process known as **labeling**. Labeling can be done manually or assisted by an algorithm. When algorithms are used, human validation is often needed to confirm accuracy. Here, we will manually label the documents by reading each one and choosing the most suitable topic from the three options.

Doc	Text	Class ID	Class Name
1	Movies are fun for everyone.	1	Cinema
2	Watching movies is great fun.	1	Cinema
3	Enjoy a great movie today.	1	Cinema
4	Research is interesting and important.	3	Science
5	Learning math is very important.	3	Science
6	Science discovery is interesting.	3	Science
7	Rock is great to listen to.	2	Music
8	Listen to music for fun.	2	Music
9	Music is fun for everyone.	2	Music
10	Listen to folk music!	2	Music

Advanced **chat language models** enable highly accurate automated document labeling through a panel of expert models. Using three LLMs, when two or more assign the same label to a document, that label is adopted. If all three disagree, either a human can decide, or a fourth model can break the tie. In many business contexts, manual labeling is becoming obsolete, as LLMs offer faster and often more reliable labeling.

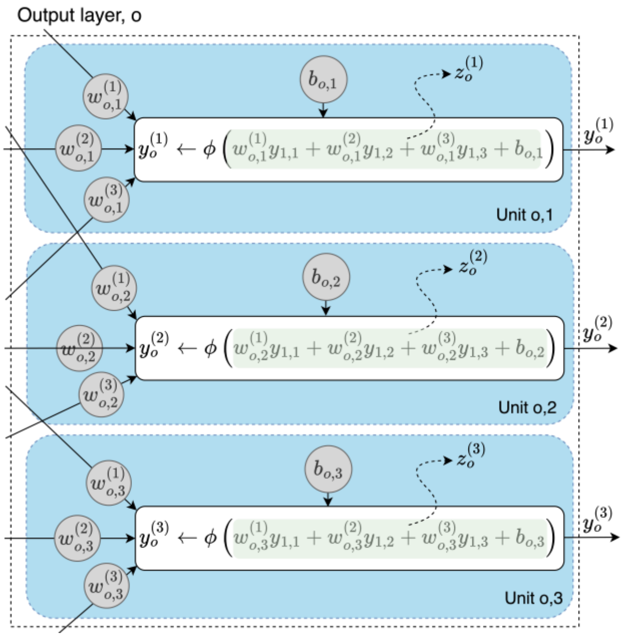
We have three classes: 1 for cinema, 2 for music, and 3 for science.[3] While binary classifiers typically use the **sigmoid** activation function with the **binary cross-entropy** loss, as discussed in Section 1.7, tasks involving three or more classes generally employ the softmax activation function paired with the cross-entropy loss.

The **softmax** function is defined as:

[3] Class labels in classification are arbitrary and unordered. You can assign numbers to the classes in any way, and the model's performance won't change as long as the mapping is consistent for all examples.

$$\text{softmax}(\mathbf{z}, k) \overset{\text{def}}{=} \frac{e^{z^{(k)}}}{\sum_{j=1}^{D} e^{z^{(j)}}}$$

Here, $\mathbf{z}$ is a D-dimensional vector of logits, k is the index for which the softmax is computed, and e is **Euler's number**. **Logits** are the raw outputs of a neural network, prior to applying an activation function, as shown below:



Output layer, o
Unit o,1
Unit o,2
Unit o,3
$y_o^{(1)} \leftarrow \phi\left(w_{o,1}^{(1)}y_{1,1} + w_{o,1}^{(2)}y_{1,2} + w_{o,1}^{(3)}y_{1,3} + b_{o,1}\right)$
$y_o^{(2)} \leftarrow \phi\left(w_{o,2}^{(1)}y_{1,1} + w_{o,2}^{(2)}y_{1,2} + w_{o,2}^{(3)}y_{1,3} + b_{o,2}\right)$
$y_o^{(3)} \leftarrow \phi\left(w_{o,3}^{(1)}y_{1,1} + w_{o,3}^{(2)}y_{1,2} + w_{o,3}^{(3)}y_{1,3} + b_{o,3}\right)$
$z_o^{(1)}$
$z_o^{(2)}$
$z_o^{(3)}$
$b_{o,1}$
$b_{o,2}$
$b_{o,3}$

The figure shows the output layer of a neural network, labeled as o. The logits $z_o^{(k)}$, for $k \in \{1,2,3\}$, are the values in light green. These represent the outputs of the units before the activation function is applied. The vector $\mathbf{z}$ is expressed as $\mathbf{z}_o = \left[z_o^{(1)}, z_o^{(2)}, z_o^{(3)}\right]^{\mathsf{T}}$.

For instance, the softmax for unit $o, 2$ in the figure is calculated as:

$$\text{softmax}(\mathbf{z}_o, 2) = \frac{e^{z_o^{(2)}}}{e^{z_o^{(1)}} + e^{z_o^{(2)}} + e^{z_o^{(3)}}}$$

Softmax transforms a vector into a **discrete probability distribution** (DPD), ensuring that $\sum_{k=1}^{D} \text{softmax}(\mathbf{z}, k) = 1$. A DPD assigns probabilities to values in a finite set, with their sum equaling 1. A **finite set** contains a countable number of distinct elements. For instance, in a classification task with classes 1, 2, and 3, these classes constitute a finite set. The softmax function maps each class to a probability, with these probabilities summing to 1.

Let's compute the probabilities step by step. Assume we have three logits, $\mathbf{z} = [2.0, 1.0, 0.5]^{\mathsf{T}}$, representing a document's classification into cinema, music, or science.

First, calculate $e^{z^{(k)}}$ for each logit:

$$
\begin{aligned}
e^{z^{(1)}} &= e^{2.0} &&\approx 7.39,\\
e^{z^{(2)}} &= e^{1.0} &&\approx 2.72,\\
e^{z^{(3)}} &= e^{0.5} &&\approx 1.65
\end{aligned}
$$

Next, sum these values: $\sum_{j=1}^{3} e^{z^{(j)}} = 7.39 + 2.72 + 1.65 \approx 11.76$.

Now use the softmax formula, $\text{softmax}(\mathbf{z}, k) = \frac{e^{z^{(k)}}}{\sum_{j=1}^{3} e^{z^{(j)}}}$, to compute the probabilities:

$$
\Pr(\text{cinema}) = \frac{7.39}{11.76} \approx 0.63, \quad \Pr(\text{music}) = \frac{2.72}{11.76} \approx 0.23, \quad \Pr(\text{science}) = \frac{1.65}{11.76} \approx 0.14
$$

> Neural network softmax outputs are better characterized as “probability scores” rather than true statistical probabilities, despite summing to one and resembling class likelihoods. Unlike logistic regression or Naïve Bayes models, neural networks don't generate genuine class probabilities. For simplicity, though, I'll refer to these probability scores as “probabilities” throughout this book.

The **cross-entropy** loss measures how well predicted probabilities match the true distribution. The true distribution is typically a **one-hot vector** with a single element equal to 1 (the correct class) and 0 elsewhere. For example, a one-hot encoding with 3 classes looks like:

Class	One-hot vector
1	$[1,0,0]^\top$
2	$[0,1,0]^\top$
3	$[0,0,1]^\top$

The cross-entropy loss for a single example is:

$$
\text{loss}(\tilde{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{C} y^{(k)} \log\left(\tilde{y}^{(k)}\right),
$$

where C is the number of classes, $\mathbf{y}$ is the one-hot encoded true label, and $\tilde{\mathbf{y}}$ is the predicted probabilities. Here, $y^{(k)}$ and $\tilde{y}^{(k)}$ represent the k^{th} elements of $\mathbf{y}$ and $\tilde{\mathbf{y}}$, respectively.

Since $\mathbf{y}$ is one-hot encoded, only the term corresponding to the correct class contributes to the summation. The summation thus simplifies by retaining only that single term. Let’s simplify it. Suppose the correct class is c, so $y^{(c)} = 1$ and $y^{(k)} = 0$ for all $k \neq c$. In the summation, only the term where $k = c$ will be non-zero. The equation simplifies to:

$$
\text{loss}(\tilde{\mathbf{y}}, \mathbf{y}) = -\log\left(\tilde{y}^{(c)}\right) \tag{2.1}
$$

This simplified form indicates that the loss corresponds to the negative logarithm of the probability assigned to the correct class. For N examples, the average loss is:

$$
\text{loss} = -\frac{1}{N}\sum_{i=1}^{N} \log\left(\hat{y}_i^{(c_i)}\right),
$$

where c_i is the correct class index for the i^{th} example.

When used with softmax in the output layer, cross-entropy loss guides the network to assign high probabilities to correct classes while reducing probabilities for incorrect ones.

For a document classification example with three classes (cinema, music, and science), the network generates three logits. These logits are passed through the softmax function to convert them into probabilities for each class. The cross-entropy loss is then calculated between these scores and the true one-hot encoded labels.

Let's illustrate this by training a simple two-layer neural network to classify documents into three classes. We first import dependencies, set a random seed, and define the dataset:

```
import re, torch, torch.nn as nn

torch.manual_seed(42) ➊

docs = [
    "Movies are fun for everyone.",
    "Watching movies is great fun.",
    ...
    "Listen to folk music!"
]

labels = [1, 1, 1, 3, 3, 3, 2, 2, 2, 2]
num_classes = len(set(labels))
```

Setting the random seed in line ➊ ensures consistent random number generation across PyTorch runs. This guarantees **reproducibility**, allowing you to attribute performance changes to code or hyperparameter modifications rather than random variations. Reproducibility is also essential for teamwork, enabling collaborators to examine issues under identical conditions.

Next, we convert documents into a bag of words using two methods: `tokenize`, which splits input text into lowercase words, and `get_vocabulary`, which constructs the vocabulary:

```
def tokenize(text):
    return re.findall(r"\w+", text.lower()) ➊

def get_vocabulary(texts):
    tokens = {token for text in texts for token in tokenize(text)} ➋
    return {word: idx for idx, word in enumerate(sorted(tokens))} ➌

vocabulary = get_vocabulary(docs)
```

In line ➊, the regular expression `\w+` extracts individual words from the text. A **regular expression** is a sequence of characters used to define a search pattern. The pattern `\w+` matches sequences of "word characters," such as letters, digits, and underscores.

The `findall` function from Python's `re` module applies the regular expression and returns a list of all matches in the input string. In this case, it extracts all words.

In line ❷, the corpus is converted into a set of tokens by iterating through each document and extracting words using the same regular expression. In line ❸, these tokens are sorted alphabetically and mapped to unique indices, forming a vocabulary.

Once the vocabulary is built, the next step is to define the feature extraction function that converts a document into a feature vector:

```
def doc_to_bow(doc, vocabulary):
    tokens = set(tokenize(doc))
    bow = [0] * len(vocabulary)
    for token in tokens:
        if token in vocabulary:
            bow[vocabulary[token]] = 1
    return bow
```

The `doc_to_bow` function takes a document string and a vocabulary and returns the bag-of-words representation of the document.

Now, let's transform our documents and labels into numbers:

```
vectors = torch.tensor(
    [doc_to_bow(doc, vocabulary) for doc in docs],
    dtype=torch.float32
)
labels = torch.tensor(labels, dtype=torch.long) - 1 ❶
```

The `vectors` tensor with shape `(10, 26)` represents 10 documents as rows and 26 vocabulary tokens as columns, while the `labels` tensor of shape `(10,)` contains the class label for each document. The labels use integer indices rather than one-hot encoding since PyTorch's cross-entropy loss function (`nn.CrossEntropyLoss`) expects this format.

Line ❶ uses `torch.long` to cast labels to 64-bit integers. The `-1` adjustment converts our original classes 1, 2, 3 to indices 0, 1, 2, which aligns with PyTorch's expectation that class indices begin at 0 for models and loss functions like `CrossEntropyLoss`.

PyTorch provides two APIs for model definition: the **sequential API** and the **module API**. While we used the straightforward `nn.Sequential` API to define our model in Section 1.8, we'll now explore building a multilayer perceptron using the more versatile `nn.Module` API:

```
input_dim = len(vocabulary)
hidden_dim = 50
output_dim = num_classes

class SimpleClassifier(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.fc1(x) ❶
        x = self.relu(x) ❷
        x = self.fc2(x) ❸
        return x

model = SimpleClassifier(input_dim, hidden_dim, output_dim)
```

The `SimpleClassifier` class implements a **feedforward neural network** with two layers. Its constructor defines the network components:

1. A fully connected layer, `self.fc1`, maps the input of size `input_dim` (equal to the vocabulary size) to 50 (`hidden_dim`) outputs.
2. A ReLU activation function introduces non-linearity.
3. A second fully connected layer, `self.fc2`, reduces the 50 intermediate outputs to `output_dim`, the number of unique labels.

The `forward` method describes the **forward pass**, where inputs flow through the layers:

- In line ❶, the input `x` of shape `(10, 26)` is passed to the first fully connected layer, transforming it to shape `(10, 50)`.
- In line ❷, output from this layer is fed through the ReLU activation function, keeping the shape `(10, 50)`.
- In line ❸, the result is sent to the second fully connected layer, reducing it from shape `(10, 50)` to `(10, 3)`, producing the model's final output with logits.

The `forward` method is called automatically when you pass input data to the model instance, like this: `model(input)`.

> While `SimpleClassifier` omits a final softmax layer, this is intentional—PyTorch's `CrossEntropyLoss` combines softmax and cross-entropy loss internally for stability. This design eliminates the need for an explicit softmax in the model's forward pass.

With our model defined, the next steps, as outlined in Section 1.8, are to define the loss function, choose the gradient descent algorithm, and set up the training loop:

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

for step in range(3000):
    optimizer.zero_grad()
    loss = criterion(model(vectors), labels)
    loss.backward()
    optimizer.step()
```

As you can see, the training loop is identical to the one in Section 1.8. Once the training is complete, we can test the model on a new document:

```
new_docs = [
    "Listening to rock music is fun.",
    "I love science very much."
]
class_names = ["Cinema", "Music", "Science"]

new_doc_vectors = torch.tensor(
    [doc_to_bow(new_doc, vocabulary) for new_doc in new_docs],
    dtype=torch.float32
)

with torch.no_grad(): ❶
    outputs = model(new_doc_vectors) ❷
    predicted_ids = torch.argmax(outputs, dim=1) + 1 ❸

for i, new_doc in enumerate(new_docs):
    print(f'{new_doc}: {class_names[predicted_ids[i].item() - 1]}')
```

Output:

```
Listening to rock is fun.: Music
I love scientific research.: Science
```

The `torch.no_grad()` statement in line ❶ disables the default gradient tracking. While gradients are essential during **training** to update model parameters, they're unnecessary during **testing** or **inference**. Since these phases don't involve parameter updates, disabling gradient tracking conserves memory and speeds up computation. Note that the terms "testing," "inference," and "evaluation" are often used interchangeably when referring to generating predictions on unseen data.

In line ❷, the model processes all inputs simultaneously during inference, just as it does during training. This parallel processing approach leverages vectorized operations, substantially reducing computation time compared to processing inputs one by one.

We only care about the final label, not the logits returned by the model. In line ❸, `torch.argmax` identifies the highest logit's index, corresponding to the predicted class. Adding 1 compensates for the earlier shift from 1-based to 0-based indexing.

While the bag-of-words approach offers simplicity and practicality, it has notable limitations. Most significantly, it fails to capture token order or context. Consider how "the cat chased the dog" and "the dog chased the cat" yield identical representations, despite conveying opposite meanings.

N-grams provide one solution to this challenge. An n-gram consists of *n* consecutive tokens from text. Consider the sentence "Movies are fun for everyone"—its bigrams (2-grams) include "Movies are," "are fun," "fun for," and "for everyone." By preserving sequences of tokens, n-grams retain contextual information that individual tokens cannot capture.

However, using n-grams comes at a cost. The vocabulary expands considerably, increasing the computational cost of model training. Additionally, the model requires larger datasets to effectively learn weights for the expanded set of possible n-grams.

Another limitation of bag-of-words is how it handles out-of-vocabulary words. When a word appears during inference that wasn't present during training—and thus isn't in the vocabulary—it can't be represented in the feature vector. Similarly, the approach struggles with synonyms and near-synonyms. Words like "movie" and "film" are processed as completely distinct terms, forcing the model to learn separate parameters for each. Since labeled data is often costly to obtain, resulting in rather small labeled datasets, it would be more efficient if the model could recognize and collectively process words with similar meanings.

Word embeddings address this by mapping semantically similar words to similar vectors.

2.2. Word Embeddings

Consider document 3 ("Enjoy a great movie today.") from earlier. We can break down this bag of words (BoW) into one-hot vectors representing individual words:

BoW	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
enjoy	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
great	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
movie	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
today	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

As we see, a bag-of-word vector of a document is a sum of one-hot vectors of its words. Now, let's examine the one-hot vectors and the BoW vector for the text "Films are my passion.":

BoW	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
films	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
are	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
my	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
passion	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

There are two key problems here. First, even when a word exists in the training data and vocabulary, one-hot encoding reduces it to a single 1 in a vector of zeros, giving the classifier almost no meaningful information to learn from.

Second, in the above document, most one-hot encoded word vectors add no value since three out of four become **zero vectors**—representing words missing from the vocabulary.

A better approach would let the model understand that "films," though unseen in training, shares semantic meaning with "movies." This would allow the feature vector for "films" to be processed similarly to "movies." Such an approach requires word representations that capture semantic relationships between words.

Word embeddings overcome the limitations of the bag-of-words model by representing words as **dense vectors** rather than **sparse** one-hot vectors. These lower-dimensional representations contain mostly non-zero values, with similar words having embeddings that exhibit high **cosine similarity**. The embeddings are learned from vast unlabeled datasets spanning millions to hundreds of millions of documents.

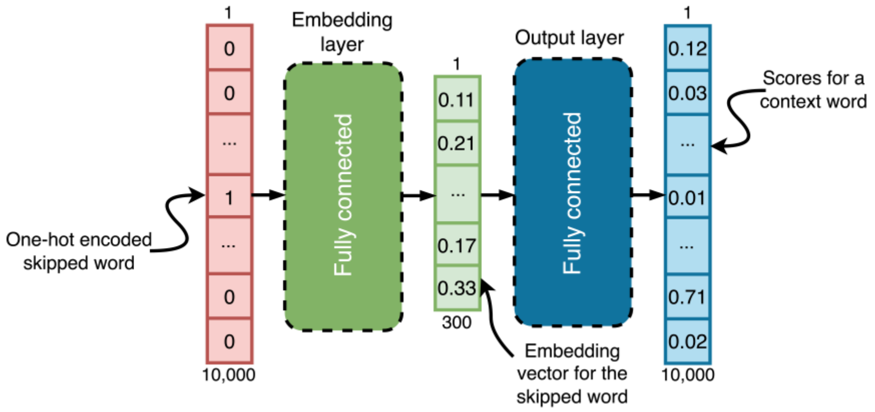
Word2vec, a widely-used embedding learning algorithm, exists in two variants. We'll examine the skip-gram formulation.

Skip-grams are word sequences where one word is omitted. For example, in *"Professor Alan Turing's * advanced computer science,"* the missing word (marked as *) might be "research," "work," or "theories"—words that fit contextually despite not being exact synonyms. Training a model to predict these skipped words from their surrounding context helps it learn semantic relationships between words. The process can also work in reverse: the skipped word can be used to predict its context words. This is the basis of the **skip-gram algorithm**.

The skip-gram size specifies how many context words are included. For a size of five, this means two words before and two after the skipped word. Here are examples of skip-grams of size five from our sentence, with different words skipped (marked as *):

Skip-gram	Skipped word
professor alan * research advanced	turing's
alan turing's * advanced computer	research
turing's research * computer science	advanced

If the corpus vocabulary contains 10,000 words, the skip-gram model with an embedding layer of 300 units, is depicted below:



This is a skip-gram model with a skip-gram size of 5 and the embedding layer of 300 units. As you can see, the model uses a one-hot encoded skipped word to predict a context word, processing the input through two consecutive fully connected layers. It doesn't predict all context words at once but makes separate predictions for each.

Here's how it works for the skip-gram *`professor alan * research advanced"* and the skipped word "turing's". We transform the skip-gram into 4 training pairs:

Skipped word (input)	Context word (target)	Position
turing's	professor	−2
turing's	alan	−1
turing's	research	+1

Skipped word (input)	Context word (target)	Position
turing’s	advanced	+2

For each pair of skipped and context words, say (turing’s, professor), the model:

1. Takes “turing’s” as input,
2. Converts it to a one-hot vector,
3. Passes it through the embedding layer to get the word embedding,
4. Passes the word embedding through the output layer, and
5. Outputs probabilities for “professor.”

For a given context word, the output layer produces a probability vector across the vocabulary. Each value represents how likely that vocabulary word is to be the context word.

A curious reader might notice: if the input for each training pair remains constant—say, “turing’s”—why would the output differ? That’s a great observation! The output will indeed be identical for the same input. However, the loss calculation varies depending on each context word.

> When using **chat language models**, you may notice that the same question often yields different answers. While this might suggest the model is non-deterministic, that’s not accurate. An LLM is fundamentally a neural network, similar to a skip-gram model but with far more parameters. The apparent randomness comes from the way these models are *used* to generate text. During generation, words are sampled based on their predicted probabilities. Though higher-probability words are more likely to be chosen, lower-probability ones may still be selected. This sampling process creates the variations we observe in responses. We will talk about sampling in Chapter 5.

The skip-gram model uses **cross-entropy** as its loss function, just as in the three-class text classifier discussed earlier but handles 10,000 classes—one for each word in the vocabulary. For each skip-gram in the training set, the model computes losses separately for each context word, such as the four words surrounding “turing’s,” then averages these losses to receive feedback on all context word predictions simultaneously.

This training approach enables the model to capture meaningful word relationships, even when working with the same input across different training pairs.

Here’s an example. For the input word “turing’s,” suppose the model assigns these probabilities to different vocabulary words: professor (0.1), alan (0.15), research (0.2), advanced (0.05). When training the model, each input-target word pair contributes to the loss function. For example, when “turing’s” appears with “professor” in the training data, the loss works to increase the score of 0.1. Similarly, when paired with “alan,” the loss works to increase 0.15, with “research” to increase 0.2, and with “advanced” to increase 0.05.

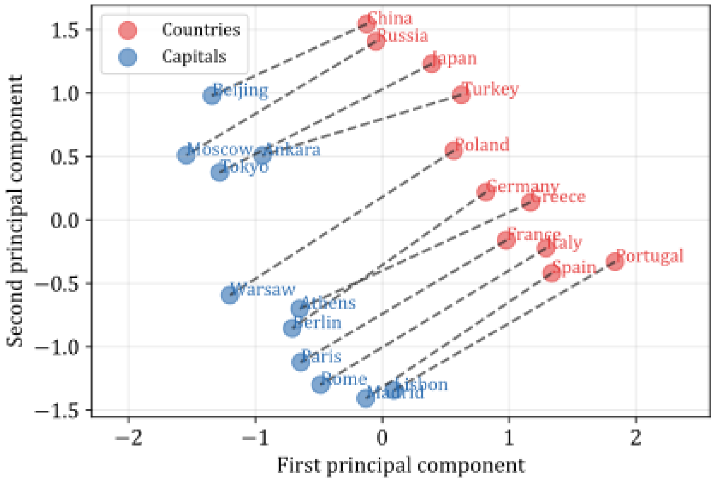
During backpropagation, the model adjusts its weights to make these scores higher for the given context words. For instance, the updated scores might be: professor: 0.11, alan: 0.17, research: 0.22, advanced: 0.07, while the scores for other vocabulary words decrease slightly.

Once training is complete, the output layer is discarded. The embedding layer then serves as the new output layer. When given a one-hot encoded input word, the model produces a 300-dimensional vector—this is the word embedding.

Word2vec is just one method for learning word embeddings from large text corpora. Other methods, such as **GloVe** and **FastText**, offer alternative approaches, focusing on capturing global co-occurrence statistics or subword information to create more robust embeddings.

Using word embeddings to represent texts offers clear advantages over bag of words. One advantage is **dimensionality reduction**, which compresses the word representation from the size of the vocabulary (as in one-hot encoding) to a small vector, typically between 100 and 1000 dimensions. This makes it feasible to process very large corpora in machine learning tasks.

Semantic similarity is another advantage of word embeddings. Words with similar meanings are mapped to vectors that are close to each other in the embedding space. For example, consider word2vec embeddings trained by Google on a news corpus containing about 100 billion words.[4] In the graph below, "Moscow" and "Beijing," or "Russia" and "China," are represented by points located near one another. This reflects their semantic relationships:



The graph displays a 2D projection of 300-dimensional word2vec embedding vectors for countries and their capitals. Words with related meanings cluster together, while nearly parallel lines connect cities to their respective countries, revealing their semantic relationships.

The skip-gram model captures semantic similarity when words occur in similar contexts, even without direct co-occurrence. For instance, if the model produces different probabilities for "films" and "movies," the loss function drives it to predict similar ones, since context words frequently overlap. Through backpropagation, the embedding layer outputs for these words converge.

Before word embeddings, **WordNet** (created at Princeton in 1985) attempted to capture word relationships by organizing words into sets of synonyms and recording semantic

[4] These embeddings can be found online by using the "GoogleNews-vectors-negative300.bin.gz" query. A backup is available on the book's wiki at thelmbook.com/data/word-vectors.

> links between them. While effective, these hand-crafted mappings couldn't scale to large vocabularies or capture the subtle patterns in word usage that naturally emerge from embedding-based approaches.

Because directly visualizing 300-dimensional vectors isn't possible, we used a **dimensionality reduction** technique called **principal component analysis** (**PCA**) to project them onto two dimensions, known as first and second **principal components**.

Dimensionality reduction algorithms compress high-dimensional vectors while maintaining their relationships. The first and second principal components in the above graph preserved the semantic connections between words, revealing their relationships.

> For resources on PCA and other dimensionality reduction methods, check the recommended material listed on the book's wiki.

Word embeddings capture the meaning of words and their relationships to other words. They are fundamental to many natural language processing (NLP) tasks. Neural language models, for example, encode documents as matrices of word embeddings. Each row corresponds to a word's embedding vector, and its position in the matrix reflects the word's position in the document.

> The discovery that word2vec embeddings support meaningful arithmetic operations (like "king $-$ man $+$ woman $\approx$ queen") was a pivotal moment, revealing that neural networks could encode semantic relationships in a space where vector operations produced changes in word meaning. This made the invention of neural networks capable of doing complex math on words, like large language models do, only a matter of time.

Modern language models, though, often use subwords—tokens smaller than complete words. Before moving on to language models—the main topic of this book—let's first examine byte-pair encoding, a widely used subword tokenization method.

2.3. Byte-Pair Encoding

Byte-pair encoding (**BPE**) is a tokenization algorithm that addresses the challenges of handling out-of-vocabulary words by breaking words into smaller units called **subwords**.

Initially a data compression technique, BPE was adapted for NLP by treating words as sequences of characters. It merges the most frequent symbol pairs—characters or subwords—into new subword units. This continues until the vocabulary reaches the target size.

Below is the basic BPE algorithm:

1. **Initialization**: Use a text corpus. Split each word in the corpus into individual characters. For example, the word "hello" becomes "h e l l o". The initial vocabulary consists of all unique characters in the corpus.
2. **Iterative merging**:

   - **Count adjacent symbol pairs**: Treat each character as a **symbol**. Go through the corpus and count every pair of adjacent symbols. For example, in “h e l l o”, the pairs are “h e”, “e l”, “l l”, “l o”.
   - **Select the most frequent symbol pair**: Identify the pair with the highest count in the entire corpus. For instance, if “l l” occurs most frequently, select it.
   - **Merge the selected pair**: Replace all occurrences of the most frequent symbol pair with a new single **merged symbol**. For example, “l l” would be replaced with a new merged symbol “ll”. The word “h e l l o” now becomes “h e ll o”.
   - **Update the vocabulary**: Add the new merged symbol to the vocabulary. The vocabulary now includes the original characters and the new symbol “ll”.
3. **Repeat**: Continue the iterative merging until the vocabulary reaches the desired size.

The algorithm is simple, but implementing it directly on large corpora is inefficient. Recomputing symbol pairs or updating the entire corpus after each merge is computationally expensive.

A more efficient approach initializes the vocabulary with all unique words in the corpus and their counts. Pair counts are calculated using these word counts, and the vocabulary is updated iteratively by merging the most popular pairs. Let’s write the code:

```python
from collections import defaultdict

def initialize_vocabulary(corpus):
    vocabulary = defaultdict(int)
    charset = set()
    for word in corpus:
        word_with_marker = '_' + word ❶
        characters = list(word_with_marker) ❷
        charset.update(characters) ❸
        tokenized_word = ' '.join(characters) ❹
        vocabulary[tokenized_word] += 1 ❺
    return vocabulary, charset
```

The function generates a vocabulary that represents words as sequences of characters and tracks their counts. Given a `corpus` (a list of words), it returns two outputs: `vocabulary`, a dictionary mapping each word—tokenized with spaces between characters—to its count, and `charset`, a set of all unique characters present in the corpus.

Here’s how it works:

- Line ❶ adds a word boundary marker “_” to the start of each word to differentiate subwords at the beginning from those in the middle. For example, “_re” in “restart” is distinct from “re” in “agree.” This helps rebuild sentences from tokens generated using the model. When a token starts with “_”, it marks the beginning of a new word, requiring a space to be added before it.
- Line ❷ splits each word into individual `characters`.
- Line ❸ updates `charset` with any new characters encountered in the word.
- Line ❹ joins `characters` with spaces to create a tokenized version of the word. For example, the word “hello” becomes `_ h e l l o`.

- Line ❺ adds `tokenized_word` to `vocabulary` with its count incremented.

After the initialization, BPE iteratively merges the most frequent pairs of tokens (bigrams) in the `vocabulary`. By removing spaces between these pairs, it forms progressively longer tokens.

```
def get_pair_counts(vocabulary):
    pair_counts = defaultdict(int)
    for tokenized_word, count in vocabulary.items():
        tokens = tokenized_word.split() ❶
        for i in range(len(tokens) - 1):
            pair = (tokens[i], tokens[i + 1]) ❷
            pair_counts[pair] += count ❸
    return pair_counts
```

The function counts how often adjacent token pairs appear in the tokenized vocabulary words. The input `vocabulary` maps tokenized words to their counts, and the output is a dictionary of token pairs and their total counts.

For each `tokenized_word` in `vocabulary`, we split it into tokens in line ❶. A nested loop forms adjacent token pairs in line ❷ and increments their count by the word's count in line ❸.

```
def merge_pair(vocabulary, pair):
    new_vocabulary = {}
    bigram = re.escape(' '.join(pair)) ❶
    pattern = re.compile(r"(?<!\S)" + bigram + r"(?!\S)") ❷
    for tokenized_word, count in vocabulary.items():
        new_tokenized_word = pattern.sub("".join(pair), tokenized_word) ❸
        new_vocabulary[new_tokenized_word] = count
    return new_vocabulary
```

The function merges the input token pair in all tokenized words from the vocabulary. It returns a new vocabulary where every occurrence of the `pair` is merged into a single token. For example, if the pair is `('e', 'l')` and a tokenized word is `"_ h e l l o"`, merging `'e'` and `'l'` removes the space between them, resulting in `"_ h el l o"`.

In line ❶, the `re.escape` function automatically adds backslashes to special characters in a string (like `.`, `*`, or `?`), so they are interpreted as literal characters rather than having their special meaning in regular expressions.

The regular expression in line ❷ matches only whole token pairs. It ensures the `bigram` is not part of a larger word by checking for the absence of non-whitespace characters immediately before and after the match. For instance `"good morning"` matches in `"this is good morning"`, but not in `"thisisgood morning"`, where `"good"` is part of `"thisisgood"`.

> The expressions `(?<!\S)` and `(?!\S)` are regex **negative lookbehind** and **negative lookahead** assertions that ensure a `bigram` stands alone. The lookbehind checks that no non-whitespace character precedes the `bigram`, meaning it follows whitespace or the start of text. The lookahead similarly ensures no non-whitespace follows the `bigram`,

meaning it precedes whitespace or the end of text. Together, these prevent the bigram from being part of longer words.

Finally, in line ❸, the function uses `pattern.sub()` to replace all occurrences of the matched pattern with the joined pair, creating the new tokenized word.

The function below implements the BPE algorithm, merging the most frequent token pairs iteratively until no merges remain or the target vocabulary size is reached:

```
def byte_pair_encoding(corpus, vocab_size):
    vocabulary, charset = initialize_vocabulary(corpus)
    merges = []
    tokens = set(charset)
    while len(tokens) < vocab_size: ❶
        pair_counts = get_pair_counts(vocabulary)
        if not pair_counts: ❷
            break
        most_frequent_pair = max(pair_counts, key=pair_counts.get) ❸
        merges.append(most_frequent_pair)
        vocabulary = merge_pair(vocabulary, most_frequent_pair) ❹
        new_token = ''.join(most_frequent_pair) ❺
        tokens.add(new_token) ❻

    return vocabulary, merges, charset, tokens
```

This function processes a corpus to produce the components needed for a tokenizer. It initializes the vocabulary and character set, creates an empty `merges` list for storing merge operations, and sets `tokens` to the initial character set. Over time, `tokens` grows to include all unique tokens the tokenizer will be able to generate.

The loop in line ❶ continues until the number of tokens supported by the tokenizer reaches `vocab_size` or no pairs remain to merge. Line ❷ checks if there are no more valid pairs, in which case the loop exits. Line ❸ finds the most frequent token pair, which is merged throughout the vocabulary in line ❹ to create a new token in line ❺. This new token is added to the `tokens` set in line ❻, and the merge is recorded in `merges`.

The function returns four outputs: the updated vocabulary, the list of merge operations, the original character set, and the final set of unique tokens.

The function below tokenizes a word using a trained tokenizer:

```
def tokenize_word(word, merges, vocabulary, charset, unk_token="<UNK>"):
    word = '_' + word
    if word in vocabulary:
        return [word]
    tokens = [char if char in charset else unk_token for char in word]

    for left, right in merges:
        i = 0
        while i < len(tokens) - 1:
            if tokens[i:i+2] == [left, right]:
                tokens[i:i+2] = [left + right]
            else:
                i += 1
    return tokens
```

This function tokenizes a `word` using `merges`, `vocabulary`, and `charset` from `byte_pair_encoding`. The `word` is first prefixed. If the prefixed `word` exists in the `vocabulary`, it returns it as the only token. Otherwise, the `word` is split into characters, with any not in `charset` replaced by `unk_token`. These characters are then iteratively merged using the order of rules in `merges`.

To tokenize a text, we first split it into words based on spaces and then tokenize each word individually. The thelmbook.com/nb/2.1 notebook contains code for training a BPE tokenizer by using a news corpus. The tokenized version of the sentence *"Let's proceed to the language modeling chapter."* using the tokenizer trained in the notebook, is:

```
["_Let", "'", "s", "_proceed", "_to", "_the", "_language", "_model", "ing", "
_part", "."]
```

Here, "let's," and "modeling," were broken into subwords. This indicates their relative rarity in the training data and a small target vocabulary size (I set 5000 tokens).

The `tokenize_word` algorithm is inefficient due to nested loops: it iterates over all merges in line ❹ while checking every token pair in line ❺. However, since modern language models have vocabularies exceeding 100,000 tokens, most input words exist in the vocabulary, bypassing subword tokenization. The notebook's optimized version uses caching and precomputed data structures to eliminate these nested loops, reducing tokenization time from 0.0549 to 0.0037 seconds. While actual performance varies by system, the optimized approach consistently delivers better speed.

For languages without spaces, like Chinese, or for multilingual models, the initial space-based tokenization is typically skipped. Instead, the text is split into individual characters. From there, BPE proceeds as usual, merging the most frequent character or token pairs to form subwords.

We're now ready to examine the core ideas of language modeling. We'll begin with traditional count-based methods and cover neural network-based techniques in later chapters.

2.4. Language Model

A **language model** predicts the next token in a sequence by estimating its conditional probability based on previous tokens. It assigns a probability to all possible next tokens, enabling the selection of the most likely one. This capability supports tasks like text generation, machine translation, and speech recognition. Trained on large unlabeled text corpora, language models learn statistical patterns in language, allowing them to be used to generate human-like text.

Formally, for a sequence $\mathbf{s}$ of L tokens $(t_1, t_2, \ldots, t_L)$, a language model computes:

$$\Pr\big(t = t_{L+1} | \mathbf{s} = (t_1, t_2, \ldots, t_L)\big) \tag{2.2}$$

Here, Pr represents the conditional probability distribution over the vocabulary for the next token. A **conditional probability** quantifies the likelihood of one event occurring given that another has already occurred. In language models, it reflects the probability of a specific token being the next one, given the preceding sequence of tokens. This sequence is often referred to as the **input sequence**, **context**, or **prompt**.

The following notations are equivalent to Equation 2.2:

$$\Pr(t_{L+1}|t_1, t_2, \dots, t_L) \text{ or } \Pr(t_{L+1}|\mathbf{s}) \tag{2.3}$$

We will select different notations, ranging from concise to detailed, based on the context.

For any token t and sequence $\mathbf{s}$, the conditional probability satisfies $\Pr(t|\mathbf{s}) \geq 0$, meaning probabilities are always non-negative. Furthermore, the probabilities for all possible next tokens in the vocabulary $\mathcal{V}$ must sum to 1: $\sum_{t \in \mathcal{V}} P\,(t|\mathbf{s}) = 1$. This ensures the model outputs a valid **discrete probability distribution** over the vocabulary.

To illustrate, let's consider an example with a vocabulary $\mathcal{V}$ containing 5 words: "are," "cool," "language," "models," and "useless." For the sequence $\mathbf{s} =$ (language, models, are), a language model could output the following probabilities for each possible next word in $\mathcal{V}$:

$$\begin{aligned}
\Pr\big(t = \text{are}|\mathbf{s} = (\text{language, models, are})\big) &= 0.01 \\
\Pr\big(t = \text{cool}|\mathbf{s} = (\text{language, models, are})\big) &= 0.77 \\
\Pr\big(t = \text{language}|\mathbf{s} = (\text{language, models, are})\big) &= 0.02 \\
\Pr\big(t = \text{models}|\mathbf{s} = (\text{language, models, are})\big) &= 0.15 \\
\Pr\big(t = \text{useless}|\mathbf{s} = (\text{language, models, are})\big) &= 0.05
\end{aligned}$$

The illustration demonstrates how the language model assigns probabilities across its vocabulary for each potential next word, with "cool" receiving the highest probability. These probabilities sum to 1, forming a valid discrete probability distribution.

This type of model is an **autoregressive language model**, also known as a **causal language model**. **Autoregression** involves predicting an element in a sequence using only its predecessors. Such models excel at text generation and include Transformer-based **chat language models** (chat LMs) and all language models discussed in this book.

In contrast, **masked language models**, such as BERT—a pioneering Transformer-based model—use a different approach. These models predict intentionally masked tokens within sequences, utilizing both preceding and following context. This bidirectional approach particularly suits tasks like text classification and named entity recognition.

Before neural networks became standard for language modeling, traditional methods relied on statistical techniques. These count-based models, still used in smartphone autocomplete, estimate the probability of word sequences based on word or n-gram frequency counts learned from a corpus. To understand these methods better, let's implement a simple count-based language model.

2.5. Count-Based Language Model

We'll focus on a trigram model ($n = 3$) to illustrate how this works. In a trigram model, the probability of a token is calculated based on the two preceding tokens:

$$\Pr(t_i|t_{i-2}, t_{i-1}) = \frac{C(t_{i-2}, t_{i-1}, t_i)}{C(t_{i-2}, t_{i-1})}, \tag{2.4}$$

where $C(\cdot)$ denotes the count of occurrences of an n-gram in the training data.

For instance, if the trigram "language models rock" appears 50 times in the corpus and "language models" appears 200 times overall, then:

$$\Pr(\text{rock}|\text{language, models}) = \frac{50}{200} = 0.25$$

This means that "rock" follows "language models" 25% of the time in our training data.

Equation 2.4 is the **maximum likelihood estimate** (MLE) of a token's probability given its context. It measures the relative frequency of a trigram compared to all trigrams sharing the same two-token history. With a larger training corpus, the MLE becomes more reliable for n-grams that occur frequently. This aligns with a basic statistical principle: larger datasets yield more accurate estimates.

However, a limited-size corpus poses a problem: some n-grams we may encounter in practice might not appear in the training data. For instance, if the trigram "language models sing" never appears in our corpus, its probability would be zero according to the MLE:

$$\Pr(\text{sing}|\text{language, models}) = \frac{0}{200} = 0$$

This is problematic because it assigns a zero probability to any sequence containing an unseen n-gram, even if it's a valid phrase. To solve this, several techniques exist, one of which is **backoff**. The idea is simple: if a higher-order n-gram (e.g., trigram) is not observed, we "back off" to a lower-order n-gram (e.g., bigram). The probability $\Pr(t_i|t_{i-2}, t_{i-1})$ is given by one of the following expressions, depending on whether the condition is true:

Expression	**Condition**
$\frac{C(t_{i-2}, t_{i-1}, t_i)}{C(t_{i-2}, t_{i-1})}$	if $C(t_{i-2}, t_{i-1}, t_i) > 0$
$\Pr(t_i|t_{i-1})$	if $C(t_{i-2}, t_{i-1}, t_i) = 0$ and $C(t_{i-1}, t_i) > 0$
$\Pr(t_i)$	otherwise

Here, $C(t_{i-2}, t_{i-1}, t_i)$ is the count of the trigram (t_{i-2}, t_{i-1}, t_i), $C(t_{i-2}, t_{i-1})$ and $C(t_{i-1}, t_i)$ are the counts of the bigrams (t_{i-2}, t_{i-1}) and (t_{i-1}, t_i) respectively.

The bigram probability and unigram probability are computed as:

$$\Pr(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i)}{C(t_{i-1})}, \quad \Pr(t_i) = \frac{C(t_i) + 1}{W + V},$$

where $C(t_i)$ is the count of the token t_i, W is the total number of tokens in the corpus, and V is the vocabulary size.

Adding 1 to $C(t_i)$, known as **add-one smoothing** or **Laplace smoothing**, addresses zero probabilities for tokens not present in the corpus. If we used the actual frequency $\Pr(t_i) = \frac{C(t_i)}{W}$, any token not found in the corpus would have a zero probability, creating problems when the model encounters valid but unseen tokens. Laplace smoothing solves this by adding 1 to each token count, ensuring all tokens, including unseen ones, receive a small, non-zero probability. The denominator is adjusted by adding V to account for the extra counts introduced in the numerator.

Now, let's implement a language model with backoff in the `CountLanguageModel` class (we'll implement Laplace smoothing in the next section):

```python
class CountLanguageModel:
    def __init__(self, n): ❶
        self.n = n
        self.ngram_counts = [{} for _ in range(n)] ❷
        self.total_unigrams = 0

    def predict_next_token(self, context): ❸
        for n in range(self.n, 1, -1): ❹
            if len(context) >= n - 1: ❺
                context_n = tuple(context[-(n - 1):]) ❻
                counts = self.ngram_counts[n - 1].get(context_n)
                if counts:
                    return max(counts.items(), key=lambda x: x[1])[0]
        unigram_counts = self.ngram_counts[0].get(())
        if unigram_counts:
            return max(unigram_counts.items(), key=lambda x: x[1])[0]
        return None
```

In line ❶, the model is initialized with an `n` parameter, defining the maximum n-gram order (e.g., `n=3` for trigrams). The `ngram_counts` list in line ❷ stores n-gram frequency dictionaries for unigrams, bigrams, trigrams, etc., populated during training. For `n=3`, given the corpus *"Language models are powerful. Language models are useful."* in lowercase with punctuation removed, `self.ngram_counts` would contain:

```
ngram_counts[0] = {(): {"language": 2, "models": 2, "are": 2, "powerful": 1,
"useful": 1}}

ngram_counts[1] = {("language",): {"models": 2}, ("models",): {"are": 2}, ("a
re",): {"powerful": 1, "useful": 1}, ("powerful",): {"language": 1}}

ngram_counts[2] = {("language", "models"): {"are": 2}, ("models", "are"): {"p
owerful": 1, "useful": 1}, ("are", "powerful"): {"language": 1}, ("powerful",
"language"): {"models": 1}}
```

The `predict_next_token` method uses backoff to predict the next token. Starting from the highest n-gram order in line ❹, it checks if the context contains enough tokens for this n-gram order in line ❺. If so, it extracts the context in line ❻ and attempts to find a match in `ngram_counts`. If no match is found, it backs off to lower-order n-grams or defaults to unigram counts. For instance, given `context=["language", "models", "are"]` and `n=3`:

- First iteration: `context_n = ("models", "are")`
- Second iteration (if needed): `context_n = ("are",)`
- Last resort: unigram counts with empty tuple key `()`

If a matching context is found, the method returns the token with the highest count for that context. For input `["language", "models"]` it will return `"are"`, the token with highest count among values for the key `("language", "models")` in `ngram_counts[2]`. However, for the input `["english", "language"]` it will not find the key `("english", "language")` in `ngram_counts[2]`, so it will backoff to `ngram_counts[1]` and return `"models"`, the token with highest count among values for the key `("language",)`.

Now, let's define the method that trains our model:

```
def train(model, tokens):
    model.total_unigrams = len(tokens)
    for n in range(1, model.n + 1): ❶
        counts = model.ngram_counts[n - 1]
        for i in range(len(tokens) - n + 1):
            context = tuple(tokens[i:i + n - 1]) ❷
            next_token = tokens[i + n - 1] ❸
            if context not in counts:
                counts[context] = defaultdict(int)
            counts[context][next_token] = counts[context][next_token] + 1
```

The `train` method takes a model (an instance of `CountLanguageModel`) and a list of tokens (the training corpus) as input. It updates the n-gram counts in the model using these tokens.

In line ❶, the method iterates over n-gram orders from 1 to `model.n` (inclusive). For each `n`, it generates n-grams of that order from the token sequence and counts their occurrences.

Lines ❷ and ❸ extract contexts and their subsequent tokens to build a nested dictionary where each context maps to a dictionary of following tokens and their counts. These counts are stored in `model.ngram_counts`, which the `predict_next_token` method later uses to make predictions based on context.

Now, let's train the model:

```
set_seed(42)
n = set_hyperparameters()
data_url = "https://www.thelmbook.com/data/brown"
train_corpus, test_corpus = download_and_prepare_data(data_url)

model = CountLanguageModel(n)
train(model, train_corpus)

perplexity = compute_perplexity(model, test_corpus)
print(f"\nPerplexity on test corpus: {perplexity:.2f}")
```

```
contexts = [
    "i will build a",
    "the best place to",
    "she was riding a"
]

for context in contexts:
    words = tokenize(context)
    next_word = model.predict_next_token(words)
    print(f"\nContext: {context}")
    print(f"Next token: {next_word}")
```

The full implementation of this model, including the methods to retrieve and process the training data, can be found in the thelmbook.com/nb/2.2 notebook. Within the `download_and_prepare_data` method, the corpus is downloaded, converted to lowercase, tokenized into words, and divided into **training** and **test** partitions with a 90/10 split. Let's take a moment to understand why this last step is critical.

In machine learning, using the entire dataset for training leaves no way to evaluate whether the model **generalizes** well. A frequent issue is **overfitting**, where the model excels on training data but struggles to make accurate predictions on unseen, new data.

Partitioning the dataset into training and test sets is a standard practice to control overfitting. It involves two steps: (1) shuffling the data and (2) splitting it into two subsets. The larger subset, called the training data, is used for training the model, while the smaller subset, called the test data, is used to evaluate the model's performance on unseen examples.

> The test set requires sufficient size to reliably estimate model performance. A test ratio of 0.1 to 0.3 (10% to 30% of the entire dataset) is common, though this varies with dataset size. For very large datasets, even a smaller test set ratio results in enough examples to provide reliable performance estimates.

The training data comes from the **Brown Corpus**, a collection of over 1 million words from American English texts published in 1961. This corpus is frequently used in linguistic studies.

When you run the code, you will see the following output:

```
Perplexity on test corpus: 299.06

Context: i will build a
Next word: wall

Context: the best place to
Next word: live

Context: she was riding a
Next word: horse
```

Ignore the perplexity number for now; we'll discuss it shortly. Count-based language models can produce reasonable immediate continuations, making them good for autocomplete systems. However, they have notable limitations. These models generally work with word-tokenized corpora, as their n-gram size is typically small (up to $n = 5$). Extending beyond this would require too much memory and lead to slower processing. Subword tokenization, while more efficient, results in many n-grams that represent only fragments of words, degrading the quality of next-word predictions.

Word-level tokenization creates another significant drawback: count-based models cannot handle out-of-vocabulary (OOV) words. This is similar to the issue seen in the **bag-of-words** approach discussed in Section 2.1. For example, consider the context: *"according to WHO, COVID-19 is a."* If "COVID-19" wasn't in the training data, the model would back off repeatedly until it relies only on *"is a,"* severely limiting the context for meaningful predictions.

Count-based models are also unable to capture long-range dependencies in language. While modern Transformer models can handle thousands of tokens, training a count-based model with a context of 1000 tokens would require storing counts for all n-grams from $n = 1$ to $n = 1000$, requiring prohibitive amounts of memory.

Additionally, these models cannot be adapted for downstream tasks after training, as their n-gram counts are fixed, and any adjustment requires retraining on new data.

These limitations have led to the development of advanced methods, particularly neural network-based language models, which have largely replaced count-based models in modern natural language processing. Approaches like recurrent neural networks and transformers, which we'll discuss in the next two chapters, handle longer contexts effectively, producing coherent and context-aware text. Before exploring these methods, let's look at how to evaluate a language model's quality.

2.6. Evaluating Language Models

Evaluating language models measures their performance and allows comparing models. Several metrics and techniques are commonly used. Let's look at the main ones.

2.6.1. Perplexity

Perplexity is a widely used metric for evaluating language models. It measures how well a model predicts a text. Lower perplexity values indicate a better model—one that is more confident in its predictions. Perplexity is defined as the exponential of the average **negative log-likelihood** per token in the test set:

$$\text{Perplexity}(\mathcal{D}, k) = \exp\left(-\frac{1}{D}\sum_{i=1}^{D} \log \Pr\big(t_i | t_{\max(1,i-k)}, \dots, t_{i-1}\big)\right) \tag{2.5}$$

Here, $\mathcal{D}$ represents the test set, D is the total number of tokens in it, t_i is the i^{th} token, and $\Pr\big(t_i | t_{\max(1,i-k)}, \dots, t_{i-1}\big)$ is the probability the model assigns to t_i given its preceding context window of size k, where $\max(1, i-k)$ ensures we start from the first token when there aren't enough preceding tokens to fill the context window. The notations $\exp(x)$ and e^x, where e is **Euler's number**, are equivalent.

The negative log-likelihood (NLL) in Equation 2.5 is the negative logarithm of the probabilities our language model assigns. When a model processes text like "*language models are*" and assigns a probability of 0.77 to the next word "cool", the NLL would be $-\log(0.77)$. It's called "negative" log-likelihood because we take the negative of the logarithm, and "likelihood" refers to these conditional probabilities the model computes. In language modeling, NLL serves two purposes: it acts as a loss function during training to help models learn better probability distributions (which we'll see in the next chapter when training a recurrent neural network language model), and as shown in the perplexity formula, it helps us evaluate how well models predict text.

Perplexity can be understood more intuitively through its geometric mean formulation. The geometric mean of a set of numbers is the D^{th} root of their product (where D is the number of values), and perplexity is the geometric mean of the inverse probabilities:

$$\text{Perplexity}(\mathcal{D}, k) = \left(\prod_{i=1}^{D} \frac{1}{\Pr\left(t_i | t_{\max(1,i-k)}, \dots, t_{i-1}\right)}\right)^{\frac{1}{D}}$$

This form shows that perplexity represents the weighted average factor by which the model is "perplexed" when predicting each token. A perplexity of 10 means that, on average, the model is as uncertain as if it had to choose uniformly between 10 possibilities at each step.

> If a language model assigns equal probability to every token in a vocabulary of size V, its perplexity equals V. This provides an intuitive upper bound for perplexity—a model cannot be more uncertain than when it assigns equal likelihood to all possible tokens.

While both formulations of perplexity shown above are mathematically equivalent (proof available on the book's wiki), the exponential form is computationally more convenient as it transforms products into sums through the logarithm, making calculations more numerically stable.

Let's calculate perplexity using the example text with word-level tokenization and ignoring punctuation: *"We are evaluating a language model for English."* To keep things simple, we assume a context of up to three words. We begin by determining the probability of each word based on its preceding context of three words as provided by the model. Here are the probabilities:

$$\begin{aligned}
\Pr(\text{We}) &= 0.10 \\
\Pr(\text{are} \mid \text{We}) &= 0.20 \\
\Pr(\text{evaluating} \mid \text{We, are}) &= 0.05 \\
\Pr(\text{a} \mid \text{We, are, evaluating}) &= 0.50 \\
\Pr(\text{language} \mid \text{are, evaluating, a}) &= 0.30 \\
\Pr(\text{model} \mid \text{evaluating, a, language}) &= 0.40 \\
\Pr(\text{for} \mid \text{a, language, model}) &= 0.15 \\
\Pr(\text{English} \mid \text{language, model, for}) &= 0.25
\end{aligned}$$

Using the probabilities, we compute the negative log-likelihood for each word:

$$
\begin{aligned}
-\log\big(P(\text{We})\big) &= -\log(0.10) \approx 2.30\\
-\log\big(P(\text{are} \mid \text{We})\big) &= -\log(0.20) \approx 1.61\\
-\log\big(P(\text{evaluating} \mid \text{We, are})\big) &= -\log(0.05) \approx 3.00\\
-\log\big(P(\text{a} \mid \text{We, are, evaluating})\big) &= -\log(0.50) \approx 0.69\\
-\log\big(P(\text{language} \mid \text{are, evaluating, a})\big) &= -\log(0.30) \approx 1.20\\
-\log\big(P(\text{model} \mid \text{evaluating, a, language})\big) &= -\log(0.40) \approx 0.92\\
-\log\big(P(\text{for} \mid \text{a, language, model})\big) &= -\log(0.15) \approx 1.90\\
-\log\big(P(\text{English} \mid \text{language, model, for})\big) &= -\log(0.25) \approx 1.39
\end{aligned}
$$

Next, we sum these values and divide the sum by the number of words (8) to get the average:

$$(2.30 + 1.61 + 3.00 + 0.69 + 1.20 + 0.92 + 1.90 + 1.39)/8 \approx 1.63$$

Finally, we exponentiate the average negative log-likelihood to obtain the perplexity:

$$e^{1.63} \approx 5.10$$

So, the model's perplexity on this text, using a 3-word context, is about 5.10. This means that, on average, the model acts as if it selects from roughly 5 equally likely options for each prediction.

Now, let's calculate the perplexity of the count-based model from the previous section. To do this, the model must be updated to return the probability of a token given a specific context. Add this function to the `CountLanguageModel` we implemented earlier:

```python
def get_probability(self, token, context):
    for n in range(self.n, 1, -1): ➊
        if len(context) >= n - 1:
            context_n = tuple(context[-(n - 1):])
            counts = self.ngram_counts[n - 1].get(context_n)
            if counts: ➋
                total = sum(counts.values()) ➌
                count = counts.get(token, 0)
                if count > 0:
                    return count / total ➍
    unigram_counts = self.ngram_counts[0].get(()) ➎
    count = unigram_counts.get(token, 0)
    V = len(unigram_counts)
    return (count + 1) / (self.total_unigrams + V) ➏
```

The `get_probability` function is similar to `predict_next_token`. Both loop through n-gram orders in reverse (line ➊) and extract the relevant context (`context_n`). If `context_n` matches in the n-gram counts (line ➋), the function retrieves the token counts. If no match exists, it backs off to lower-order n-grams and, finally, unigrams (line ➎).

Unlike `predict_next_token`, which returns the most probable token directly, `get_probability` calculates a token's probability. In line ➌, `total` is the sum of counts for tokens following the context, acting as the denominator. Line ➍ divides the token count by `total` to compute its probability. If no higher-order match exists, line ➏ uses **add-one smoothing** with unigram counts.

The `compute_perplexity` method computes a language model's perplexity for a token sequence. It takes three arguments: the model, the token sequence, and the context size:

```
def compute_perplexity(model, tokens, context_size):
    if not tokens:
        return float('inf')
    total_log_likelihood = 0
    num_tokens = len(tokens)
    for i in range(num_tokens): ➊
        context_start = max(0, i - context_size)
        context = tuple(tokens[context_start:i]) ➋
        word = tokens[i]
        probability = model.get_probability(word, context)
        total_log_likelihood += math.log(probability) ➌
    average_log_likelihood = total_log_likelihood / num_tokens ➍
    perplexity = math.exp(-average_log_likelihood) ➎
    return perplexity
```

In line ➊, the function iterates through each token in the sequence. For every token:

- Line ➋ extracts its context, using up to `context_size` tokens before it. The expression `max(0, i - context_size)` ensures indices stay within bounds like in Equation 2.5.
- In line ➌, the log of the token's probability is added to the cumulative log-likelihood. The `get_probability` method from the model handles the probability calculation.

Once all tokens are processed, line ➍ computes the average log-likelihood by dividing the total log-likelihood by the number of tokens.

Finally, in line ➎, the perplexity is computed as the exponential of the negative average log-likelihood, as described in Equation 2.5.

By applying this method to the `test_corpus` sequence from the thelmbook.com/nb/2.2 notebook, we observe the following output:

```
Perplexity on test corpus: 299.06
```

This perplexity is very high. For example, **GPT-2** has a perplexity of about 20, while modern LLMs achieve values below 5. Later, we'll compute perplexities for RNN and Transformer-based models and compare them with the perplexity of this count-based model.

2.6.2. ROUGE

Perplexity is a standard metric used to evaluate language models trained on large, unlabeled datasets by measuring how well they predict the next token in context. These models are referred to as **pretrained models** or **base models**. As we'll discuss in the chapter on large language models, their ability to perform specific tasks or answer questions comes from **supervised finetuning**. This additional training uses a labeled dataset where input contexts are matched with target outputs, such as answers or task-specific results. This enables problem-solving capabilities.

Perplexity is not an ideal metric for evaluating a finetuned model. Instead, metrics are needed that compare the model's output to reference texts, often called **ground truth**. A common choice is **ROUGE** (Recall-Oriented Understudy for Gisting Evaluation). ROUGE is widely used for tasks like summarization and machine translation. It evaluates text quality by measuring overlaps, such as tokens or n-grams, between the generated text and the reference.

ROUGE has several variants, each focusing on different aspects of text similarity. Here, we'll discuss three widely used ones: ROUGE-1, ROUGE-N, and ROUGE-L.

ROUGE-N evaluates the overlap of n-grams between the generated and reference texts, with N indicating the length of the n-gram. One of the most commonly used versions is **ROUGE-1**.

ROUGE-1 measures the overlap of unigrams (single tokens) between the generated and reference texts. As a recall-focused metric (hence the "R" in ROUGE), it assesses how much of the reference text is captured in the generated output.

Recall is the ratio of matching tokens to the total number of tokens in the reference text:

$$\text{recall} \overset{\text{def}}{=} \frac{\text{Number of matching tokens}}{\text{Total number of tokens in reference texts}}$$

Formally, ROUGE-1 is defined as:

$$\text{ROUGE-1} \overset{\text{def}}{=} \frac{\sum_{(g,r)\in\mathcal{D}} \sum_{t\in r} \text{count}\,(t, g)}{\sum_{(g,r)\in\mathcal{D}} \text{length}\,(r)}$$

Here, $\mathcal{D}$ is the dataset of (generated text, reference text) pairs, count(t, g) counts how often a token t from the reference text r appears in the generated text g, and the denominator is the total token count across all reference texts.

To understand this calculation, consider a simple example:

Reference text	**Generated text**
Large language models are very important for text processing.	Large language models are useful in processing text.

Let's use word-level tokenization and calculate:

- **Matching words**: large, language, models, are, processing, text (6 words)
- **Total words in the reference text**: 9
- **ROUGE-1**: $\frac{6}{9} \approx 0.67$

A ROUGE-1 score of 0.67 means roughly two-thirds of the words from the reference text appear in the generated text. However, this number alone has little value. ROUGE scores are only useful for *comparing* how different language models perform on the same test set, as they indicate which model more effectively captures the content of the reference texts.

ROUGE-N extends the ROUGE metric from unigrams to n-grams while using the same formula.

ROUGE-L relies on the **longest common subsequence** (LCS). This is the longest sequence of tokens appearing in both generated and reference texts in the same order, without being adjacent.

Let g and r be the generated and reference texts with lengths L_g and L_r. Then:

$$\text{recall}_{\text{LCS}} \overset{\text{def}}{=} \frac{\text{LCS}(g,r)}{L_r}, \quad \text{precision}_{\text{LCS}} \overset{\text{def}}{=} \frac{\text{LCS}(g,r)}{L_g}$$

Here, $\text{LCS}(L_g, L_r)$ represents the number of tokens in the LCS between the generated text g and the reference text r. The **recall** measures the proportion of the reference text captured by the LCS, while the **precision** measures the proportion of the generated text that matches the reference. Recall and precision are combined into a single metric as follows:

$$\text{ROUGE-L} \overset{\text{def}}{=} (1+\beta^2) \times \frac{\text{recall}_{\text{LCS}} \times \text{precision}_{\text{LCS}}}{\text{recall}_{\text{LCS}} + \beta^2 \times \text{precision}_{\text{LCS}}}$$

Here, β controls the trade-off between precision and recall in the ROUGE-L score. Since ROUGE favors recall, β is usually set high, such as 8.

Let's revisit the two texts used to illustrate ROUGE-L. For these sentences, there are two valid longest common subsequences, each with a length of 5 words:

LCS 1	**LCS 2**
Large, language, models, are, text	Large, language, models, are, processing

Both subsequences are the longest sequences of tokens that appear in the same order in both sentences, but not necessarily consecutively. When multiple LCS options exist, ROUGE-L can use any of them since their lengths are identical.

Here's how the calculations work here. The length of the LCS is 5 words. The reference text is 9 words long, and the generated text is 8 words long. Thus, recall and precision are:

$$\text{recall}_{\text{LCS}} = \frac{5}{9} \approx 0.56, \quad \text{precision}_{\text{LCS}} = \frac{5}{8} \approx 0.63$$

With $\beta = 8$, ROUGE-L is then given by:

$$\text{ROUGE-L} = \frac{(1+8^2)\cdot 0.56 \cdot 0.63}{0.56 + 8^2 \cdot 0.63} \approx 0.56$$

ROUGE scores range from 0 to 1, where 1 means a perfect match between generated and reference texts. However, even excellent summaries or translations rarely approach 1 in practice.

Choosing the right ROUGE metric depends on the task:

- ROUGE-1 and ROUGE-2 are standard starting points. ROUGE-1 checks overall content similarity using unigram overlap, while ROUGE-2 evaluates local fluency and phrase accuracy using bigram matches.

- ROUGE-L is preferred over ROUGE-1 or ROUGE-2 when evaluating text quality in terms of sentence structure and flow, particularly in summarization and translation tasks, since it captures the longest sequence of matching words that appear in the same relative order, better reflecting grammatical coherence.

- In cases where preserving longer patterns is key—such as maintaining technical terms or idioms—higher-order metrics like ROUGE-3 or ROUGE-4 might be more relevant.

A combination of metrics, such as ROUGE-1, ROUGE-2, and ROUGE-L, often gives a more balanced evaluation, covering both content overlap and structural flexibility.

Keep in mind, though, that ROUGE has limitations. It measures lexical overlap but not semantic similarity or factual correctness. To address these gaps, ROUGE is often paired with human evaluations or other methods for a fuller assessment of text quality.

2.6.3. Human Evaluation

Automated metrics are useful, but human evaluation is still necessary to assess language models. Humans can evaluate qualities that automated metrics often miss, like fluency and accuracy. Two common approaches for human evaluation are Likert scale ratings and Elo ratings.

Likert scale ratings involve assigning scores to outputs using a fixed, typically symmetric scale. Raters judge the quality by selecting a score, often from −2 to 2, where each scale point corresponds to a descriptive label. For instance, −2 could mean "Strongly Disagree" or "Poor," while 2 might mean "Strongly Agree" or "Excellent." The scale is symmetric because it includes equal levels of agreement and disagreement around a neutral midpoint, making positive and negative responses easier to interpret.

Likert scales are flexible for evaluating different aspects of language model outputs, such as fluency, coherence, relevance, and accuracy. For example, a rater could separately rate a sentence's grammatical correctness and its relevance to a prompt, both on a scale from −2 to 2.

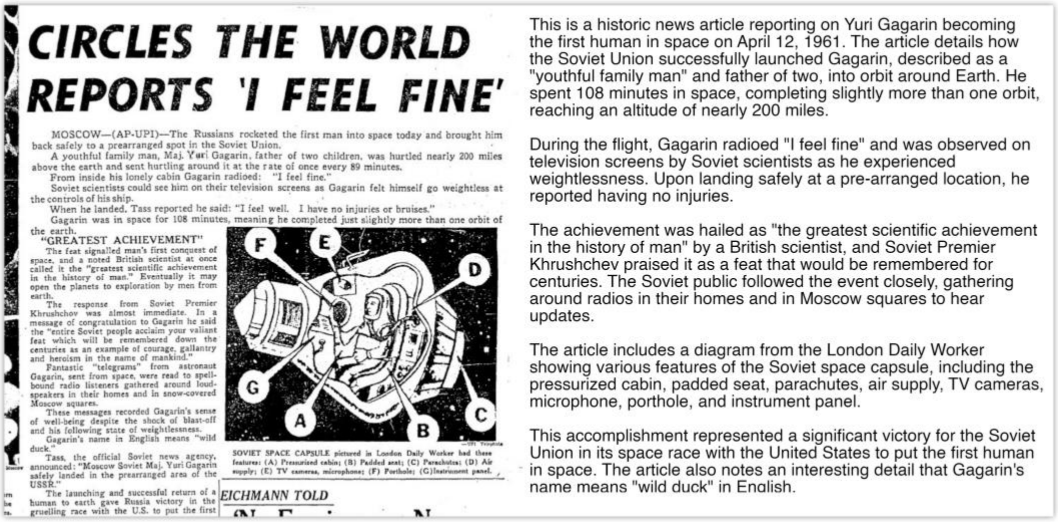
However, the method has limitations. One issue is **central tendency bias**, where some raters avoid extreme scores and stick to the middle of the scale. Another challenge is inconsistency in how raters interpret the scale—some may reserve a 2 for exceptional outputs, while others may assign it to any high-quality response.

To mitigate these issues, researchers often involve multiple raters, phrase similar questions in different ways for the same rater, and use detailed rubrics that clearly define each scale point.

Let's illustrate Likert scale evaluation using a scenario where machine-generated summaries of news articles are assessed.

Human raters compare a model-generated summary to the original article. They rate it on three aspects using 5-point Likert scales: coherence, informativeness, and factual accuracy.

For example, consider the news article on the left and the generated summary on the right:



CIRCLES THE WORLD REPORTS 'I FEEL FINE'

MOSCOW—(AP-UPI)—The Russians rocketed the first man into space today and brought him back safely to a prearranged spot in the Soviet Union.

A youthful family man, Maj. Yuri Gagarin, father of two children, was hurtled nearly 200 miles above the earth and sent hurtling around it at the rate of once every 89 minutes.

From inside his lonely cabin Gagarin radioed: "I feel fine."

Soviet scientists could see him on their television screens as Gagarin felt himself go weightless at the controls of his ship.

When he landed, Tass reported he said: "I feel well. I have no injuries or bruises."

Gagarin was in space for 108 minutes, meaning he completed just slightly more than one orbit of the earth.

"GREATEST ACHIEVEMENT"

The feat signalled man's first conquest of space, and a noted British scientist at once called it the "greatest scientific achievement in the history of man." Eventually it may open the planets to exploration by men from earth.

The response from Soviet Premier Khrushchov was almost immediate. In a message of congratulation to Gagarin he said the "entire Soviet people acclaim your valiant feat which will be remembered down the centuries as an example of courage, gallantry and heroism in the name of mankind."

Fantastic "telegrams" from astronaut Gagarin, sent from space, were read to spell-bound radio listeners gathered around loud-speakers in their homes and in snow-covered Moscow squares.

These messages recorded Gagarin's sense of well-being despite the shock of blast-off and his following state of weightlessness.

Gagarin's name in English means "wild duck."

Tass, the official Soviet news agency, announced: "Moscow Soviet Maj. Yuri Gagarin safely landed in the prearranged area of the USSR."

The launching and successful return of a human to earth gave Russia victory in the gruelling race with the U.S. to put the first

SOVIET SPACE CAPSULE pictured in London Daily Worker had these features: (A) Pressurized cabin; (B) Padded seat; (C) Parachutes; (D) Air supply; (E) TV cameras, microphone; (F) Porthole; (G) Instrument panel.

EICHMANN TOLD

This is a historic news article reporting on Yuri Gagarin becoming the first human in space on April 12, 1961. The article details how the Soviet Union successfully launched Gagarin, described as a "youthful family man" and father of two, into orbit around Earth. He spent 108 minutes in space, completing slightly more than one orbit, reaching an altitude of nearly 200 miles.

During the flight, Gagarin radioed "I feel fine" and was observed on television screens by Soviet scientists as he experienced weightlessness. Upon landing safely at a pre-arranged location, he reported having no injuries.

The achievement was hailed as "the greatest scientific achievement in the history of man" by a British scientist, and Soviet Premier Khrushchev praised it as a feat that would be remembered for centuries. The Soviet public followed the event closely, gathering around radios in their homes and in Moscow squares to hear updates.

The article includes a diagram from the London Daily Worker showing various features of the Soviet space capsule, including the pressurized cabin, padded seat, parachutes, air supply, TV cameras, microphone, porthole, and instrument panel.

This accomplishment represented a significant victory for the Soviet Union in its space race with the United States to put the first human in space. The article also notes an interesting detail that Gagarin's name means "wild duck" in English.

Raters assess the summary based on how effectively it meets these three criteria.

The scale for assessing coherence—that is, how well-organized, readable, and logically connected the summary is—might look like this:

Very poor	Poor	Acceptable	Good	Excellent
-2	-1	0	1	2

The scale for assessing informativeness, that is, how well the summary captures the essence and main points of the original article, might look this way:

Not informative	Slightly	Moderately	Very	Extremely informative
-2	-1	0	1	2

The scale for assessing factual accuracy—that is, how precisely the summary represents facts and data from the original article—might look like this:

Very low	Some inaccuracies	Mostly Accurate	Very accurate	Perfect
-2	-1	0	1	2

In this example, raters would select one option for each of the three aspects. The use of descriptive anchors at each point on the scale helps standardize understanding among raters.

After collecting ratings from multiple raters across various summaries, researchers analyze the data through several approaches:

- Calculate average scores for each aspect across all summaries and raters to get an overall performance measure.
- Compare scores across different versions of the model to track improvements.
- Analyze the correlation between different aspects (e.g., is high coherence associated with high factual accuracy?).

Although Likert scale ratings were originally intended for humans, the rise of advanced **chat LMs** means raters can now be either humans or language models.

Pairwise comparison is a method where two outputs are evaluated side-by-side, and the better one is chosen based on specific criteria. This simplifies decision-making, especially when outputs are of similar quality or changes are minor.

The method builds on the principle that relative judgments are easier than absolute ones. Binary choices often produce more consistent and reliable results than absolute ratings.

In practice, raters compare pairs of outputs, such as translations, summaries, or answers, and decide which is better based on criteria like coherence, informativeness, or factual accuracy.

For example, in machine translation evaluation, raters compare pairs of translations for each source sentence, selecting which one better preserves the original meaning in the target language. By repeating this process across many pairs, evaluators can compare different models or versions.

Pairwise comparison helps rank models or model versions by having each rater evaluate multiple pairs, with each model compared against others several times. This repetition minimizes individual biases, resulting in more reliable evaluations. A related approach is **ranking**, where a rater orders multiple responses by quality. Ranking requires less effort than pairwise comparisons while still capturing relative quality.

Results from pairwise comparisons are typically analyzed statistically to determine significant differences between models. A common method for this analysis is the Elo rating system.

Elo ratings, originally created by Arpad Elo in 1960 for ranking chess players, can be adapted for language model evaluation. The system assigns ratings based on "wins" and "losses" in direct comparisons, quantifying relative model performance.

In language model evaluation, all models typically start with an initial rating, often set to 1500. When two models are compared, the probability of one model "winning" is calculated using their current ratings. After each comparison, their ratings are updated based on the actual result versus the expected result.

The probability of model A with rating $\mathrm{Elo}(A)$ winning against model B with rating $\mathrm{Elo}(B)$ is:

$$\Pr(A \text{ wins}) = \frac{1}{1 + 10^{\left(\mathrm{Elo}(B) - \mathrm{Elo}(A)\right)/400}}$$

The value 400 in the Elo formula acts as a scaling factor, creating a logarithmic relationship between rating differences and winning probabilities. Arpad Elo chose this number ensuring that a 400-point rating difference reflects $10{:}1$ odds in favor of the higher-rated chess player. While originally designed for chess, this scaling factor has proven effective in other contexts, including language model evaluation.

After a match, ratings are updated using the formula:

$$\mathrm{Elo}(A) \leftarrow \mathrm{Elo}(A) + k \times \left(\mathrm{score}(A) - \Pr(A \text{ wins})\right),$$

where k (typically between 4 and 32) controls the maximum rating change, and $\text{score}(A)$ reflects the outcome: 1 for a win, 0 for a loss, and 0.5 for a draw.

Consider an example with three models: LM_1, LM_2, and LM_3. We'll evaluate them based on their ability to generate coherent text continuations. Assume their initial ratings are:

$$\begin{aligned} \text{Elo}(\text{LM}_1) &= 1500 \\ \text{Elo}(\text{LM}_2) &= 1500 \\ \text{Elo}(\text{LM}_3) &= 1500 \end{aligned}$$

We'll use $k = 32$ for this example.

Consider this prompt: *"The scientists were shocked when they discovered…"*

Continuation by LM_1: *"…a new species of butterfly in the Amazon rainforest. Its wings were unlike anything they had ever seen before."*

Continuation by LM_2: *"…that the ancient artifact they unearthed was emitting a faint, pulsating light. They couldn't explain its source."*

Continuation by LM_3: *"…the results of their experiment contradicted everything they thought they knew about quantum mechanics."*

Let's say we conduct pairwise comparisons and get the following results:

1. LM_1 vs LM_2: LM_1 wins
   - $\Pr(\text{LM}_1 \text{ wins}) = 1/\left(1 + 10^{((1500-1500)/400)}\right) = 0.5$
   - New rating for $\text{LM}_1 \leftarrow 1500 + 32(1 - 0.5) = 1516$
   - New rating for $\text{LM}_2 \leftarrow 1500 + 32(0 - 0.5) = 1484$
2. LM_1 vs LM_3: LM_3 wins
   - $\Pr(\text{LM}_1 \text{ wins}) = 1/\left(1 + 10^{((1500-1516)/400)}\right) \approx 0.523$
   - New rating for $\text{LM}_1 \leftarrow 1516 + 32(0 - 0.523) \approx 1499$
   - New rating for $\text{LM}_3 \leftarrow 1500 + 32(1 - 0.477) \approx 1517$
3. LM_2 vs LM_3: LM_3 wins
   - $\Pr(\text{LM}_2 \text{ wins}) = 1/\left(1 + 10^{((1517-1484)/400)}\right) \approx 0.453$
   - New rating for $\text{LM}_2 \leftarrow 1484 + 32(0 - 0.453) \approx 1470$
   - New rating for $\text{LM}_3 \leftarrow 1517 + 32(1 - 0.547) \approx 1531$

Final ratings after these comparisons:

$$\begin{aligned} \text{Elo}(\text{LM}_1) &= 1499 \\ \text{Elo}(\text{LM}_2) &= 1470 \\ \text{Elo}(\text{LM}_3) &= 1531 \end{aligned}$$

Elo ratings quantify how models perform relative to each other. In this case, LM_3 is the strongest, followed by LM_1, with LM_2 ranking last.

Performance isn't judged from a single match. Instead, multiple pairwise matches are used. This limits the effects of random fluctuations or biases in individual comparisons, giving a better estimate of each model's performance.

A variety of prompts or inputs ensures evaluation across different contexts and tasks. When human raters are involved, several raters assess each comparison to reduce individual bias.

To avoid order effects, both the sequence of comparisons and the presentation of outputs are randomized. Elo ratings are updated after every comparison.

How many matches are needed until the results are reliable? There's no universal number that applies to all cases. As a general guideline, some researchers suggest that each model should participate in at least 100–200 comparisons before considering the Elo ratings stable and ideally 500+ comparisons for high confidence. However, for high-stakes evaluations or when comparing very similar models, thousands of comparisons may be necessary.

> Statistical methods can be used to calculate a **confidence interval** for a model's Elo rating. Explaining these techniques is beyond the scope of this book. For those interested, the **Bradley–Terry model** and **bootstrap resampling** are good starting points. Both are well-documented, with resources linked on the book's wiki.

Elo ratings provide a continuous scale for ranking models, making it easier to track incremental improvements. The system rewards wins against strong opponents more than wins against weaker ones, and it can handle incomplete comparison data, meaning not every model needs to be compared against every other model. However, the choice of k significantly affects rating volatility; a poorly chosen k can undermine the evaluation's stability.

To address these limitations, Elo ratings are often used alongside other evaluation methods. For instance, researchers might use Elo ratings for ranking models in pairwise comparisons, while collecting Likert scale ratings to assess absolute quality. This combined approach yields a more comprehensive view of a language model's performance.

Now that we've covered language modeling and evaluation methods, let's explore a more advanced model architecture: recurrent neural networks (RNNs). RNNs made significant progress in processing text. They introduced the ability to maintain context over long sequences, allowing for the creation of more powerful language models.