Language Models (n-gram and feed forward)

Jonathan May

September 8, 2022(Prepared for Fall 2022)

1 What they are and why they're useful

A language model is, formally, a probabilistic formal language, i.e. an n-tuple that includes a vocabulary Σ and contains mapping mechanism $\Sigma^* \rightarrow \mathbb{R}_{\geq 0}$. Furthermore, the sum of all (infinite) $\mathbf{x} = x_1, x_2, \ldots, x_n \in \Sigma^*$ should be 1. Practically speaking we usually want to answer the question "What is the probability of the next word?"

We are formally seeking $P(x_1, x_2, \ldots, x_n)$ so we can conveniently use the chain rule, insert start and end tokens x_0, x_{stop}, and re-express as $P(x_1|x_0)P(x_2|x_1, x_0)$
$P(x_3|x_2, x_1, x_0) \ldots P(x_{\text{stop}}|x_n, \ldots, x_0)$.

Why do we care? This can cover both syntax

$P($ the cat slept peacefully $) > P($ slept the peacefully cat$)$

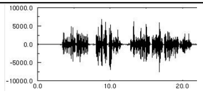
and semantics

$P($ she studies morphosyntax $) > P($ she studies more faux syntax $)$

Furthermore the notion can be generalized beyond a single sentence and model a continuous stream of language.

Language models help us to generate:

- translations
- spelling/grammar corrections
- summarizations
- text recognized from speech

Task	Input	Options	Final (post-LM)
spelling correction	no much effort	no much effect so much effort no much effort not much effort	not much effort
speech recognition		she studies morphosyntax she studies more faux syntax	she studies morphosyntax

		she's studies morph or syntax	
translation	ella se va a casa	she is going home she is going house she goes to home to home she is going	she is going home

These can also be used for prediction (type 'Where can I' into google, or start typing a text message).

This application comes out of the generative models we looked at before. If we wanted originally some $P(Y|X)$ this is equivalent to $P(X|Y)P(Y)/P(X)$ where $P(X|Y)$ can be thought of as a 'noisy channel' corrupting unobserved Y into X. Then $P(Y)$ is the language model the data creator used when generating Y before corrupting it into X, which is what is seen. ($P(X)$ isn't needed; we see X, we don't care about its likelihood). Y could be anything; it could be a sentiment (but that's not much of a language) or it could be a tag sequence, or a language sequence. We'll increasingly consider cases where it's a natural language sequence.

2 N-gram models

Similar to Naive Bayes, we can make an independence assumption, e.g. that $P(x_i|x_{i-1}, x_{i-2}, x_{i-3}, x_{i-4}) = P(x_i|x_{i-1}, x_{i-2})$ (trigram model). And we can estimate these conditional probabilities from data, just like we estimated conditions of word given label, before.

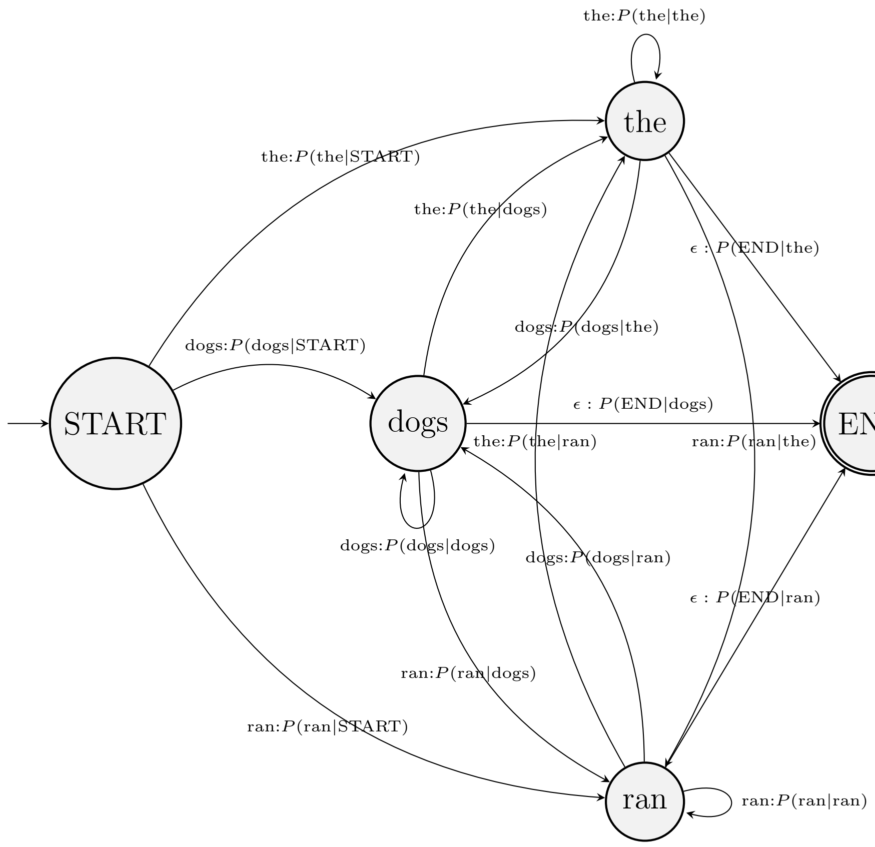
So if you want $P(\text{mast}|\text{before, the})$ we can use a corpus (say, Moby Dick) and unix tools from before:

```
sed 's/ /\n/g' mobydick.txt | grep -v "^$" > md.words
paste md.words.txt <(tail -n+2 md.words.txt) <(tail -n+3 md.words.txt) \
    | grep -ic "^before\tthe\t"
29
paste md.words.txt <(tail -n+2 md.words.txt) <(tail -n+3 md.words.txt) \
    | grep -ic "^before\tthe\tmast"
4
```

So, 4/29 = 13.8% (at least in nautical novels).

2.1 N-gram as FSA

FSAs are suitable for representing statistical n-gram language models; in fact this was actually how they were represented (e.g. back in pre-neural speech recognition days); there are nice properties of FSAs like closure under composition that allow you to chain little pieces of truth together. Here is an example of a bigram FSA:



If you had some other constraints (e.g. metrical constraints) you could also encode those as an FSA and then intersect the two to get, say, a sentence with appropriate meter and high n-gram likelihood.

2.2 Using N-gram language models

Language models can be used for both *evaluation* and *generation.*

For evaluation, imagine we had the following snippet of text and wanted to know its probability.

```
Call me Ishmael . Some years ago never mind how long precisely
```

A 3-gram language model would estimate this as $P(\text{Call}|\text{START},\text{START})P(\text{me}|\text{Call}, \text{START})P(\text{Ishmael}|\text{Call}, \text{me})P(.|\text{me}, \text{Ishmael})\ldots$ This could be used when comparing different alternatives, as above.

For generation, we proceed as follows: Let's say you've already started with `Call`. Then from the set of $P(\_|\text{Call}, \text{START})$, *sample* a word proportionally to the distribution. E.g. If we have:

x	$P(x\|\text{call})$
him	.179
it	.143
of	.071
the	.071
me	.071
all	.036
our	.036
...	...

you imagine a wheel where `him` takes up 17.9% of the wheel, `it` the next 14.3% and so on. You spin the wheel and choose the word you land on. Let's say you get `it`. Then you choose from $P(\_|\text{it, call})$ using a new table, and so on. Note that it's generally not a good idea to just take the most probable argument, nor is it a good idea to sample uniformly.

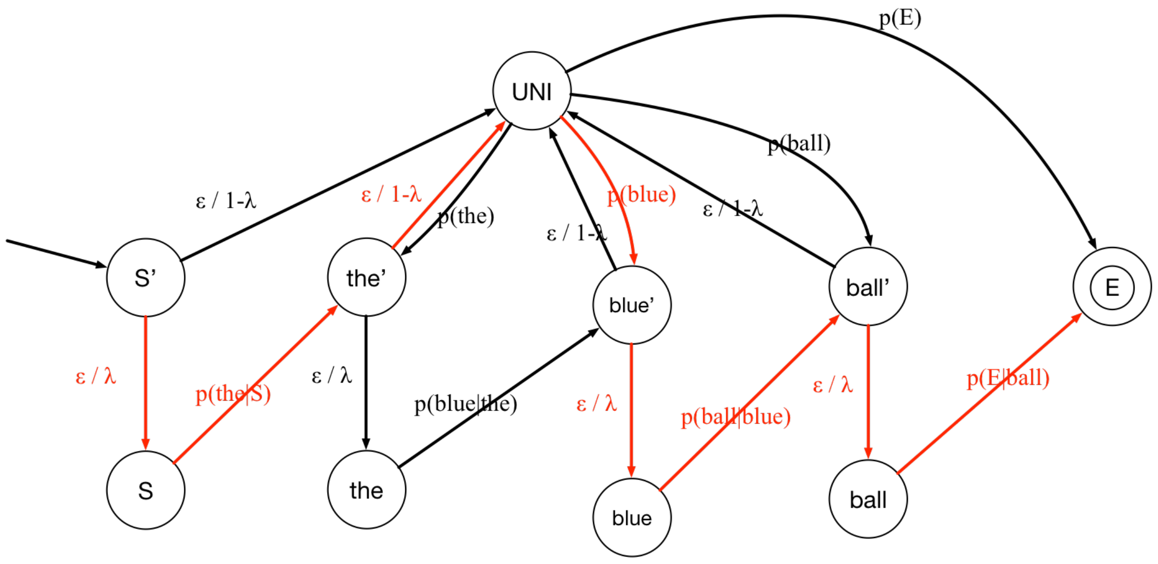
2.3 Problems with n-grams: sparsity, (Backoff and Smoothing) and storage

N-gram language models are of great utility but they have some problems that need handling. For one thing, they are quite *sparse*. 99.8% of the 5-grams in Moby Dick, for instance, occur exactly once[1].

What to do? One thing we can do is, as before, smooth. So if "go to sea as the" does not occur in training (it doesn't in Moby Dick) we can still add some small amount to each vocabulary term so we don't get zero probability for the whole sentence.

But what if "head to sea as a" does not occur because the context, "head to sea as __" does not occur? Explicitly smoothing every possible 4-gram would explode the memory needed to represent the language model. We instead (or really, additionally) condition on 3-gram context and interpolate between the two models, i.e. $\lambda P(a|4-gram)+(1-\lambda)P(a|3-gram)$. Even this can be represented as an FSA:

[1]The three most frequent 5-grams, each occurring four times, are [in the middle of the], [go to sea as a], and [' Queequeg,' said I, ']



Note another problem with n-gram language models is their size. There is a parameter (a probability) for every n-gram seen in training, plus for some n-grams not seen in training (due to smoothing), plus all $n-k$ grams for $k = 1$ to $n-1$ (due to backoff). More training data makes for better language models, but also for larger language models. Lossless (e.g. trie storage) and lossy (e.g. Bloom filters – a hash function based approach that was sometimes wrong but with low probability) compression techniques were all the rage until about 2011, but we won't discuss them here so we can instead move on to neural language models, which made these approaches unnecessary.

3 Intrinsic evaluation: Perplexity

There's no way to conceive of held-out 'labeled' data in language modeling. So how are we to judge the quality of a language model? We hold out some portion of natural language and after building the model ask it what it thinks of the held-out portion (i.e. how probabilistic it is). Since the held-out portion is a sample of real language, the model should give it a high probability. Naturally, we wouldn't expect the probability to be 1, as if the model is a true language model, it should distributed probability mass across all (generally infinite) sentences of the language.

It won't do, though, to have models report probabilities on one particular piece of text (as there will be overfitting) nor can we compare across different pieces of text, as they have different sizes. We instead want to report *per-word* behavior, since although we know words don't have equal 'amounts' of language (whatever that means) they come as close as anything else.[2] Rather than simply describe the probability per word we use an information-theoretical approach. Consider *cross-entropy*, which is used to train logistic regression and neural models (particularly ones with categorical output, which an LM is an instance of). The cross entropy being calculated is:

[2]There is work that measures per-character or even per-byte perplexity but it is less common.

$$H(\tilde{p}, q) = -\sum_{x \in \mathcal{X}} \tilde{p}(x) \log q(x)$$

where x is a possible member of language X (i.e. a sentence), $\tilde{p}$ is the 'true' distribution of language, and q is our model's distribution of language. H, the cross-entropy, measures the average number of bits (assuming a log base of 2) it takes to properly calculate the memebrs of X using the suboptimal model q instead of $\tilde{p}$. Think of this as a 'codebook' with instructions on how to turn faulty distribution q into the true distribution; H is the size of the entry of each item in the book. We don't know the true distribution of $\tilde{p}$, but we have a sample M of it, so we assume every $x \in M$ has probability 1, and everything else has probability 0. So we can rewrite as:

$$H(\tilde{p}, q) = -\sum_{x \in \mathcal{M}} \log q(x)$$

Furthermore, we generally predict language one word at a time and we want a metric for the average 'goodness' of our model per word. So we re-cast M as a sequence of words $x_1, x_2, \ldots, x_{|M|}$ and write the average cross-entropy as:

$$H_{\text{ave}}(\tilde{p}, q) = -\frac{1}{|M|} \sum_{x_i \in \mathcal{M}} \log q(x_i | x_{i-1}, \ldots, x_1)$$

Rather than report the average number of bits needed to represent the truth using q instead of $\tilde{p}$ we instead cast this as the 'degree of confusion.' If 5 bits were needed, then a codebook with words that represent up to $2^5 = 32$ choices are needed, and so we are 32-way 'confused' or 'perplexed' when we use q. We thus calculate the *per-word perplexity* as

$$2^{H_{\text{ave}}(\tilde{p}, q)}$$

However, this assumes cross-entropy was calculated with a base 2 log, and in general that's not what we do; we prefer to use a base of e. So perplexity is in fact usually calculated in terms of 'nats', i.e .

$$\exp(H_{\text{ave}}(\tilde{p}, q))$$

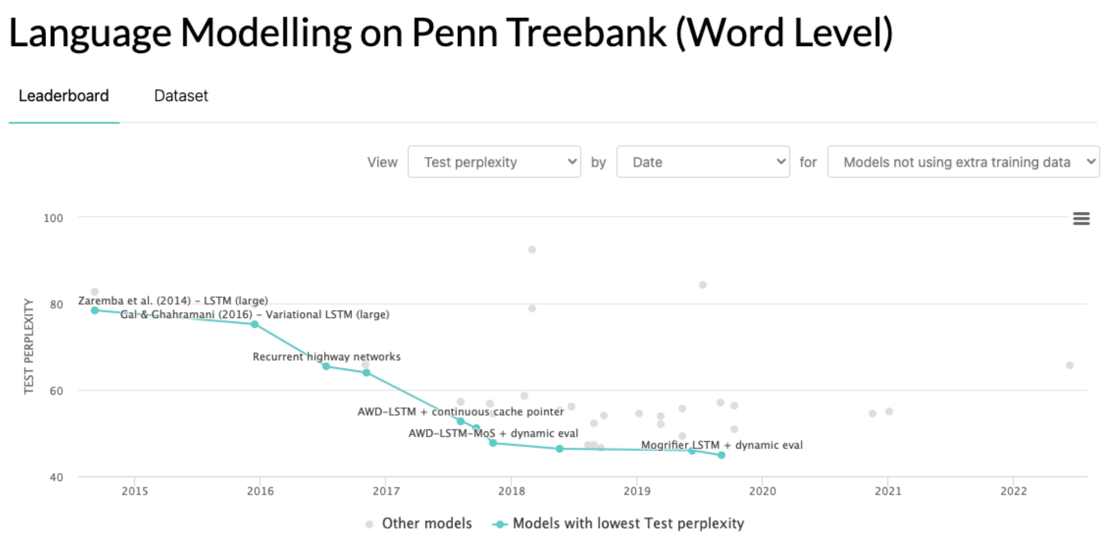
A really bad language model that predicts each word in V uniformly would get perplexity of $|V|$. A really good one would have perplexity of 1.

Let's try a Shannon game with characters (words are too hard).

Despite using per-word averages it is usually a good idea to compare across common benchmarks.

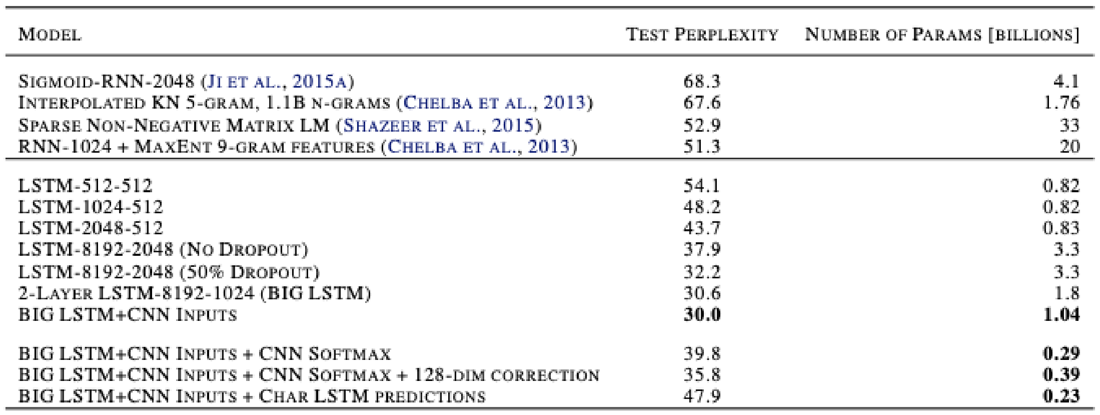
The 1m-word Penn treebank with a vocabulary limited to 10,000 types gets 141 ppl using a good 5-gram model, and under 60 using neural models; here is the latest:[3]

[3]`https://paperswithcode.com/sota/language-modelling-on-penn-treebank-word`, (retrieved 9/8/22)



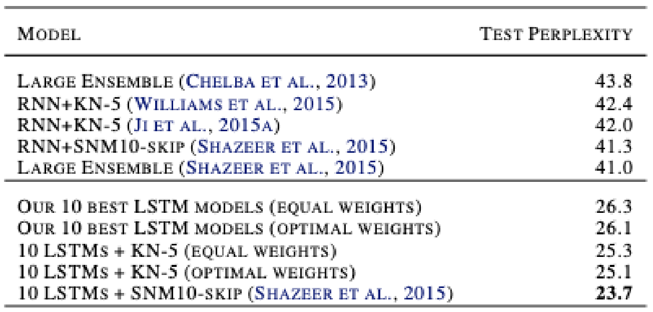
On a larger wikipedia-based 1b-word corpus (the billion word benchmark), state of the art as of 2016 is around 25.[4]

Table 1. Best results of single models on the 1B word benchmark. Our results are shown below previous work.



MODEL	TEST PERPLEXITY	NUMBER OF PARAMS [BILLIONS]
SIGMOID-RNN-2048 (JI ET AL., 2015A)	68.3	4.1
INTERPOLATED KN 5-GRAM, 1.1B N-GRAMS (CHELBA ET AL., 2013)	67.6	1.76
SPARSE NON-NEGATIVE MATRIX LM (SHAZEER ET AL., 2015)	52.9	33
RNN-1024 + MAXENT 9-GRAM FEATURES (CHELBA ET AL., 2013)	51.3	20
LSTM-512-512	54.1	0.82
LSTM-1024-512	48.2	0.82
LSTM-2048-512	43.7	0.83
LSTM-8192-2048 (NO DROPOUT)	37.9	3.3
LSTM-8192-2048 (50% DROPOUT)	32.2	3.3
2-LAYER LSTM-8192-1024 (BIG LSTM)	30.6	1.8
BIG LSTM+CNN INPUTS	**30.0**	**1.04**
BIG LSTM+CNN INPUTS + CNN SOFTMAX	39.8	**0.29**
BIG LSTM+CNN INPUTS + CNN SOFTMAX + 128-DIM CORRECTION	35.8	**0.39**
BIG LSTM+CNN INPUTS + CHAR LSTM PREDICTIONS	47.9	**0.23**

Table 2. Best results of ensembles on the 1B Word Benchmark.

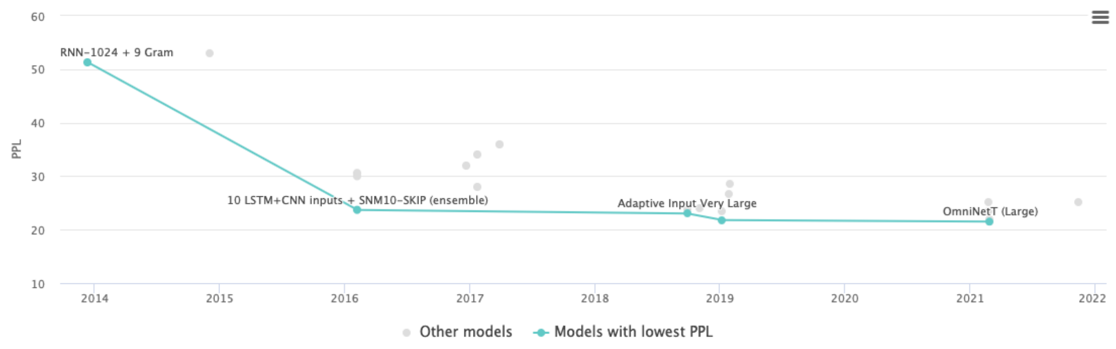


MODEL	TEST PERPLEXITY
LARGE ENSEMBLE (CHELBA ET AL., 2013)	43.8
RNN+KN-5 (WILLIAMS ET AL., 2015)	42.4
RNN+KN-5 (JI ET AL., 2015A)	42.0
RNN+SNM10-SKIP (SHAZEER ET AL., 2015)	41.3
LARGE ENSEMBLE (SHAZEER ET AL., 2015)	41.0
OUR 10 BEST LSTM MODELS (EQUAL WEIGHTS)	26.3
OUR 10 BEST LSTM MODELS (OPTIMAL WEIGHTS)	26.1
10 LSTMS + KN-5 (EQUAL WEIGHTS)	25.3
10 LSTMS + KN-5 (OPTIMAL WEIGHTS)	25.1
10 LSTMS + SNM10-SKIP (SHAZEER ET AL., 2015)	**23.7**

Progress since then[5]:

[4] `https://arxiv.org/pdf/1602.02410.pdf`

[5] `https://paperswithcode.com/sota/language-modelling-on-one-billion-word`, (retrieved 9/8/22)



So let's talk next about the impact of neural networks on language modeling!