Arbitrary Precision and Arbitrary n-Grams

Alec Story and Thomas Levine
avs38 and tkl22

February 28, 2011

Implementation

Our design is built around dictionaries for each n-gram we consider. We break the input text into words and add sentence separators (although we do not break the sentences apart, so trigrams or higher can cross sentence boundaries), and then process the list of words, tallying the number of each n-gram seen in the dictionary.

We also can optionally account for unknown words by replacing the first instance of each word with an unknown symbol; this is simply an additional processing step applied to the raw list of words.

We compute all n-gram count dictionaries up to the desired n-gram length, and use these to compute the probabilities of each n-gram. This dictionary of probabilities is used to construct sentences and to measure perplexity. We either use logarithmic or arbitrary-precision arithmetic depending on what the user specifies.

We implemented Good-Turing smoothing, which adjusts the n-gram counts before they are turned into probabilities. Like unknown-handling, this is simply a pipeline processing step.

Sentence generation begins with a beginning of sentence symbol, and then chooses words from the n-gram dictionary corresponding to the minimum of the desired n-gram length or the number of words we have already created.

Perplexity works similarly, scanning the input list and computing the running sum or product probability, depending on whether we are using arbitrary precision or logarithmic arithmetic. Because computing the perplexity for each word is simply a matter of looking up a tuple in the appropriate dictionary (possibly containing an unknown), this step is fairly rapid. If we do not have the appropriate n-gram in the dictionary, we first try replacing the last word with an unknown, and, if that fails, we shorten the list of words until a match is found, which, at shortest, is a single unknown by itself.

The dictionary approach was simple and intuitive, but fairly slow; the generation of sentences spends the majority of its time on dictionary operations, and uses only moderate amounts of memory. Much of the time in sentence generation is spent computing the list of n-gram options to choose from, since this requires, in our implementation, scanning through the entire n-gram dictionary to find matches for the $(n-1)$-gram. If we precomputed this as we populated the dictionary, substantial time could be saved, although choosing a random element from that list still takes $O(n)$ time.

Sentence Generation

For sentence generation, we did not include our unknown processing, and do not convert the first occurrence of each word to an unknown token.

Each subsection contains two examples from the Shakespeare corpus and two from War and Peace. These corpora were chosen because they were determined at rough glance to be the easiest to parse.

Unsmoothed

Unigrams

> Bring in Take made the pass a Caesar in gold well leave

> Exeunt . hope , and Milan your A Will thy have Come the . . advantage , together your either to '

> He fell other along " therefore , and had about He chief shorten now him life . something

> I had transference shrugged man with you The lit . to lift the her had he . and have the hand individuals dragging the , place been to Natasha in inside . revelry hide nation room beaming eager all of the the , remounts to

Bigrams

> Enter LYSANDER , and you lie dull actor on my speech , an ordinary pitch , Loyal and full of our tongues , see , That which physic to my horse for her youth and sealed bags Of base earth ; Though twenty several mistress will you have done To have found .

> Silence , who .

> The horses ' familiar to the army , envy and looked questioningly : "You won't be thus depriving him of the estate with his will send me forty thousand men in choosing her governess kept saying good-by .

> I have begun to Tushin and yet further battles , the man in the defense of that temple , they were standing beside whom you seen in the battle of dry branches of meeting it has brought the smile Berg with a mocking , and even impossible to be long paced angrily .

Trigrams

How - traitor ?

The Greeks upon advice , Hath alter'd that good wisdom Whereof I know by their christen names , Unless thou tell her so .

No , but it can't be helped It happens to the first to arrive , Princess , I think .

Rostopchin , coming to Petersburg , look out at night his feet .

Smoothed

Unigram

th ! come Lear things of the I

me . I

they its intimate just such But There the them said shout irresistible always heard none with and of dinner had To

day good-natured Secondly same , joyful on talking the appeared advantage . unreasoning men by

Bigram

Hector .

I arrest thee here , which time , that his fault , and blows .

But my wife should have nobody and ask the enemy was too .

Say it's farther .

Trigram

She would have made fair work !

Come , my lord , I'll bring you to please his Majesty , Herod of Jewry dare not say he is much abus'd with tears thou keep'st command .

All that he had no news from the chase .

If I decline the honor of the bushes .

Trigrams lead to more comprehensible sentences, which is to be expected, and bigrams are barely useful. The unigrams also seemed to produce longer sentences; it's possible that trigrams and bigrams lead to a progression from words that are likely at the beginning of sentences to words that are likely at the end, and then to end the sentence, despite sentence ends being fairly infrequent. Additionally, *War and Peace* seems to produce longer sentences (particularly when observed over more iterations than can be reproduced comfortably here), probably because the sentences in the original are longer, so sentence ends are rarer.

Our n-gram generator only treated sentence endings by adding a single symbol, '#', and did not break the input into individual sentences, nor do we break the perplexity input into sentences in the same manner.

Arbitrary Precision

One of the two extensions we chose was to implement the probability and perplexity calculations using arbitrary precision arithmetic (we also implemented it using logarithmic arithmetic). This can lead to slower processing, but guarantees that there is no data loss until we move into processing floats during exponentiation.

We used python's `gmpy` package to represent the probabilities as arbitrary rational numbers (python's integers are already arbitrary-precision), and `mpmath` to perform the final exponentiation to calculate the perplexity (the n-th root). The libraries could only compute non-integer powers of integers or floats, and integer powers were rounded to the nearest integer, and therefore, inaccurate, so (very high precision) floats provided the best results. This final stage of computation of perplexity was the only point where imprecise mathematics was used. See table 1

Arbitrary n-grams

Because the computations were general enough, we were able to easily abstract the code to produce count, smoothed count, and probability tables for arbitrary n-grams. We also compute all of the smaller n-grams, which are required for generating or analyzing the beginning of a sequence of words, and for other computations.

Large n-grams (particularly $n > 4$) tend to result in whole Shakespeare sentence fragments. For example, "O , my offence is rank , it smells to heaven ; It hath the primal eldest curse upon't , A brother's murther !" was generated by a hexagram, and is a direct quote from Hamlet, act 3 scene 3. This is a well-documented phenomenon, and is due to the fact that it's quite possible that the preceding $(n-1)$-gram appears only once in the entire corpus, so the sentence generator has no choice but to choose the next word in the quotation.

Experimental design

We trained our model on the training data and then ran two experiments to study the effect of n-gram length and probability computation type on perplexity on the two corpora. The two probability computation types were arbitrary precision arithmetic and logarithmic arithmetic. For each experiment, we tested n-gram length of n=1 to 5.

We hypothesized that longer n-grams would yield lower perplexity and that the two probability computation types would yield similar perplexity.

Corpus	n	smoothing	math	perplexity
Shakespeare	1	on	log	499.299251351
Shakespeare	1	on	ap	499.299251353146
Shakespeare	1	off	log	487.213020203
Shakespeare	1	off	ap	487.213020203167
War and peace	1	on	log	439.45958473
War and peace	1	on	ap	439.459584729599
War and peace	1	off	log	431.095976221
War and peace	1	off	ap	431.095976221135
Shakespeare	2	on	log	61.1651128384
Shakespeare	2	on	ap	61.1651128383365
Shakespeare	2	off	log	53.723243187
Shakespeare	2	off	ap	53.7232431870133
War and peace	2	on	log	50.5728265194
War and peace	2	on	ap	50.5728265193686
War and peace	2	off	log	44.4939985115
War and peace	2	off	ap	44.4939985115293
Shakespeare	3	on	log	42.3740593974
Shakespeare	3	on	ap	42.3740593973802
Shakespeare	3	off	log	29.644105241
Shakespeare	3	off	ap	29.644105240972
War and peace	3	on	log	33.5298099459
War and peace	3	on	ap	33.529809945862
War and peace	3	off	log	23.6337013897
War and peace	3	off	ap	23.6337013896575
Shakespeare	4	on	log	41.6859932104
Shakespeare	4	on	ap	41.6859932104407
Shakespeare	4	off	log	25.5850207553
Shakespeare	4	off	ap	25.5850207552466
War and peace	4	on	log	33.0110361869
War and peace	4	on	ap	33.0110361868792
War and peace	4	off	log	20.5857956417
War and peace	4	off	ap	20.5857956417055
Shakespeare	5	on	log	42.4661223625
Shakespeare	5	on	ap	42.4661223624809
Shakespeare	5	off	log	25.0344976529
Shakespeare	5	off	ap	25.034497652926
War and peace	5	on	log	33.6199337718
War and peace	5	on	ap	33.6199337717408
War and peace	5	off	log	20.0240585504
War and peace	5	off	ap	20.0240585503833
Shakespeare	6	on	log	42.7513356663
Shakespeare	6	on	ap	42.7513356662635
Shakespeare	6	off	log	24.9268403396
Shakespeare	6	off	ap	24.9268403395792

Table 1: Perplexity values for various n-gram parameters

Perplexity of test set

We evaluated our model on the test set by computing the perplexity of the test set. In addition to the variables above, we ran this test with smoothing on and with smoothing off.

Perplexity of generated sentences

We also evaluated our model on sentences generated by our model. We did this because sentence generation involves substantial probability computations, where the difference in probability computation type would be relevant.

For each combination of the independent variables, we generated 23 sentences from the *War and Peace* corpus and tested their perplexity. We did the same the Shakespeare corpus but generated only 7 sentences for each treatment combination.

Results

Perplexity of test set

For each corpus, perplexity of test set decreased when the models were run for longer n-grams, and the perplexity leveled off starting at $n = 3$ (figure 1). Probability computation type had no effect.

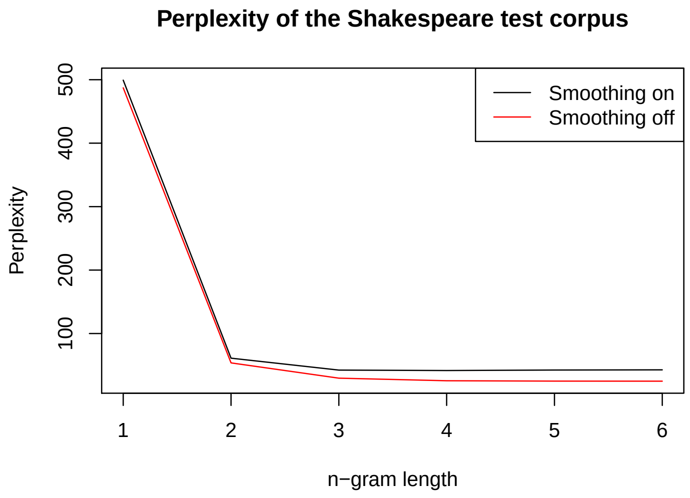
Smoothing slightly increased perplexity. The increase was larger for larger n.

Perplexity of generated sentences

Perplexity of generated sentences decreased with when longer n-grams were used in the model (figure 2). The perplexity leveled off starting at $n = 3$. Probability computation type did not affect perplexity.

Arbitrary precision arithmetic

Our hypothesis was that the arbitrary precision math would reveal a small amount of error in computing the perplexity logarithmically. Our results seem to indicate that any such error is very small (differing only in the last digit in measuring the perplexity of whole corpora), but that arbitrary-precision mathematics is much slower than floating point mathematics with logs when finding the product of many numbers. See table 2. This suggests that the logarithmic computation is indeed an accurate way to compute perplexity, particularly in light of the more than 10-fold speed improvements it results in.



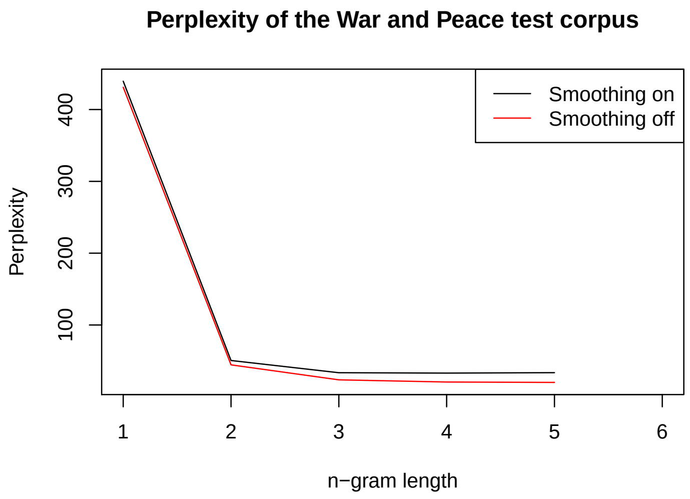


Figure 1: Perplexities of the test corpora by n-gram length and probability computation method for models trained on the respective training corpora.

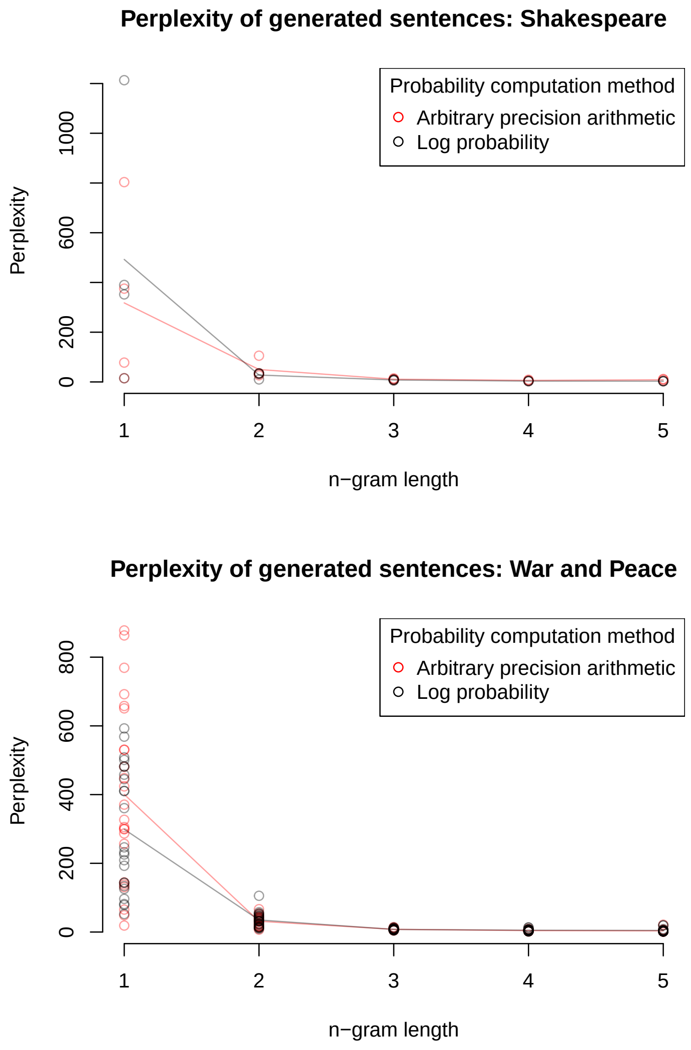


Figure 2: Perplexities of generated sentences by n-gram length and probability computation method for the two corpora.

Operation	Arbitrary Precision	Logarithmic
Trigram tallying	136.936949968	137.023640156
Probability computation	63.5648961067	60.1038038731
Perplexity computation	**990.322752953**	65.2168021202

Table 2: Running times with arbitrary and floating-point precision in seconds. Average of 10 trials, trained on the Shakespeare training corpus, perplexity calculated for Shakespeare training corpus

Conclusion

Arbitrary precision does not result in more accurate perplexity computations over logarithmic computation, at least to floating point precision on the output of the perplexity, and such precision is probably dwarfed by noise in the inputs. Such computations, however, take substantially longer to compute, so arbitrary precision computations are probably not worth performing.

As expected, lengthening n-grams to moderate values decreases perplexity of the test corpus, but increasing the n-gram length beyond 3 did not yield benefits, so it appears that trigrams are reasonable for use computing perplexity.