Attached files: lm.ipynb including codes and execution results and lm.html code and execution output for better readability

About the data:

The file heartdisease.txt contains the text of a number of Persian blog pages about heart diseases. This collection has been collected manually and includes information such as symptoms of heart diseases, treatment methods, how the human heart works and structures, and similar information in the field of general medical information about health and heart disease. This file only contains the copied and pasted text of these texts and no processing has been done on it. The file is read line by line and stored on text.

First step: pre-processing of Persian text

I use the parsivar library to pre-process the Persian language. First, I normalize each line of text. Text normalization converts characters that have multiple Persian or Persian-Arabic writing forms into a standard Persian character type. Adjusts spaces with punctuation marks and spaces and half spaces. The normalized text is in the text_norm list. With parsivar, I unify the text as a set of sentences. Then, with another function of the same Tokenizer model, I unify each text sentence into a list of its component words. The tokenized_sentences list is a collection of these lists. Because in the tokenization of punctuation marks, each one was taken as a token and I do not need punctuation marks in the language modeling that I am going to do, I will remove all the tokens that are in tokenized_sentences and only have one punctuation mark from the list. . I have a total of 2696 lists (unitized sentences). In order to have two test and training sets, I do a random sampling from this list and take 200 sentences, this is the evaluation set, and what remains is the training set. I count the number of tokens in each list and I have 2458 and 31271 words/tokens in each set respectively.

Second step: language modeling without smoothing and then adding various smoothing functions

Despite the recommendation to work with the mentioned expressions in the case of practice, including SRILM, because I had problems in the process of installing these tools and could not learn how to work with them, I used the nltk library for modeling. It should be noted that parts of this code can be executed with the latest version of this nltk, and since this version has problems with the current version of parsivar, I suggest that before running this section, temporarily disable parsivar and nltk upgrade to the new version.

The n_gramModel function is almost constant in all parts of the work and adds <s> and <\s> tokens to the beginning and end of the training text in all stages for the requested n-gram (unigram, bigram, trigram). And then it converts the matrix of tokenized sentences into a one-dimensional list. It extracts and counts all the combinations of n-gram and finally by calculating the probabilities, it prints the number of words with which the model is built and returns the language model. Because I work with a small dataset, I set the number of words in the language model equal to all the words that appear more than once in the training text. For unvisited words the model uses <unk> as a replacement. The only difference between this function in different parts of the work is the use of different smoothing methods. All these steps are done with the models and functions of the nltk library. In addition, in order to make the obtained model more reliable, instead of counting only n-grams, m-grams are counted for m<=n. For the first part, n_gramModel works with an MLE model that does not perform any smoothing. In this first part, outside of this function, in order to prepare the test set, I first added > and <\s> tokens for the beginning and end of the text, and then converted it to a one-dimensional list, and for all three models, unigram, bigram , I extracted trigram tokens to be used to calculate entropy and confusion.

In the continuation of the work and in each step, I used a smoothing method that executes the same n_gramModel function by changing the language model used and calculates confusion and entropy every time. In the table below, the name of each used smoothing method is listed in the attached file, and its results are also mentioned.

No smoothing		
unigram	bigram	trigram

`model vocabulary = 3614`
`perplexity = inf`
`entropy = inf`
	`model vocabulary = 3616`
	`perplexity = inf`
	`entropy = inf`
		`model vocabulary = 3616`
		`perplexity = inf`
		`entropy = inf`

Lidstone یا add-k		
unigram	bigram	trigram

`perplexity = 656.4`
`entropy = 9.4`
	`perplexity = 1552.8`
	`entropy = 10.6`
		`perplexity = 2714.1`
		`entropy = 11.4`

Laplace یا add-one		
unigram	bigram	trigram

`perplexity = 656.4`
`entropy = 9.4`
	`perplexity = 1219.9`
	`entropy = 10.3`
		`perplexity = 2302.1`
		`entropy = 11.2`

KneserNey		
unigram	bigram	trigram

`Due to the error of division by zero, perplexity and entropy could not be calculated.`	`perplexity = inf`
	`entropy = inf`
		`perplexity = inf`
		`entropy = inf`

WittenBell		
unigram	bigram	trigram

`perplexity = inf`
`entropy = inf`
	`perplexity = inf`
	`entropy = inf`
		`perplexity = inf`
		`entropy = inf`

Backoff (Stupid Backoff)		
unigram	bigram	trigram

`perplexity = inf`
`entropy = inf`
	`perplexity = inf`
	`entropy = inf`
		`perplexity = inf`
		`entropy = inf`

AbsoluteDiscounting		

unigram	bigram	trigram

`perplexity = inf`
`entropy = inf`
	`perplexity = inf`
	`entropy = inf`
		`perplexity = inf`
		`entropy = inf`

Except for the two modes add-one and add-k (I set k equal to 0.01), in all other modes, the model gave an infinite value for the entropy and confusion number for all unigrams, bigrams, and trigrams. This is an imaginable result for moles without smoothing, but I did not expect such a result in cases where smoothing methods were also used. I imagine that because I am not familiar with modeling and I am working with nltk for the first time, maybe I have made a mistake in using functions and models, and in fact all these results are the same results without smoothing! But I did not understand where this error comes from or whether this idea is correct or not. On the other hand, if the words of the test text are unfamiliar to the model or if the number of words with which the text can be modeled is large compared to the size of the training set, the confusion of the model will be very high and inf because the words that are in the test are all in the model. will be unknown.

Therefore, all the analysis that I can provide from this modeling is related to the comparison of the two smoothing modes add-one and add-k, which were discussed in the class, and their difference with the mole without smoothing. Smoothing is basically used to reduce the zeros of the n-gram count matrix and their probabilities, so when the result is better than the first case with both methods, it is completely predictable. In both Laplace and Leadstone methods, a value is added to the counts to eliminate the zeros of the probability matrices. In Laplace, this added value is 1, and in Leadstone, to improve Laplace's method (the problem of generating large numbers), this added value is less than one, and here I got 0.01. Both of them give equal evaluation values on unigram, but when I modeled with higher degrees of n-gram, the difference was evident, and this difference is evident both in the logarithm scale (entropy) and in a more significant way on the confusion itself, which is used with trigram and bigram. From Laplace of add-k, k=0.01 has a better result and gives less confusing results on this text.

In all of these 6 reported cases, friction and, by nature, entropy increase with more n. The best model I got in this work is the unigram with Laplace.