Exercises on *n*-gram language models

Ion Androutsopoulos, 2025–26

Submit as a group of 2–3 members (unless specified otherwise in the lectures) a report for exercise 3 (max. 10 pages, PDF format). Explain briefly in the report the algorithms/methods that you used, how they were trained, how you tuned any hyper-parameters, how you prepared your data etc. Present in the report your experimental results and demos (e.g., screenshots) showing how your code works. Do not include code in the report, but include a link to a Colab notebook (https://colab.research.google.com/) containing your code. Make sure to divide fairly the work of your group to its members and describe in your report the contribution of each member. The contribution of each member will also be checked during the oral examination of your submission. For delayed submissions, one point will be subtracted per day of delay.

1. Calculate the entropy in the cases of slide 36 (e-mail messages). Hint: for $P(C) \to 0$, use L'Hôpital's rule.

Answer:

a) When the training examples comprise 200 spam και 600 ham messages, then:

$$P(C=1) = \frac{200}{800} = \frac{1}{4},\ P(C=0) = \frac{600}{800} = \frac{3}{4},$$
$$log_2 \frac{1}{4} = -2,\ log_2 \frac{3}{4} = log_2 3 - log_2 4 = 1.585 - 2 = -0.415$$

$$H(C) = \frac{1}{4} \cdot 2 + \frac{3}{4} \cdot 0.415 = 0.811$$

b) When the training examples comprise 400 spam και 400 ham messages, then:

$$P(C=1) = \frac{400}{800} = \frac{1}{2},\ P(C=0) = \frac{400}{800} = \frac{1}{2},\ log_2 \frac{1}{2} = -1,$$

$$H(C) = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 1 = 1$$

c) When the training examples are spam, then:

$$P(C=1) = 1,\ P(C=0) = 0,\ log_2 P(C=1) = 0,\ log_2 P(C=0) \to -\infty$$

$$H(C) = -P(C=1) \cdot log_2 P(C=1) - P(C=0) \cdot log_2 P(C=0) =$$

$$= -1 \cdot 0 - \frac{log_2 P(C=0)}{\frac{1}{P(C=0)}}$$

Using De L'Hôpital's rule for the second term, for $P(C=0) \to 0^+$, we get:

$$H(C) \to -\frac{\frac{1}{P(C=0)\ln 2}}{-\frac{1}{P(C=0)^2}} = \frac{P(C=0)}{\ln 2} = 0$$

Similarly, when all the training examples are ham, again $H(C) = 0$.

2. [Optional.] (i) Implement (in any programing language) the dynamic programing algorithm that computes the Levenshtein distance (slides 54–59); your implementation should print tables like the one of slide 59 (without the arrows). You may want to compare the outputs of your implementation to those of http://www.let.rug.nl/~kleiweg/lev/. (ii) Extend your implementation to accept as input a word *w*, a vocabulary *V* (e.g., words that occur at least 10 times in a corpus), and a maximum distance *d*, and return the words of *V* whose Levenshtein distance to *w* is up to *d*.

→ **3.** (i) Implement (in any programming language) a bigram and a trigram language model for sentences, using Laplace smoothing (slide 7) or optionally (if you are very keen) Kneser-Ney smoothing (slides 49–50), which is much better. In practice, *n*-gram language models compute the sum of the logarithms of the *n*-gram probabilities of each sequence, instead of their product (why?) and you should do the same. Assume that each sentence starts with the pseudo-token **start** (or two pseudo-tokens **start1**, **start2** for the trigram model) and ends with the pseudo-token **end**. Train your models on a training subset of a corpus (e.g., a subset of a corpus included in NLTK – see http://www.nltk.org/). Include in the vocabulary only words that occur, e.g., at least 10 times in the *training* subset. Use the same vocabulary in the bigram and trigram models. Replace all out-of-vocabulary (OOV) words (in the training, development, test subsets) by a special token **UNK**.

(ii) Estimate the cross-entropy and perplexity of your two models on a test subset of the corpus, treating the entire test subset as a single sequence of sentences, with **start** (or **start1**, **start2**) at the beginning of each sentence, and **end** at the end of each sentence. Do not include probabilities of the form $P(*start*|...)$ or $P(*start1*|...)$, $P(*start2*|...)$ in the computation of cross-entropy and perplexity, since we are not predicting the start pseudo-tokens; but include probabilities of the form $P(*end*|...)$, since we do want to be able to predict if a word will be the last one of a sentence. You must also count **end** tokens (but not **start**, **start1**, **start2** tokens) in the total length *N* of the test corpus.

(iii) Write some code to show how your bigram and trigram language models can auto-complete an incomplete sentence. For example, given "I would like to commend the" (slide 43), generate completions like "rapporteur on his work **end**" or "president of the Commission on his intervention **end**". You can simply use the most probable next word (according to your language model) at each time-step (greedy decoding). If you are keen, you may also want to use beam search, or methods like top-*K* or nucleus sampling, to improve the diversity of the texts you generate.[1] In any case, do not allow **UNK** to be generated (e.g., if it is the most probable next word and you are using greedy decoding, generate the second most probable word). Confirm (showing some examples of generated texts) that your trigram model generates more fluent texts than your bigram model. Show also some interesting cases (e.g., good and bad sentence completions) when using your two models.

You are allowed to use NLTK (http://www.nltk.org/) or other tools and libraries for sentence splitting, tokenization, counting *n*-grams, but you should write your own code for everything else (e.g., estimating probabilities, computing cross-entropy and perplexity, greedy or beam search decoding). You may want to compare, however, the cross-entropy and perplexity results of your implementation to results obtained by using existing code (e.g., from NLTK or other toolkits).

Do not forget to include in your report:

[1] See, for example, https://towardsdatascience.com/decoding-strategies-that-you-need-to-know-for-response-generation-ba95ee0faadc.

- A short description of the algorithms/methods that you used, including a discussion of how you tuned any hyper-parameters, and how you prepared your data (e.g., any preprocessing steps, training/development/test splits).
- Cross-entropy and perplexity scores for each model (bigram, trigram) for sub-question (ii).
- Input/output examples (e.g., screenshots) demonstrating how your sentence completion works, including interesting cases (e.g., good and bad completions) for sub-question (iii).

4. (a) Using the following sentences as a tiny training corpus:

<start> he plays football
<start> he plays cricket
<start> she enjoys good football
<start> she plays good music
<start> he prays to god
<start> please buy me the other ball
<start> he pleases the other players by playing good football

estimate the probabilities $P(t_1^4)$ that a bigram model with Laplace smoothing would return, for each one of the following sentences t_1^4:

t_1^4: <start> he please god football
t_1^4: <start> he plays good football

Assume that the vocabulary V contains all the words of the training corpus (excluding <start>), hence $|V| = 21$. You do not need to perform numeric calculations that can easily be computed with a calculator.

Answer:

The bigram language model estimates *P*(<start>, he, please, god, football) as follows:

P(he | <start>) *P*(please | he) *P*(god | please) *P*(football | god) =

(4+1)/(7+21) (0+1)/(4+21) (0+1)/(1+21) (0+1)/(1+21)

where we used Laplace smoothing.

Similarly for *P*(<start>, he, plays, good, football). (Write down the calculations by yourself.)

Note: In practice, we avoid multiplying probabilities, because they often lead to very small numbers that cannot be represented well by computers. Instead, we usually compute the logarithm of the probability of a word sequence, i.e., we would compute log *P*(<start>, he, please, god, football), instead of *P*(<start>, he, please, god, football), which would lead to the following sum of four logarithms, instead of a product of four probabilities.

log[(4+1)/(7+21)] + log[(0+1)/(4+21)] + log[(0+1)/(1+21)] + log[(0+1)/(1+21)]

(b) Assume that a user wrote the sequence w_1^4 using the keyboard of a mobile phone:

w_1^4: <start> he pls gd ftball

Estimate the probabilities $P(t_1^4|w_1^k)$ of the two hypotheses (word sequences the user might have intended to write) t_1^4 of sub-question (a), using a noisy channel model (see slides) and the language model of sub-question (a). Assume that $P(w_i|t_i) \cong \frac{1}{LD(w_i,t_i)+1}$, where $LD(w_i, t_i)$ is Levenshtein distance from word w_i to t_i. Show your calculations in detail, without computing the Levenshtein distances.

Answer: Using the noisy channel of the slides:

$P(t_1^4|w_1^4) = P(t_1^4)\ P(w_1^4|t_1^4)\ /\ P(w_1^4)$

For t_1^4= <start> he please god football:

P(<start>, he, please, god, football)
P(<start> he pls gd ftball | <start>, he, please, god, football) / $P(w_1^4)$

The probability *P*(<start>, he, please, god, football) is estimated by the language model as in sub-question (a).

Assuming $P(w_i|t_i) \cong \frac{1}{LD(w_i,t_i)+1}$, the probability *P*(<start> he pls gd ftball | <start>, he, please, god, football) becomes:

P(he, he) *P*(pls, please) *P*(gd, god) *P*(ftball, football) =

1/(*LD*(he, he)+1) 1/(*LD*(pls, please)+1) 1/(*LD*(gd, god)+1) 1/(*LD*(ftball, football)+1)

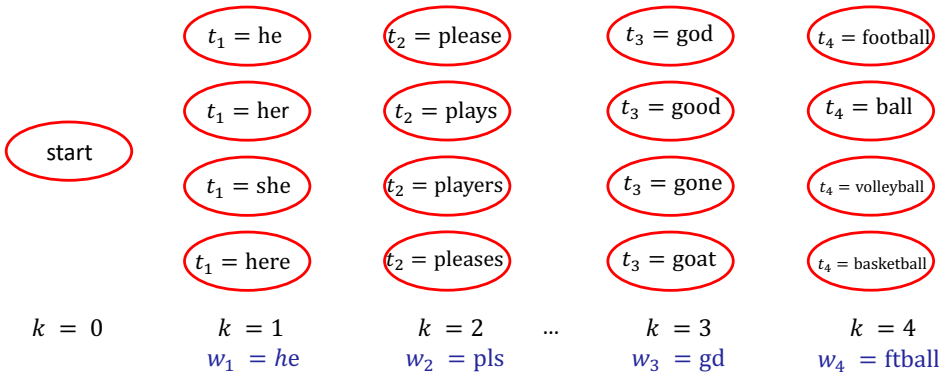
We don't need to compute $P(w_1^4)$, because it is the same for both hypotheses t_1^4 = <start> he please god football and t_1^4 = <start> he plays good football.

Similarly, we estimate $P(t_1^4|w_1^4)$ for the hypothesis t_1^4 = <start> he plays good football. (Write down the calculations by yourself.) Eventually, we select the hypothesis t_1^4 with the highest $P(t_1^4|w_1^4)$.

Note: In practice, we would again compute the logarithm of each product of probabilities, obtaining a sum of logarithms of probabilities, instead of a product of probabilities.

(c) Explain in detail how the beam search decoder of the slides would work in the following case. We use the noisy channel model of the previous sub-question, with the bigram language model of sub-question (a), trained on the tiny training corpus of that sub-question. The beam size $b = 2$.

Beam search decoder



For $k = 1$, the candidate paths t_1^k are the following. For each path, we compute $P(t_1^k)P(w_1^k|t_1^k)$. (In practice, we would compute the logarithm.) The $b = 2$ best paths are marked with asterisks.

<start, he>: P(he|start) P(he|he) = (4+1)/(7+21) 1/(0+1) = 5/28 = 0.179 **
<start, her>: P(her|start) P(he|her) = (0+1)/(7+21) 1/(1+1) = 1/28 1/2 = 0.018
<start, she>: P(she|start) P(he|she) = (2+1)/(7+21) 1/(1+1) = 3/28 1/2 = 0.054 **
<start, here>: P(here|start) P(he|here) = (0+1)/(7+21) 1/(2+1) = 1/28 1/3 = 0.012

For $k = 2$, the candidate paths t_1^k are the following. Again, we compute $P(t_1^k)P(w_1^k|t_1^k)$ for each path. Complete the calculations by yourselves. Mark the $b = 2$ best paths with asterisks. Notice that we can reuse computations (parts of the products) from $k = 1$.

<start, he, please>: P(he|start) P(he|he) P(please|he) P(pls|please) = …
<start, he, plays>: P(he|start) P(he|he) P(plays|he) P(pls|plays) = …
<start, he, players>: P(he|start) P(he|he) P(players|he) P(pls|players) = …
<start, he, pleases>: P(he|start) P(he|he) P(pleases|he) P(pls|pleases) = …
<start, she, please>: P(she|start) P(he|she) P(please|she) P(pls|please) = …
<start, she, plays>: P(she|start) P(he|she) P(plays|she) P(pls|plays) = …
<start, she, players>: P(she|start) P(he|she) P(players|she) P(pls|players) = …
<start, she, pleases>: P(she|start) P(he|she) P(pleases|she) P(pls|pleases) = …

For $k = 3$, the candidate paths t_1^k are the following. Again, we compute $P(t_1^k)P(w_1^k|t_1^k)$ for each path. Complete by yourselves. Mark the $b = 2$ best paths with asterisks

…

For $k = 4$, the candidate paths t_1^k are the following. Again, we compute $P(t_1^k)P(w_1^k|t_1^k)$ for each path. Complete by yourselves. Which hypothesis (path) <start, t_1, t_2, t_3, t_4> is selected?

…