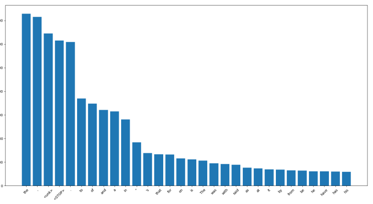
1 N-gram Language Models

1.1 N-gram Language Modeling & Perplexity

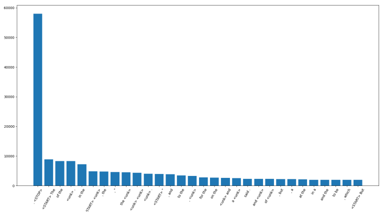
(a) To construct the N-gram models, we need to calculate the conditional distribution over the n-th token given the previous $n-1$ tokens. To do so, we count the number of occurrences of the string $x_i, \ldots, x_{i+n-1}$ and divide by the number of occurrences of $x_i, \ldots, x_{i+n-2}$.

For the Unigram model we basically count the number of occurrences of each word and divide by the sum of all occurrences of all words.

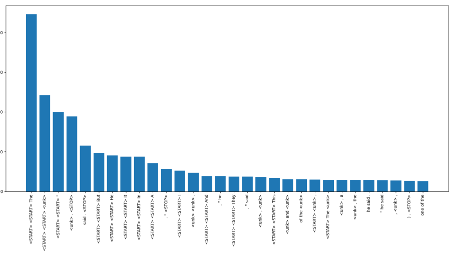
We can see the top 30 most frequent tokens for all 3 models below.



Unigram



Bigram



Trigram

Figure 1: Top 30 most frequent tokens

(b) To calculate the perplexity, we need to calculate the likelihood of each sentence in the given data under the probabilities given by the model, take the geometric mean of its inverse, and then calculate the average over the whole corpus.

There are two things that we need to keep in mind:

- Unobserved tokens: to solve this problem, we replace the tokens seen less than 3 times with the special <unk> token during training, and replace the unseen tokens with <unk> during test time. This would prevent undefined behavior (e.g divide by 0) for the unigram model.But the problem is still unsolved for bigram and trigram models, since there may still be bigrams/trigrams that we have not seen during training. To work around this problem we assume independence for the bigram model i.e. $p(x|y) = p(x)$ and the Markov assumption for the trigram model i.e. $p(x_t|x_{t-1}, x_{t-2}) = p(x_t|x_{t-1})$. With these assumptions, we will no linger witness undefined behavior during perplexity calculation.
- Conditioning: This problem only applies to bigram and trigram models since they calculate conditional probabilities. To work this problem around, we need to pad each sentence with $n-1$ <START> tokens.

(c) A potential downside of Laplace smoothing is increasing the perplexity. This lies in the relation between perplexity and entropy:

$$PP(W) = p(w_1, \dots, w_N)^{-\frac{1}{N}} = 2^{H(W)}$$

By smoothing, we are giving mass to the unobserved sequences, which could lead to an increase in entropy and thus an increase in perplexity as we will see in the results in section (e).

(d) We can see the effect of k in the plots below.

As we can see, trigram has lower perplexity than bigram, but both show the same increasing pattern over k.

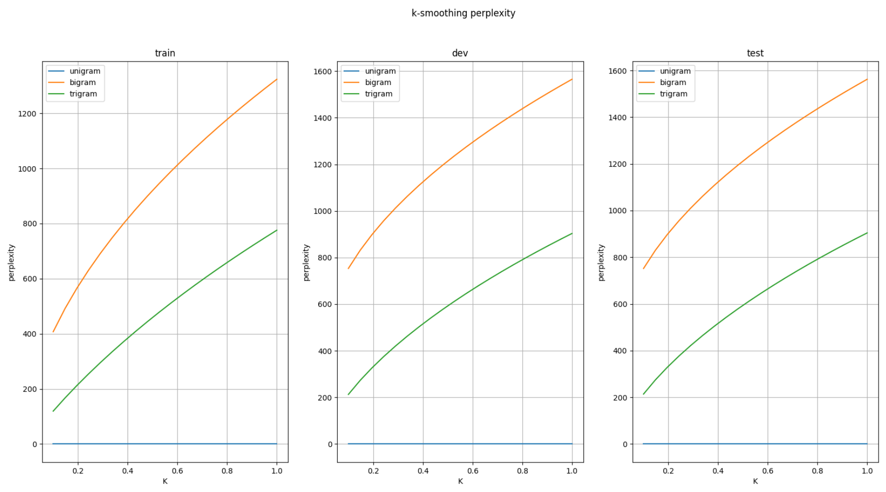


Figure 2: Top 30 most frequent tokens

(e) The results are given in the table below.

As wee can see, increasing context-length leads to perplexity improvement without smoothing.

As discussed in section (c), smoothing would increase perplexity; as we can see in the table.

	Smoothing	**No Smoothing**
Unigram	**Train**: $2.53e-05$ **Dev**: $2.5e-05$ **Test**: $2.5e-05$	**Train**: 1080.36 **Dev**: 985.55 **Test**: 998.72
Bigram	**Train**: 1323.57 **Dev**: 1565.41 **Test**: 1562.44	**Train**: 74.06 **Dev**: 193.59 **Test**: 194.49
Trigram	**Train**: 775.76 **Dev**: 903.56 **Test**: 904.22	**Train**: 8.37 **Dev**: 132.22 **Test**: 132.14

Table 1: n-gram perplexity

1.2 Interpolation

(a) The results are shown in the table below with the best result in boldface. My strategy for finding suitable lambdas was to try giving more attention to more/less recent tokens to see how it would affect perplexity. (note that these results come from the dev set)

λ_1	λ_2	λ_3	**perplexity**
0.33	0.33	0.33	151.89
0.6	0.3	0.1	190.79
0.3	0.6	0.1	157.67
0.3	0.1	0.6	148.6
0.1	**0.3**	**0.6**	**134.4**
0.6	0.1	0.3	184.38

(b) The same set that achieved the lowest perplexity on the dev set, achieves the lowest perplexity on the test set with a perplexity of 134.48.

(c) Halving the the training data leads the an increase in perplexity, this is because reducing sample size would make our estimates more biased, hence it would decrease the performance on unseen data as can be seen in the table below

λ_1	λ_2	λ_3	perplexity
0.33	0.33	0.33	167.99
0.6	0.3	0.1	206.39
0.3	0.6	0.1	172.1
0.3	0.1	0.6	165.88
0.1	0.3	0.6	150.44
0.6	0.1	0.3	202.25

(d) Setting a threshold of ≤ 5 would yield a lower entropy compared to ≤ 1, thus it would give lower perplexity compared to ≤ 1 under similar conditions as can be seen below.

λ_1	λ_2	λ_3	perplexity
0.33	0.33	0.33	169.86
0.6	0.3	0.1	212.63
0.3	0.6	0.1	175.89
0.3	0.1	0.6	166.54
0.1	0.3	0.6	150.52
0.6	0.1	0.3	206.07

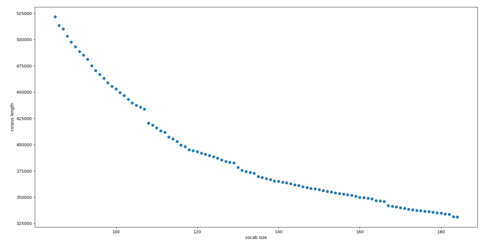
Table 2: threshold=1

λ_1	λ_2	λ_3	perplexity
0.33	0.33	0.33	123.75
0.6	0.3	0.1	156.06
0.3	0.6	0.1	129.18
0.3	0.1	0.6	120.57
0.1	0.3	0.6	109.51
0.6	0.1	0.3	149.94

Table 3: threshold=5

2 Byte-Pair Encoding (BPE)

(a) The training data is 147408 tokens long under the final vocab. The final vocab size is 4873. below we can see the change in vocab size and corpus length over the first 100 iterations.

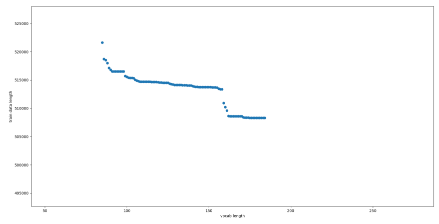


(b) An advantage is that it is simpler to implement than BPE, and a potential downside is that the number of tokens would be much more compared to BPE, which means that it is less efficient in terms of memory usage.

(c) We end up with 3265 tokens in total. No Issues witnessed regarding the words not in the training set. A potential scenario that may cause issues is when we train the encoder on one domain and apply it to another one e.g. training on English text and applying to Japanese could cause problems, since the two differ in alphabets.

3 WordPiece

(a) We can see the plot over the first 100 iterations below. The final vocab size is 4085, and the final length of the training data is 368607.



(b) The test data is 89374 tokens long.The tokenized sequences of the sentences are as follows:

['A', 'n', 'a', 'l', 'y', 's', 't', 's', ' ', 'we', 're', ' ', 'e', 'x', 'pecting', ' ', 't', 'he', ' ', 'o', 'p', 'posit', 'e', ',', ' ', 'a', ' ', 'de', 'e', 'pen', 'ing', ' ', 'o', 'f', ' ', 't', 'he', ' ', 'de', 'ficit', '.']

['F', 'i', 've', ' ', 'minutes', ' ', 'la', 'ter', ',', ' ', 'a', ' ', 's', 'e', 'c', 'on', 'd', ' ', 'person', ' ', 'a', 'r', 'ri', 've', 'd', ',', ' ', 'a', 'ge', 'd', ' ', 'a', 'ro', 'u', 'n', 'd', ' ', 't', 'hi', 'r', 't', 'y', ',', ' ', 'w', 'i', 't', 'h', ' ', 'k', 'n', 'i', 'fe', ' ', 'wounds', '.']

(c) When applied to the training data, BPE gave us 147408 tokens, while WordPiece gave us more than 300000 tokens, which implies that BPE is more efficient in that regard. At inference time, it took BPE 0.02 seconds to be applied on the test data, while it took WordPiece more than 5 seconds.

4 Open-ended Exploration for Tokenization

(a) Our sample sentence is:

”If you get on the side where all the hot-shots are, then it’s a game, all right—I’ll admit that. But if you get on the other side, where there aren’t any hot-shots, then what’s a game about it? Nothing. No game.

-Holden Caulfield”

We use Claude and Llama3 for tokenization, and we pick Persian as the second language.

	English	Persian
Claude	70	165
Llama3	63	65

Table 4: Number of tokens

The translation was done by myself since Persian is my mother tongue.

(b) Another scenario which could lead to such a problem is when a single word in one language needs more words of another language to be described. Take the word ”wimpy” as an example, which is a single word in English, but there is a Persian synonym with 4 words.

(c) If not handled well, the time it takes for the model to tokenize a given text would vary accross different languages, which means that it could possibly take much longer for some languages, in which case we wouldn’t be able to attract the speakers of those languages because of the poor user experience.

(d) Although it may not be the best one, but one possible solution could be training tokenizers tailored to each specific language, in which way our tokenizers would be more efficient since they are optimized for that specific language.