

universität freiburg

Databases and Information Systems WS 25/26

Lecture 12: N-gram Models, LLMs, RAG

January 27, 2026

Prof. Dr. Hannah Bast
Department of Computer Science
University of Freiburg

Overview of this lecture

■ Organizational

- Your experiences with ES11

Vector Space Model
Word Embeddings

■ Contents

- N-gram models unigram, bigram, trigram
- Large language models (LLMs) via API
- Retrieval-Aug. Generation (RAG) how to deal with large inputs
- ES12: Build a small search engine that uses LLMs and RAG to search in the transcripts of the lecture videos

Experiences with ES 11

■ Excerpts from your feedback [more submissions than for ES10 !]

"I really liked the lecture and especially the topic (seeing how earlier stuff can be computed efficiently using linear algebra)"

"It took time to get used to PyTorch"

"The exercise was mathematically very interesting, but code-wise it was a nightmare. I really hate PyTorch"

"This sheet would have been very hard without the doctests"

"173 exam registrations, 5 submissions for ES 10 → wow"

"If the sheets were obligatory, I would have more pressure to submit them on time ... but I always do them at least afterwards"

"I have many other deadlines, and the sheets for this course are not mandatory ... therefore, I work on them later on"

■ Tokens

- Tokens are the individual building blocks of our text
- Here is an example text and three ways to tokenize it (one grey box per token, which may include spaces and punctuation)

Text: **To be, or not to be**

Into words: **To be or not to be**

Into subwords: **To be, or not to be**

Into characters: **T o b e , o r n o t t o b e**

For the n-gram models (see the next slides + our live code), we will tokenize the given text into individual **characters**

Modern language models tokenize the text into **subwords**

■ Unigram model

- Let **V** be our vocabulary = the set of distinct tokens
- Then a **unigram** model is a probability distribution over **V**
- A natural way to derive a unigram model from a given text is to compute the number of occurrences for each $v \in V$, and then normalize this to a probability distribution P
- For example: $V = \{ "a", "b", "c" \}$, $\text{Text} = "abaabacc"$

$$\#a = 4, \#b = 2, \#c = 2, \quad |\text{Text}| = 8$$

$$\text{Pr}(a) = \frac{4}{8} = 0.5$$

$$\text{Pr}(b) = \frac{2}{8} = 0.25$$

$$\text{Pr}(c) = \frac{2}{8} = 0.25$$

Let's write code for this together and derive (or "learn") a unigram model from some of our movie plots from ES11

■ Using a unigram model as a generative model

- Generate one character after the other
- Pick each character from the probability distribution, independently of the previous (and following) characters
- Trick: you can pick a character at random from a probability distribution derived from normalized counts as follows:

1. Compute the cumulative counts $c_1, \dots, c_m$, where $m = |V|$

2. Pick a random integer r from $1 \dots c_m$

3. Find the smallest j with $r \leq c_j$

4. Generate the j -th character from V

$\#a=4, \#b=2, \#c=2$
cumulative counts.
a 4 pick $\in 1 \dots 8$
b 6 e.g. 5
c 8
1 2 3 4 5 6 7 8
a a a a b b c c

Let's implement this together and generate some text

■ Bigram model

- Let **V** be our vocabulary = the set of distinct tokens
- Then a **bigram** model is, for each $v \in V$ (the previous token), a probability distribution over **V** (the next token)
- A natural way to derive a bigram model from a given text is to compute the number of occurrences of each sequence of **two** tokens from **V** ... a sequence of two tokens is called a **bigram**
- For example: $V = \{ "a", "b" \}$, Text = "abaaaaabbba"

$\# \underline{a} \underline{a} = 4$	$\Rightarrow \Pr(\underline{a} \underline{a}) = \#aa / (\#aa + \#ab) = 4/6 = 2/3$
$\# \underline{a} \underline{b} = 2$	$\Rightarrow \Pr(\underline{b} \underline{a}) = \#ab / (\#aa + \#ab) = 2/6 = 1/3$
$\# \underline{b} \underline{a} = 2$	$\Rightarrow \Pr(\underline{a} \underline{b}) = \dots = 2/4 = 1/2$
$\# \underline{b} \underline{b} = 2$	$\Rightarrow \Pr(\underline{b} \underline{b}) = \dots = 2/4 = 1/2$

Let's write code for this together and derive (or "learn") a **bigram** model from some of our movie plots from ES11

■ Trigram model

- Let **V** be our vocabulary = the set of distinct tokens
- Then a **trigram** model is, for each pair $v_1, v_2 \in V$ (the previous tokens), a probability distribution over **V** (the next token)
- A natural way to derive a trigram model from a given text is to compute the number of occurrences of each sequence of **three** tokens from **V** ... a sequence of three tokens is called a **trigram**
- For example: $V = \{ "a", "b" \}$, Text = "abaaaaabbba"

$$\begin{aligned} \#aaa &= 3, \#aab = 1 &\Rightarrow \Pr(a|aa) &= \frac{3}{4}, \Pr(b|aa) &= \frac{1}{4} \\ \#aba &= 1, \#abb = 1 &\Rightarrow \Pr(a|ab) &= \frac{1}{2}, \Pr(b|ab) &= \frac{1}{2} \\ \#baa &= 1, \#bab = 0 &\Rightarrow \Pr(a|ba) &= 1, \Pr(b|ba) &= 0 \\ \#bba &= 1, \#bbb = 1 &\Rightarrow \Pr(a|bb) &= \frac{1}{2}, \Pr(b|bb) &= \frac{1}{2} \end{aligned}$$

Let's write code for this together and derive (or "learn") a **trigram** model from some of our movie plots from ES11

■ N-gram model

$$|V| = 100$$

$$n = 1 : 100^1 \text{ unigrams}$$

$$n = 2 : 100^2 = 10k \text{ bigrams}$$

$$n = 3 : 100^3 = 1M \text{ trigrams}$$

- When generalizing this to **n**-grams, for arbitrary **n** $\in \mathbf{N}$, there are two problems:

1. The number of **n**-grams for a vocabulary of size **|V|** becomes infeasibly large for large **n**, namely: $\underbrace{|V| \cdot \dots \cdot |V|}_{n \text{ many}} = |V|^n$

Solution: neural n-gram models, see next slides

2. Many sequences of **n – 1** tokens will not occur at all, so when we encounter them we cannot predict the next token

Solution: pick the next token at random (called "smoothing", because we change the probability distribution)

■ Neural language models

- A neural language model computes a probability distribution over V (the next token), from given $v_1, \dots, v_n \in V$ (the previous tokens), just like an n -gram model
- But **not** via simple counts, like we have seen on the previous slides, but with a forward pass through a neural network

We discuss neural language models in depth in the WS 26/27

For now, understand that it is the same kind of model as on the previously slides, just trained and computed differently

- When the neural network computing the distribution is very large, these models are called **large language models**

■ "Base" models vs. "Instruct" models

- The n-gram or language models we have seen so far, predict the next token from the previous n tokens

These are sometimes called **base** models

- In practice, people want models that answer questions or talk to them similarly as a human would

These are sometimes called **instruct** models

They are also n-gram models, just trained in way such that the produced tokens feel more like a human conversation

- Instruct models typical have a so-called **system prompt**, which tells the n-gram model how to "behave"

See the next slide for how this is used in practice

■ Neural language models, terminology

- The number n of previous tokens is called the **context size**

Unlike our simple count-based n -gram models, neural language models enable **very large** context sizes ... up to 1 million

- The sequence of previous tokens is called the **prompt**

For prompts with more than n tokens, simply consider the last n tokens ... or see slides 14 – 17

- LLM services often have an additional **system prompt**

The tokens for a particular request are then simply the concatenation of the system prompt and the request prompt

■ Using an LLM for Exercise Sheet 12

- For ES12, we provide an LLM under the following URL
<https://ad-llm.cs.uni-freiburg.de/v1/chat/completions>
- These LLMs do not just provide the next token, but the next k tokens, using a HTTP request like the following

Content-Type: application/json

Authorization: Bearer YOUR_API_KEY

```
{ "model": "Qwen/Qwen3-Next-80B-A3B-Instruct-FP8",  
  "messages": [  
    { "role": "system", "content": SYSTEM_PROMPT },  
    { "role": "user", "content": PROMPT },  
  ],  
  "max_completion_tokens": k, "temperature": 0.7 }
```

■ Motivation

- The context size of existing language models is very large
400 k tokens for GPT-5, and **1 M** tokens for Gemini Pro
- However, there are many applications, where the want to query text that is larger than that

For ES12, we provide the transcripts of the recordings of all the lectures of this course (including the one today)

The length of these transcripts is larger than the context size of the LLM we provide at <https://ad-llm.cs.uni-freiburg.de>

- Idea: based on the query, extract a subset of the text, and pass that to the LLM, together with a suitable prompt

■ Preprocessing

- Split the (large) input text into (possibly overlapping) chunks

For ES12, take each paragraph from the subtitles as one chunk
(there is one paragraph for each timestamp)

Extend each chunk such that the sentences are complete
(search for nearest sentence delimiter to the left and right)

- For each chunk, compute an embedding of the kind we have seen in Lecture 11 and Exercise Sheet 11

For ES12, remove the timestamps when computing the embedding (they do not carry interesting information for that purpose)

■ Computing the embeddings

- For Exercise Sheet 11, we have computed the embeddings of text as the sum of the embeddings of the contained words

That is simplistic and does not give the best representation for a piece of text (not each word is equally relevant for the text)

- For Exercise Sheet 12, we provide another API that computes the embeddings for a given piece of text

<https://ad-ilm.cs.uni-freiburg.de/v1/embeddings>

Content-type: application/json

Authorization: Bearer YOUR_API_KEY

```
{ "model": "mixedbread-ai/mxbai-embed-large-v1",  
  "input": "Please compute the embedding for this text" }
```

■ Processing a query

- For a given query or prompt, first compute an embedding
For ES12: using the same API as for the text chunks
- Compute the **k** embeddings with the smallest cosine similarity
For ES12, you can do this in linear time; for real applications, you would precompute a special data structure for this
- Pass these **k** embeddings to the LLM, together with a suitable system prompt

For ES12, play around with different values for **k**, different chunk sizes, and different system prompts

References

■ Wikipedia

- N-gram models

https://en.wikipedia.org/wiki/Word_n-gram_language_model

- Large language models

https://en.wikipedia.org/wiki/Large_language_model

- Retrieval-augmented generation

https://en.wikipedia.org/wiki/Retrieval-augmented_generation