1. Introduction

Consider a corpus $\mathcal{C}$ of tokens from a language $\mathcal{T} = \{1, \dots, T\}$ (w.l.o.g.) that at the very least "covers" $\mathcal{T}$ in the sense that every $t \in \mathcal{T}$ appears in $\mathcal{C}$. By a corpus we basically mean, here, a giant body of text or a "long" sequence of elements of $\mathcal{T}$. Can we compute the empirical distribution of next tokens for some batch size B? Let's define what we mean clearly: we want to estimate the conditional probabilities

$$\rho_B(\mathbf{t}, \tau) = \mathbb{P}(\ t_{B+1} = \tau \mid t_1, \dots, t_B\ )$$

with "(Conditional) Empiricial Frequencies" or (C)EFs. This is, of course, a purely Markov representation of token sequences.

Before detailing how, why would we do such a thing? Markov modeling of language aged out forever ago. Yet a plausible "hypothesis" is that these probabilities are what LLMs like GPT are fundamentally modeling. Any functional representation, such as a simple variant of multiclass logit regression or the deep networks in modern LLMs, impose structure on sequential predictions. Such structure exists to "smooth" or "fill" gaps in knowledge of some "true" conditional probabilities for "small" data volumes (relatively). As the training corpus size grows a model from a suitably "universal" class token sequence predictor estimated consistently will increasingly tend to the (C)EFs and effectively "regurgitate" patterns from its training data.[1] We mostly ignore the mathematics of the underlying statistical assumptions here (consistency, unbiasedness).

This idea – that the best a LLM could do is defined by the (C)EFs – has implications for conceptualizing what LLMs are. First, should this hypothesis be correct, it is a formal, almost philosophical illustration of how it is impossible for LLMs to be generatively "innovative". In a strict sense they can only repeat the patterns of the past, as described by the (C)EFs in the training data. Which is not to say they are not still *useful*, by any means; after all, who is to say we ourselves don't just resample the past. We should still be able to reflect on the structure and limitations of such tools, as well as ourselves. Second, if model structure aims at universality, and there is enough data to faithfully represent the (C)EFs, then the structure has to *compress* the (C)EFs. In slightly plainer words: as we increase the volume of data used to train a LLM from a reasonably "universal" class, either

[1] An empirical distribution (from iid samples) converges w.p.1 to the true distribution from which samples are drawn. We can't formally speak of "consistency" of a NN-LLM as its model and parameters are fictitious, but we can safely suppose the point is to generate a distribution with a similar target, perhaps meaning an empirical distribution of samples from the LLM has the same convergence property.

(a) estimating the LLM parameters must be vastly simpler than computing the (C)EFs directly or (b) sampling sequences from the LLM must be vastly simpler than from the (C)EFs (or both). Something like GPT-3 is hardly "parsimonious" being based on over 175 *billion* parameters requiring O(1TB) to describe, with yet larger models also publicized, and generative sampling requiring complex distributed computing on specialized hardware. This is not to say we actually *have* enough data to fully describe all sayable things (far from it, probably), just to point out an inevitability of the current trend towards more and more data.

At least to me, it is not entirely clear that (C)EFs are "efficiently" computable (in, so to speak, "training"), or sampling from them is efficient, or, most importantly, if samples from (C)EFs appear to mimic "real" language structure like LLMs do (as defined in the training corpus). Here we'll revisit history and focus on those questions. What datastructure models (C)EFs? How would we (efficiently, scalably) compute them from a (large) corpus? How would we generate sequences? Are those sequences at all representative of text in the training corpus?

2. Modeling

Again we want to estimate a Markov model of token sequences via the (C)EFs

$$\rho_B(\mathbf{t},\tau) = \mathbb{P}(\ t_{B+1} = \tau \mid t_1,\dots,t_B\ )$$

This is probably to be done via something like

$$\hat{\rho}_B(\mathbf{t},\tau) = \frac{\text{\# of occurrences of } \mathbf{t}\circ\tau \text{ in } \mathcal{C}}{\text{\# of occurrences of } \mathbf{t} \text{ in } \mathcal{C}}$$

where $\circ$ is concatenation. Why *something like*? The question would be whether *exactly* this empirical value would be *generative* without an underlying structural model. Simply, suppose we have some $\mathbf{t}$, we generate a next τ, but such that $(t_2,\dots,t_B,\tau)$ is not in $\mathcal{C}$. Strictly speaking, we can't then use literal B-prefix conditional empirical distributions as they would not tell us what comes next. However we'll stick to the naivest possible calculation, and outline a well-defined sampling process below.

Clearly $\rho_{B-1}(\mathbf{t},\tau)$ is a *marginal* relative to $\rho_B(\mathbf{t},\tau)$, because ρ_B is more specific than ρ_{B-1}. That is,

$$\rho_{B-1}(\mathbf{t},\tau) = \mathbb{P}(\ t_B = \tau \mid t_1,\dots,t_{B-1}\ ) = \sum_t \rho_0(t)\rho_B\big(t\circ\mathbf{t},\tau\big)$$

where $\rho_0(t) = \mathbb{P}(T = t)$. Also

$$\hat{\rho}_0(\tau) = \frac{\text{\# of occurrences of } \tau \text{ in } \mathcal{C}}{|\mathcal{C}|}$$

which we are assured is defined and positive (because the corpus covers the token language).

So how would we compute $\hat{\rho}_B$? To sketch a start, we can do this in a single pass over $\mathcal{C}$ as follows:

(0) Initialize a hashmap $\mathcal{P}_B$ whose keys are B-token strings and values are also hashmaps with single-token keys and whose values are floats (technically `doubles`). We will store $\hat{\rho}_B(\mathbf{t}, \tau) = \mathcal{P}_B[\mathbf{t}][\tau]$. Initialize $c = B$.
(1) Set $\mathbf{t} = \mathcal{C}[c - B : c)$, $\tau = \mathcal{C}[c]$
(2) If $\mathcal{P}_B[\mathbf{t}]$ does not exist, initialize $\mathcal{P}_B[\mathbf{t}]$ as needed, and set $\mathcal{P}_B[\mathbf{t}][\tau] = 1$. Otherwise, if $\mathcal{P}_B[\mathbf{t}][\tau]$ does not exist, set $\mathcal{P}_B[\mathbf{t}][\tau] = 1$. Otherwise increment $\mathcal{P}_B[\mathbf{t}][\tau]$.
(3) Increment c and go back to (1) unless $c = |\mathcal{C}|$, in which case continue to (4).
(4) For all keys $\mathbf{t}$ defined in $\mathcal{P}_B$, compute

$$S(\mathbf{t}) = \sum_{\tau \in \mathcal{P}_B[\mathbf{t}]} \mathcal{P}_B[\mathbf{t}][\tau]$$

and update

$$\mathcal{P}_B[\mathbf{t}][\tau] \leftarrow \mathcal{P}_B[\mathbf{t}][\tau] / S(\mathbf{t})$$

Technically this is a bit like a double-pass algorithm considering the normalization in (4), but it is still $O(|\mathcal{C}|)$. We also might want to reverse the ordering of tokens in the keys of $\mathcal{P}_B$, to enable easier search with some tools that can do bulk return with partial key matching, but that's a detail. This is also embarassingly parallelizable, distributing the right overlapping subsets of $\mathcal{C}$ and merging globally, though we don't outline the details.

Now, we also need to reduce to $P_{B-1}, P_{B-2}, \dots, P_0$ for any hope of prediction. Specifically, we could predict a token τ for which the concatenated subsequence $(\mathbf{t}[2:]) \circ \tau$ does not occur in the corpus. By assumption P_0 exists and is "complete", as the empirical frequencies of tokens in the corpus are defined by virtue of covering. That is, we can always simply sample from the simple occurrence likelihood. Our process could be to take find longest suffix of $\mathbf{t}$ that exists in the corpus,

$$\mathbf{t}[:-k] \quad \text{where} \quad k = \arg\min_{0 \leq k \leq |\mathbf{t}|} \{ |\mathbf{t}[:-k]| : \mathbf{t}[:-k] \subset \mathcal{C} \}$$

and sample from $P_{|\mathbf{t}[:-k]|}[\mathbf{t}[:-k]]$. In the "worst case" $k = |\mathbf{t}|$ and we choose from P_0.

Prediction actually means a $\mathcal{T}$-set of suffix trees may be more suitable, where we access $P_B[\mathbf{t}]$ via following the "reverse" or suffix path

$$t_{|\mathbf{t}|} \to t_{|\mathbf{t}|-1} \circ t_{|\mathbf{t}|} \to \cdots$$

from a (guaranteed-to-exist) root $t_{|\mathbf{t}|}$ to the deepest accessible node. This could also aid in the "marginalization" process. Each node would contain a hashmap representing the distribution over next most likely tokens.

Presuming we have the (C)EFs, how can we "validate"? We quote "validate" presuming that the (C)EFs are all there is, so when new sequences are observed they are more like new information than test samples to be predicted. (This data-centric view is wrong, of course. There is higher level structure in language, whose rendering in a model is likely hard to capture without rule-based models especially with naive character-based tokenizations. More advanced "language-aware" tokenizers might fare better.) In any case, we could view "validation" as a joint distribution prediction task: Specifically, given a novel sequence $\mathbf{t} \circ \tau$, what is the probability of predicting this sequence? If $\tau \in \mathcal{P}_B[\mathbf{t}]$, then the probability is estimated by $\mathcal{P}_B[\mathbf{t}][\tau]$; otherwise if $\tau \in \mathcal{P}_{B-1}[\mathbf{t}]$ it is $\tau \in \mathcal{P}_{B-1}[\mathbf{t}][\tau]$; and so on down to

$\mathcal{P}_0[\tau]$. The closer this is to 1, the "better" the (C)EF model is. But of course, "better" is a strange adjective here, as it means something more like how *un*-novel the new sequence actually is.

3. A Simple Example

The code ‘efnlp‘ takes a perhaps shockingly naive approach to modeling (C)EFs. We use ‘dict‘-based suffix trees to parse a (character) tokenized corpus of text, and generate text with a best-matching-marginal sampling approach as discussed above.

Some researchers use the full corpus of Shakespeare's writing to quickly analyze language models. Here is the start of sample of 10k token long generated Shakespeare using (C)EFs with character tokens and 10-token sequences:

> Having the fearful'st time to chide.
>
> Nurse:
> Mistress, how mean you that? no mates for you,
> Unless you have lately told us;
> The Volsces
> May say 'This mercy we have spent our harvest of his coffers shall be joyful
> of thy company.
>
> ARCHBISHOP OF CARLISLE:
> Marry. God forbid! Where's Abhorson, there?
>
> ABHORSON:
> What, household Kates.
> Here comes a man

This is pitiful Shakespeare but not at all gibberish. For the most part the words are words, line starts are capitalized, there is punctuation, there are "character" headings before statement blocks suitable to a play, and "Abhorson" is even called for in one line and responds next. Honestly I find this output intriguingly realistic-ish for computing using only counts and ratios.

Here's the run command for that sample and (modified) outut:

Listing 1. Analyzing and generating some Shakespeare with (C)EFs

```
$ python -m efnlp -c data/tinywillspeare.txt \
        -m -b 10 -g 100000 -o sample-results.txt
[...:31:07.445610] Forming (character) language
[...:31:07.491561] Encoding corpus
[...:31:07.569177] Corpus is 1,115,393 tokens long
[...:31:07.569217] Parsing prefix/follower tokens
```

```
[...:31:41.985965] Normalizing to empirical frequencies
[...:31:51.631065] Memory (roughly) required: 62.4 MB
        (about 8,183,314 dbl, 16,366,628 fl)
[...:31:51.631112] Sampling and decoding 100000 tokens
[...:31:52.810630] Writing sampled results to \
        sample-results.txt
```

We shorten an ISO timestamp in the "logs" here, and first number in the log (`31`) is the minutes place. Note the entire exercise – encoding, estimating, and generating – completes in under a minute. Sampling 10k tokens takes about a second (1.2s, for about 0.1ms per token sampled). We more or less store $6x$ the corpus volume in (C)EF data, the equivalent of 8M texttttdoubles (16M `floats`). This is all on a 4-year-old 2.3 GHz i9 with 16GB 2.7 GHz memory without any attention to code optimization (e.g., our datastructure is likely wasteful), in a single process, and with a bunch of other stuff (including chrome) running.

One take at a NN-LLM using transformers modeling this data claims to do a decent job with 10M parameters (so more memory if `doubles`, less if `floats`) after estimating parameters on a GPU (and SOTA software) for 15mins. Sampling time for generation is not available, nor is a sample, for comparison; though tree based lookups and simple random sampling is likely to be dramatically more efficient than layers of linear and nonlinear operations with 10M parameters.

4. Scaling

In an N-token long corpus $\mathcal{C}$ there are $N-B-1$ $(B+1)$-long subsequences of prefixes and (single token) successors. A simple estimate is that this requires, at most, storing $(N-B-1)(B+1+d)$ data elements where d counts data required for empirical frequencies (a counter, and/or a probability). Note this is not linear in $B \leq N-1$, and is maximized at $B_* = (N-2-d)/2$ with a maximal value of

$$\frac{N^2-d^2}{4} - (N+d) \approx \frac{N^2}{4} - N = N\left(\frac{N}{4}-1\right)$$

which is, in any case, $O(N^2)$. Presumably we won't need nearly this much storage though; for $B = \eta N$, expecting η to be small (say 10^{-5} for the Shakespeare), we need something akin to $O(\eta N^2)$. In any case this is an *overestimate* by missing two things: that only certain prefixes will exist in the corpus (or more generally the language), and for those there will be a probably-much-smaller-than T set of successors. This *underestimates* any storage we require for capturing the marginals for matching based on less than B tokens in a prefix, although in principle those can computed on demand instead of cached (with slower sampling of course).

A more "realistic" (minimum) storage scaling concept is $(N-B-1)(B+1+d)r(B)s(B)$ where

$$r(B) = \frac{\text{\# of prefixes}}{N-B-1} \quad \text{and} \quad s(B) = \frac{\text{\# of patterns}}{\text{\# of prefixes}}$$

(leaving N implicit). Broadly speaking we (probably) know some features of these "functions" r and s. Specifically, $r(1) = L/N$ ($\ll 1$), $r(B) \leq 1$ (by definition), and $r(B) \uparrow 1$; in fact $r(N-2) = 1$, because # of prefixes → 1. Also $s(B) \geq 1$ (every prefix has at least one pattern) and probably $s(B) \downarrow 1$ (as longer prefixes are considered, their successors are more likely to be unique). Note that $r(B)$ is not the fraction of *possible* prefixes, which would instead be (# of prefixes/T^B) but we should probably expect that in "natural" language the number of *valid* prefixes is far, far smaller than T^B (while admitting it may be much larger than $N - B - 1$). Loosely speaking, if T^B is at all relevant, the language would seem to be highly unstructured.

As an example, for the $1,115,393$ tokens of all of Shakespeare we have the following parsing details:

B	# prefixes	$r(B)$	# patterns	$s(B)$	τ/prefix	memory
1	65	0.0%	1,403	0.1%	21.6	3kB
2	1,403	0.1%	11,556	1.0%	8.2	36kB
3	11,556	1.0%	50,712	4.5%	4.4	221kB
4	50,712	4.5%	141,021	12.6%	2.8	876kB
5	141,021	12.6%	283,313	25.4%	2.0	2.5MB
7	447,352	40.1%	609,659	54.7%	1.4	10.1MB
10	858,920	77.0%	937,254	84.0%	1.1	31.9MB
12	991,391	88.9%	1,027,857	92.2%	1.0	50.4MB
15	1,069,423	95.9%	1,081,060	96.9%	1.0	80.6MB
20	1,103,358	98.9%	1,106,345	99.2%	1.0	133MB

With 1-token prefixes (bigrams) the prefixes are the language (as required by covering) and, while there are quite a few successors per "prefix" (~ 22) stored in at least $49kB$, we definitely have far fewer successors than "random" (22 $T/3$). Still the generative output from such a sparse model is expectedly and unequivocally junk. The number of prefixes found increases but with diminishing returns; by 10- or 12-token prefixes we already have "mostly" unique prefixes (77% and 88.9% respectively), and 20-token prefixes are for all intents and purposes unique. (To be clear, by "unique" here we mean non-recurrent in the corpus.) The number of successors per prefix decreases (of course): with 5-token prefixes, we already have only 2 successors per prefix (on average) and by 10-tokens successors are on the whole unique. As hinted at above this is a limit on how long our prefixes should probably be: by design every prefix will have *at least one successor*, so if we have (even on average) a *single* successor per prefix we have probably captured all there is to capture from prefix sequences.

Parsing and generation (sampling) times are also important, and listed in the table below for implementations in three languages. Generation of 1M samples was used to estimate the "time/tok" frequency (time per token)

	`python`		`go`		`c++` -O3	
B	parse	time/tok	parse	time/tok	parse	time/tok
1	1.0s	1.4μs	49ms	0.1μs	131ms	0.1μs
2	2.0s	1.7μs	109ms	0.1μs	248ms	0.2μs
3	3.3s	2.1μs	222ms	0.2μs	419ms	0.3μs
4	4.3s	2.6μs	361ms	0.3μs	612ms	0.4μs
5	6.4s	3.2μs	585ms	0.5μs	1.1s	0.5μs
7	12.0s	4.9μs	1.2s	0.7μs	2.0s	0.7μs
10	28.0s	7.0μs	2.6s	0.9μs	1.9s	0.8μs
12	37.3s	8.3μs	4.1s	1.1μs	2.5s	1.0μs
15	54.3s	9.7μs	5.2s	1.2μs	3.2s	1.0μs
20	129.0s	12.7μs	8.4s	1.6μs	4.4s	1.3μs

Unsurprisingly, compiled `go` and `c++` implementations are faster. Parsing times are a fraction than in the `python` code. Sampling can be done in O(μs) per token in each implementation, but still roughly an order of magnitude faster than `python` with `go` or `c++`.

5. Languages

There are surely deep fields with associated terminology related to NLP we do not know. Here, a `language` is an encoding strategy for turning text into tokens; it might also be appropriately called a `tokenizer`, though that conflicts a bit with a specific instantiation of an algorithm for doing so. Text is drawn from a character alphabet, probably unicode (e.g. `utf-8`), and associated to tokens $\{1, \dots, N\}$ through a `language`. Formally, a `language` $\mathcal{L}$ is a surjective mapping

$$\mathcal{L} : \mathcal{S}(\{1, \dots, N\}) \to \mathcal{S}(\mathcal{A})$$

between sequence sets of the token space $\{1, dotsc, N\}$ and the text alphabet $\mathcal{A}$. (Note in software the tokens will start at 0 not 1 but this is an artificial distinction.) Two token sequences may map to the same text, but no text is *not* mapped to at least one token sequence. The `language`s we consider are based on "sequence encoders": *ordered* subsets $\mathcal{E}_< \subset \mathcal{S}(\mathcal{A})$ identified to tokens by their index in the ordering. Obviously any such sequence encoder is an equivalence class over permutations in the ordering.

Consider for example the alphabet $\mathcal{A} = \{a, b\}$, and the trivial encoder $\mathcal{E}_< = (\{a\}, \{b\})$ with $a < b$. In this case sequences of the alphabet are identified (uniquely) to binary strings (taking the token space to be $\{0, 1\}$). Our `language` $\mathcal{L}$ is the substitution mapping

$$(\mathcal{L}(t_n))_n = (1 - t_n)a + t_n b$$

admitting some severely abusing notation. Consider instead $\mathcal{E}_< = (\{aa\}, \{a\}, \{b\})$ with $aa < a < b$. Now the mapping is no longer unique, for there are two ways to "render" aa: $\mathcal{L}(0) = aa$ and $\mathcal{L}(1, 1) = aa$.

From the perspective of "decoding" token sequences non-uniqueness is not an issue. For encoding, however, it raises questions about how to choose between associations. Encoding here means affecting the inverse map $\mathcal{L}^{-1}$ to turn text into tokens. It is reasonable to suppose we have a preference for *compression*: when encoding choose the *shortest* token sequence that represents the text.

Greedily encoding text with the longest match using either prefix or suffix search (alone) can fail. Consider $\mathcal{E}_< = (\{aa\}, \{ab\}, \{a\})$ with $aa < ab < a$. A greedy forward prefix search of aab encodes $aa \to 0$ but is then left with b which is un-encodable; the only encoding is, of course, $(2, 1)$. Consider instead $\mathcal{E}_< = (\{aa\}, \{ba\}, \{a\})$ with $aa < ba < a$. A greedy backwards suffix search of baa again encodes $aa \to 0$ and is left with b which is un-encodable. The encoding should be $(1, 2)$ These examples have two characteristics: First, the encodings are *not* "character-complete" by which we mean

$$(c_1, \dots, c_N) \in \mathcal{E} \quad \implies \quad c_n \in \mathcal{E} \text{ for all } n$$

(every character is encodable on its own). Specifically, the failures in the examples rely on us not being able to encode b alone. Second, greedy suffix search would overcome the greedy prefix failure, and greedy prefix search would overcome the greedy suffix failure. We may return to this later as a heuristic encoding strategy, but it's a bit superfluous.

The first observation is more fundamental: If an encoding is character-complete then any sequence is greedily encodable (at all) with either greedy prefix or suffix search. We may not have the *shortest* encoding, but at least we know the algorithm cannot fail to encode. Notably, OpenAI's GPT-2 encoder referenced in their `tiktoken` codebase is character-complete in the sense defined above, and hence greedy algorithms could be applicable.

Why would we burden ourselves with multi-character encodables at all?

First, it is a reasonable thing to expect from the grammar of languages. Consider the bane of many an english student: "i before e except after c". Taken too literally, this means $[abde \cdots z]ei$ (e.g. "$(sei)ze$") should *not occur* but cei ("$(cei)ling$") should. Of course, the rule is "wrong", as "$(hei)ght$" or "$spe(cie)s$" shows. In any case we might define an encoder with cei and Xie for all $X \neq c$ to "represent" rules around the digraphs ie and ei. We might similarly wish to encode other common character concatenations, e.g. th, gh, ion, ing etc, purely out of a desire to better represent character occurrences in text.

At this point it is worth noting that any such activity is a layer of language modeling in itself. Consider, for example, a tokenization based on all english words (plus a space, maybe pluralization, and some punctuation). A (C)EF model of token sequences for such an encoding would be a model of contraction-less sentence construction. No characters could be garbled, but nonsensical sayings could certainly emerge. Moreover, it should be apparent that our suffix tree modeling approach to (C)EFs will, in effect, *find* what co-character occurrences are common; in fact, this is almost a complete algorithm to *find* encodable multi-character sequences (more or less, build a character based tree and "merge" edges that are unique or have high probability).

Second, tokenizing longer character sequences is may be computationally efficient. The more "language" information that is packed into tokens, the more "text" we can model with a shorter token sequence. From the (C)EF perspective we can effectively have a longer Markov chain rendering with the same tree depth. The "cost" (if any) will come from a larger token space. Shakespeare can be constructed from only 65 of the 128 ASCII characters; english Wikipedia (according to a 2020 Kaggle dataset) has only 168 UTF-8 characters in it's top 0.0005% of character occurrences.

5.1. **Is Greedy Search Fast or Smart?** `tiktoken` tokenization of shakespeare with the GPT-2 encoder takes about 334ms. This library is not pure python, utilizing compiled `rust` code for basic operations to enhance speed. Greedy tokenization of shakespeare in pure `python` with GPT-2 encoder takes about 114s, but only about 50ms in `go`.

Greedy suffix tokenization compresses the 1,115,393 characters in shakespeare by about half (48%) using a token sequence with 541,060 tokens. `tiktoken` encodes a much smaller token string with 338,024 tokens (30% of original, 62% of greedy). The following table summarizes the distribution, via counts (histogram) of tokens that have the same decoded string lengths, using these two tokenization strategies:

	decoded string length (# UTF-8 characters)										
method	1	2	3	4	5	6	7	8	9	10+	2+
`tiktoken`	100,222	39,141	54,712	49,361	45,883	21,685	12,194	7,929	3,782	3,115	237,802
greedy suffix	296,177	72,256	76,582	57,743	23,711	8,929	4,064	1,144	334	120	244,883

The greedy suffix search is biased towards short strings, with more than half (296k vs 244k) of the encoded tokens representing a single character. In contrast, `tiktoken` renders Shakespeare with half as many tokens representing single instead of multiple characters (100k vs 237k). (Note that greedy suffix search has more multi-character tokens, but that's only because it has more tokens overall.)

To be clear, these comparisons are about *compression* and contain no information about *effectiveness*, at least on their own.

Mountain View CA
Email address: `morrowwr@gmail.com`