Probabilistic Language Models 1.0

Noah A. Smith

March 2017

Abstract

We introduce and explain language models. These notes are inspired in part by notes by Collins [2011a], who provides more detail about some topics. See also Jurafsky and Martin [2016] for a textbook treatment.

What makes a good English sentence? A human might tell you that a good sentence is grammatical, meaningful, and "fits" into the context in which it is uttered or written. In NLP, all of these considerations eventually come to play a role, but we begin with a much simpler observation: the vast majority of sequences of words are not good sentences, but almost anything is possible. Probabilistic language modeling—assigning probabilities to pieces of language—is a flexible framework for capturing a notion of *plausibility* that allows anything to happen but still tries to minimize surprise.

1 The Problem

Formally, the language modeling problem is as follows. Let $\mathcal{V}$ be the vocabulary: a (for now, finite) set of discrete symbols. Let ◯, called the "stop" symbol, be one element of $\mathcal{V}$. Let V denote the size of $\mathcal{V}$ (also written as $|\mathcal{V}|$. Let $\mathcal{V}^\dagger$ be the (infinite) set of *sequences* of symbols from $\mathcal{V}$ whose final symbol is ◯.

A language model is a probability distribution for random variable $\boldsymbol{X}$, which takes values in $\mathcal{V}^\dagger$ (i.e., sequences in the vocabulary that end in ◯). Therefore, a language model defines $p : \mathcal{V}^\dagger \rightarrow \mathbb{R}$ such that:

$$\forall \boldsymbol{x} \in \mathcal{V}^\dagger, p(\boldsymbol{x}) \geq 0, \text{and} \tag{1}$$

$$\sum_{\boldsymbol{x} \in \mathcal{V}^\dagger} p(\boldsymbol{X} = \boldsymbol{x}) = 1. \tag{2}$$

The steps to building a language model include:

1. Selecting the vocabulary $\mathcal{V}$.
2. Defining the parameters of the probability distribution p.
3. Estimating those parameters from a training dataset of sentences $\boldsymbol{x}_{1:n} = \langle \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n \rangle$, where each $\boldsymbol{x}_i$ is a sequence in $\mathcal{V}^\dagger$.

We next explain why we want to build language models (Section 2), then discuss how we evaluate them (Section 3). We discuss each step in turn (Sections 4–5). We then discuss algorithms relating to language models (Section 6). Finally, we present log-linear language models, which reparameterize the probability distribution using features. (Section 7).

2 Why Language Modeling?

One motivation for language modeling comes from the **noisy channel** paradigm, a metaphor and modeling pattern that has inspired considerable research in NLP.

Suppose we have two random variables, $\boldsymbol{O}$ (which is the observed input to our system) and $\boldsymbol{D}$ (which is the desired output of our system). We might view $\boldsymbol{O}$'s value as a ciphertext and $\boldsymbol{D}$ as a plaintext into which we would like to **decode** the value of O. To do this, we must solve:

$$\boldsymbol{d}^* = \operatorname*{argmax}_{\boldsymbol{d}} p(\boldsymbol{d} \mid \boldsymbol{o}) \tag{3}$$

$$= \operatorname*{argmax}_{\boldsymbol{d}} \frac{p(\boldsymbol{o} \mid \boldsymbol{d}) \cdot p(\boldsymbol{d})}{p(\boldsymbol{o})} \tag{4}$$

$$= \operatorname*{argmax}_{\boldsymbol{d}} \underbrace{p(\boldsymbol{o} \mid \boldsymbol{d})}_{\text{channel model}} \cdot \underbrace{p(\boldsymbol{d})}_{\text{source model}} \tag{5}$$

The last line of the above formula shows how the noisy channel pattern decomposes the model over $\boldsymbol{O}$ and $\boldsymbol{D}$. A good plaintext is likely *a priori* (i.e., under the source model) and also likely to have generated the observed ciphertext (i.e., through the channel model). If either of these is too small for a value $\boldsymbol{d}$, then $\boldsymbol{d}$ is not a good decoding.

The noisy channel model corresponds to a "story" about how data is generated. The story is visualized as follows:

$$\boxed{\text{source}} \longrightarrow \boldsymbol{D} \longrightarrow \boxed{\text{channel}} \longrightarrow \boldsymbol{O}$$

First, the source distribution randomly generates a value for the plaintext random variable $\boldsymbol{D}$. Second, this plaintext passes through a channel that corrupts it (or "adds noise to it"), resulting in a value for the observed random variable $\boldsymbol{O}$. Decoding works in the reverse direction, applying Bayes rule (Equation 4) to solve for the most likely value of $\boldsymbol{D}$ given the observation that $\boldsymbol{O} = \boldsymbol{o}$.

Classic examples of the noisy channel model include speech recognition and machine translation; in both cases, the source model is a language model over word sequences in the output language; $\boldsymbol{D}$ ranges over $\mathcal{V}^\dagger$. In speech recognition, the *acoustic* model gives a distribution over sounds given words. In machine translation, the *translation* model gives a distribution over input-language sentences given output-language sentences. Both of these are channel models; note that they counterintuitively model a process of transforming an *output* into an *input*.

The other motivation for studying language models first in a natural language processing course is that the techniques we use in language modeling provide a kind of simplest first case for methods that are used repeatedly later on.

3 Evaluating Language Models with Perplexity

A high-quality language model will assign high probability to real sentences it has never observed (i.e., that were not part of the data used to *train* the model). The standard way to measure the quality of a language model is to consider a dataset that is separate from the training data, called a **test dataset**, $\bar{\boldsymbol{x}}_{1:m}$, and calculate the model's perplexity on this dataset:

$$\text{perplexity}(p; \bar{\boldsymbol{x}}_{1:m}) = 2^{-\left(\frac{1}{M} \log_2 \prod_{i=1}^{m} p(\bar{\boldsymbol{x}}_i)\right)} = 2^{\left(\frac{1}{M} \sum_{i=1}^{m} -\log_2 p(\bar{\boldsymbol{x}}_i)\right)} \tag{6}$$

where M is the number of words in $\bar{\boldsymbol{x}}_{1:m}$ (i.e., $\sum_{i=1}^{m} |\bar{\boldsymbol{x}}_i|$). Lower is better.

Do not be thrown by the "two to the power ..." at the beginning of this expression, or by the logarithm! Remember that exponentiation and logarithms are inverses; what's going on in this formula is kind of intuitive if we take it step by step. We'll consider the expression on the right of Equation 6.

1. First, calculate the probability of each element of the test data $\bar{\boldsymbol{x}}_i$, for $i \in \{1, \ldots, m\}$.
2. Taking the (base 2) logarithm of each of those values gives us a nonpositive number that could have arbitrarily large magnitude. Big negative values go with tiny probabilities; values close to zero go with larger probabilities. Recall that the logarithm function is *monotonic*; as a increases (decreases), $\log_2 a$ also increases (decreases).
3. Negating those logarithms gives us something known in machine learning as the **log loss**. Now, the values are nonnegative; large values mean low probabilities (bad), and values close to zero mean higher probabilities. This is sometimes described as a measure of surprise (or, in information theory, "information") that the model experiences when it observe a test sentence.
4. Take a per-word average of those sentence losses, by summing them up and dividing by the number of words. This converts to a per-word surprise score.
5. Raise 2 to this power. Again, this is a monotonic transformation (as a increases, so does 2^a, only much faster).

Perplexity can therefore be understood as a kind of branching factor: "in general," how many choices must the model make among the possible next words from $\mathcal{V}$? Consider a simpler case where we have only one test sentence, $\bar{\boldsymbol{x}}$. Perplexity is then $2^{-\left(\frac{1}{|\bar{\boldsymbol{x}}|}\log_2 p(\bar{\boldsymbol{x}})\right)}$. A few boundary cases are interesting to consider:

- If the model (miraculously) assigns probability of 1 to $\bar{\boldsymbol{x}}$, then perplexity will be 1, since $2^{-\frac{1}{M}\log_2 1} = 1$. Here, there is always one choice, and it's always correct! This never happens, because in reality anyone *could* say anything at any time.
- A model that assigns $p(\bar{\boldsymbol{x}}) = \left(\frac{1}{V}\right)^{|\bar{\boldsymbol{x}}|}$—essentially rolling an evenly weighted V-sided die for each word in $\bar{\boldsymbol{x}}$—will have perplexity V. Every word is uniformly possible at every step and therefore pretty surprising. It doesn't get more "random" than this, as there are always V choices.
- A model that assigns $p(\bar{\boldsymbol{x}}) = 0$ will have *infinite* perplexity, because $\log_2 0 = -\infty$.

Perplexity is not a perfect measure of the quality of a language model. It is sometimes the case that improvements to perplexity don't correspond to improvements in the quality of the output of the system that uses the language model. Some researchers see it as a "pure," system-agnostic way to test the quality of a model (or model family, or estimation procedure). When you see perplexity scores in the literature, take them with a grain of salt.

Two important caveats to remember about perplexity are (i) that you cannot directly compare two models' perplexity if they do not use identical vocabularies, and (ii) perplexity is not meaningful if the sum-to-one constraint (Equation 2) is not satisfied by your language model.

4 Vocabulary

Our starting point for defining $\mathcal{V}$, the vocabulary for our language model, is to include every word that occurs in the training dataset $\boldsymbol{x}_{1:n}$. It should be immediately obvious that a finite training dataset is not going to include all of the possible words in a natural language, because:

- New words show up all the time, either because they are borrowed from other languages or because they are invented ("neologisms").
- Many languages form words productively through their rules of *morphology*, so that the number of possible words results from a combinatorial explosion.

- Real-world data often shows variation in how words are written or spelled, either because there aren't standards or because people sometimes do not adhere to them.

A common solution is to introduce a special symbol into $\mathcal{V}$ called the "unknown word" denoted "UNK" for any out-of-vocabulary word. In the training dataset, some words—usually some or all instances of some or all of the rarest word types—are replaced by UNK.

Another solution, for morphologically rich languages, is to build a computational model that maps between the "surface forms" of words (as they are observed in text) and their "underlying" morpheme sequences. For example, in English, the word *starting* might map to ⟨*start*, *-ing*⟩, the word *vocabularies* might map to ⟨*vocabulary*, *-s*⟩, and the word *morphologically* might map to ⟨*morph*, *-ology*, *-ical*, *-ly*⟩. If every word in a text dataset can be decomposed into its morphemes, we could build a language model with $\mathcal{V}$ as the set of morphemes, rather than surface words. Morphological ambiguity, where a word can be broken down in multiple ways,[1]

A third solution is to build language models at the level of characters, or to combine character- and word-level language models. This powerful idea has become more popular recently, but in order to read those papers, you first need a basic understanding of how language models work with the simpler "UNK" solution.

5 Parameterizing a Language Model and Estimating the Parameters

We proceed through a series of language models.

5.1 Creepy Language Model

The first language model we consider is quite simple. It assigns probability to a sequence $\boldsymbol{x}$ proportional to the number of times that sequence was observed in $\boldsymbol{x}_{1:n}$. More formally:

$$p(\boldsymbol{X} = \boldsymbol{x}) = \frac{|\{i \mid \boldsymbol{x}_i = \boldsymbol{x}\}|}{n} = \frac{c_{\boldsymbol{x}_{1:n}}(\boldsymbol{x})}{n} \tag{7}$$

where $c_{\boldsymbol{x}_{1:n}}(\cdot)$ returns the count of its argument in the training dataset.

This model does a great job of fitting the training dataset; it won't waste any probability mass on sequences not seen in training. But that is its downfall; consider what will happen when we evaluate this model on a test example $\bar{\boldsymbol{x}}$ that happens not to have been seen in training data. Its probability will be 0, and perplexity will be infinite.

I call this the "creepy" language model because of what will happen if we randomly sample from it. Imagine simulated data, drawn according to the distribution p in Equation 7. The only sentences drawn will be ones seen in training data. This model just memorizes what it has seen before; it does not generalize at all.

To get away from the problem of creepiness, we will use the chain rule of probability to decompose the distribution. Here we use $X_{i:j}$ to denote the collection of random variables $\langle X_i, X_{i+1}, \ldots, X_{j-1}, X_j \rangle$, all within the sequence $\boldsymbol{X}$. We also introduce a special start symbol $x_0 = \bigcirc$ that is not in the vocabulary $\mathcal{V}$,

[1] Consider this example: *unlockable* might refer to something that can be unlocked or something that cannot be locked. Both of these would probably break down into ⟨*un-*, *lock*, *-able*⟩, but they correspond to attaching the affixes to *lock* in different orders. In more morphologically rich languages, the potential for ambiguity is greater.

assumed to appear just before every sentence; the random variable X_0 always takes the value x_0.[2]

$$p(\boldsymbol{X} = \boldsymbol{x}) = \left(\begin{array}{l} p(X_1 = x_1 \mid X_0 = x_0) \\ \cdot\, p(X_2 = x_2 \mid X_{0::}1 = x_{0:1}) \\ \cdot\, p(X_3 = x_3 \mid X_{0:2} = x_{0:2}) \\ \vdots \\ \cdot\, p(X_\ell = \bigcirc \mid X_{0:\ell-1} = x_{0:\ell-1}) \end{array} \right) \tag{8}$$

$$= \prod_{j=1}^{\ell} p(X_j = x_j \mid X_{0:j-1} = x_{0:j-1}) \tag{9}$$

Note that this decomposition does not involve any assumptions. Each word is generated conditioned on the history of words that came before, and the entire history is considered.

How would we estimate the component distributions? The starting point for estimating distributions like these is the same as in the creepy language model: probabilities should be proportional to counts (or frequencies). For conditional probabilities like those in Equation 9, the relative frequency estimate is:

$$p(X_j = x_j \mid X_{0:j-1} = x_{0:j-1}) = \frac{c_{\boldsymbol{x}_{1:n}}(x_{0:j})}{c_{\boldsymbol{x}_{1:n}}(x_{0:j-1})} \tag{10}$$

That is, the probability is estimated as the frequency of observing a history/word pair, divided by the frequency of observing just the history (in the training dataset). Plugging in to Equation 9, we get:

$$p(\boldsymbol{X} = \boldsymbol{x}) = \prod_{j=1}^{\ell} \frac{c_{\boldsymbol{x}_{1:n}}(x_{0:j})}{c_{\boldsymbol{x}_{1:n}}(x_{0:j-1})} \tag{11}$$

$$= \frac{\displaystyle\prod_{j=1}^{\ell} c_{\boldsymbol{x}_{1:n}}(x_{0:j})}{\displaystyle\prod_{j=1}^{\ell} c_{\boldsymbol{x}_{1:n}}(x_{0:j-1})} \tag{12}$$

$$= \frac{c_{\boldsymbol{x}_{1:n}}(x_{0:1}) \cdot c_{\boldsymbol{x}_{1:n}}(x_{0:2}) \cdot c_{\boldsymbol{x}_{1:n}}(x_{0:3}) \cdots c_{\boldsymbol{x}_{1:n}}(x_{0:\ell-1}) \cdot c_{\boldsymbol{x}_{1:n}}(x_{0:\ell})}{c_{\boldsymbol{x}_{1:n}}(x_0) \cdot c_{\boldsymbol{x}_{1:n}}(x_{0:1}) \cdot c_{\boldsymbol{x}_{1:n}}(x_{0:2}) \cdot c_{\boldsymbol{x}_{1:n}}(x_{0:3}) \cdots c_{\boldsymbol{x}_{1:n}}(x_{0:\ell-1})} \tag{13}$$

$$= \frac{c_{\boldsymbol{x}_{1:n}}(x_{0:\ell})}{c_{\boldsymbol{x}_{1:n}}(x_0)} \tag{14}$$

$$= \frac{c_{\boldsymbol{x}_{1:n}}(\boldsymbol{x})}{n} \tag{15}$$

This is identical to Equation 7, so it should be clear that nothing has changed yet. (Note that $c_{\boldsymbol{x}_{1:n}}(x_0)$ is exactly n, the number of sentences in the training dataset, since the start symbol $\bigcirc$ occurs before the beginning of each sentence, and only there.)

5.2 Unigram Model

The **unigram model** adds an extremely strong assumption to the above decomposition: every word is distributed *independent* of its history. This gives us a very simple model of language, in which a V-sized die is rolled once for every word, ignoring all previous words. (When the die rolls $\bigcirc$, the sentence ends.) You should see that this model is very different from the creepy language model.

[2] So it isn't really random!

Mathematically:

$$p(X_j = x_j \mid X_{0:j-1} = x_{0:j-1}) \overset{\text{assumption}}{=} p_{\boldsymbol{\theta}}(X_j = x_j) \tag{16}$$

$$p_{\boldsymbol{\theta}}(\boldsymbol{X} = \boldsymbol{x}) = \prod_{j=1}^{\ell} p_{\boldsymbol{\theta}}(X = x_j) \tag{17}$$

We use $\boldsymbol{\theta}$ to denote the set of parameters of the model.

This model requires that we estimate the probability for each word $v \in \mathcal{V}$. The simplest way to do this is to store a value for each word. Let $p(X = v)$ take a value denoted by θ_v ("the unigram probability of v"); its estimate from data is denoted by $\hat{\theta}_v$. The relative frequency estimate for θ_v is:

$$\hat{\theta}_v = \frac{|\{i, j \mid [\boldsymbol{x}_i]_j = v\}|}{N} \tag{18}$$

$$= \frac{c_{\boldsymbol{x}_{1:n}}(v)}{N} \tag{19}$$

where N is the total number of words in the training dataset, $\sum_{i=1}^{n} |\boldsymbol{x}_i|$. Both when calculating N and when estimating the distribution $\boldsymbol{\theta}$, we include the stop symbol ◯; because it occurs once per training data instance, its probability will be $\frac{n}{N}$. We do not count the start symbol ◯, because it is always assumed to be given (not generated by the model).

The unigram model will have V parameters, one for every word in $\mathcal{V}$.

Unigram models are sometimes called "bag of words" models, because they assign the same probability to a sentence $\boldsymbol{x}$ and any permutation of it; from the model's perspective, $\boldsymbol{x}$ is a bag (i.e., a collection that can contain duplicates, unlike a set) of words. This terminology is commonly used when building language models over documents, where each $\boldsymbol{x}$ is a document, rather than a sentence. In general, "bag of words" models view each document as a histogram of word frequencies, without regard to the order the words.

Unigram models are easy to understand, computationally cheap to estimate and store, and they work well for some problems (e.g., information retieval). Linguistically, they are very simplistic, assigning high probability to some absurdly implausible sentences. For example, the most common word in typical English corpora is *the*. This means that $\theta_{\text{the}} > \theta_v$ for all $v \in \mathcal{V} \setminus \{\text{the}\}$. This means that the most common sequence of length $\ell = 5$ is "*the the the the* ◯"; this is more likely than "*I want ice cream* ◯" and "*good morning to you* ◯" and "*make America great again* ◯".

5.3 Bigram Models

The creepy model doesn't generalize well, but the unigram model's strong independence assumption means that it can't pick up on any patterns beyond the relative frequencies of words. The next model we consider weakens that independence assumption, ever so slightly.

A **bigram model** lets each word depend directly on its predecessor, the most recent word:

$$p(X_j = x_j \mid X_{0:j-1} = x_{0:j-1}) \overset{\text{assumption}}{=} p_{\boldsymbol{\theta}}(X_j = x_j \mid X_{j-1} = x_{j-1}) \tag{20}$$

$$p_{\boldsymbol{\theta}}(\boldsymbol{X} = \boldsymbol{x}) = \prod_{j=1}^{\ell} p_{\boldsymbol{\theta}}(X_j = x_j \mid X_{j-1} = x_{j-1}) \tag{21}$$

This model also is known as a (first-order) Markov model, because the assumption in Equation 20 is a (first-order) Markov assumption.

The parameters of this model, $\boldsymbol{\theta}$, include a value for the probability of every word v in $\mathcal{V}$, conditioned on every word v' in $\mathcal{V}$, plus the start symbol ◯. The relative frequency estimate is:

$$\hat{\theta}_{v|v'} = \frac{|\{i, j \mid [\boldsymbol{x}_i]_{j-1} = v' \wedge [\boldsymbol{x}_i]_j = v\}|}{|\{i, j \mid [\boldsymbol{x}_i]_{j-1} = v'\}|} \tag{22}$$

$$= \frac{c_{\boldsymbol{x}_{1:n}}(v'v)}{\sum_{u \in \mathcal{V}} c_{\boldsymbol{x}_{1:n}}(v'u)} \tag{23}$$

It's important to notice that the denominator here is the count of the *history* word, occurring as a history word with any vocabulary item $u \in \mathcal{V}$. That means that when we estimate (for example) $\theta_{\text{The}|◯}$, the denominator will count the number of times some word occurs after ◯ (which will be the number of sentences n, since every sentence's first word follows an implied ◯).

The number of parameters that need to be estimated for the bigram model is given by the number of histories ($V + 1$, accounting for ◯) times the vocabulary size (V): $V(V + 1)$.

5.4 n-Gram Models

Generalizing from unigram and bigram models, **n-gram models** condition each word's probability on the history of the most recent $(\mathsf{n} - 1)$ words, also known as a $(\mathsf{n} - 1)$-order Markov assumption. When $\mathsf{n} = 3$, we have a **trigram model**; for larger values of n, we simply use the number (e.g., "four-gram," "five-gram," etc.).

Here is the general form for n-gram models:

$$p(X_j = x_j \mid X_{0:j-1} = x_{0:j-1}) \overset{\text{assumption}}{=} p_{\boldsymbol{\theta}}(X_j = x_j \mid X_{j-\mathsf{n}+1:j-1} = x_{j-\mathsf{n}+1:j-1}) \tag{24}$$

$$p_{\boldsymbol{\theta}}(\boldsymbol{X} = \boldsymbol{x}) = \prod_{j=1}^{\ell} p_{\boldsymbol{\theta}}(X_j = x_j \mid X_{j-\mathsf{n}+1:j-1} = x_{j-\mathsf{n}+1:j-1}) \tag{25}$$

The parameters of this model, $\boldsymbol{\theta}$, include a value for the probability of every word v in $\mathcal{V}$, conditioned on every history. Histories are now a bit more complicated, because we need a $(\mathsf{n} - 1)$-length history for every word, including the first one (which up until now was only preceded by $x_0 = ◯$). To keep notation simple, we assume that every sentence is preceded by as many copies of ◯ as we need to ensure that x_1 has a $(\mathsf{n} - 1)$-length history. This means that $x_{-(\mathsf{n}-2)} = x_{-(\mathsf{n}-3)} = \cdots = x_{-1} = x_0 = ◯$. We use $\boldsymbol{h}$ to denote a history of length $(\mathsf{n} - 1)$. The relative frequency estimate is:

$$\hat{\theta}_{v|\boldsymbol{h}} = \frac{|\{i, j \mid [\boldsymbol{x}_i]_{j-\mathsf{n}+1:j-1} = \boldsymbol{h} \wedge [\boldsymbol{x}_i]_j = v\}|}{|\{i, j \mid [\boldsymbol{x}_i]_{j-\mathsf{n}+1:j-1} = \boldsymbol{h}\}|} \tag{26}$$

$$= \frac{c_{\boldsymbol{x}_{1:n}}(\boldsymbol{h}v)}{\sum_{u \in \mathcal{V}} c_{\boldsymbol{x}_{1:n}}(\boldsymbol{h}u)} \tag{27}$$

As before, the denominator here is the count of the *history* word, occurring as a history word.

Notice that, as we increase n, these models approach the creepy language model, remembering more and more of each word's history. With relative frequency estimation, they will never allow a word to occur following a history it wasn't observed with in the training dataset. As we decrease n, we approach a unigram model that simply ignores the history. The best n value for your problem will depend on the training dataset, especially its size (n and N), and on the size of the vocabulary (V). When making a high-level design choice like n, it is advisable to reserve a portion of data, separate from your training dataset, and use it to make the choice. High-level choices like n's value are sometimes called **model selection** or **hyperparameter selection**. Here's a simple procedure for choosing n:

1. When constructing your training dataset, hold some data aside (i.e., do not include it in $\boldsymbol{x}_{1:n}$). Call this your **development dataset**; we'll refer to it as $\dot{\boldsymbol{x}}_{1:d}$.
2. Let $\mathcal{N}$ be the set of values for n you are willing to consider.
3. For each $g \in \mathcal{N}$:
   (a) Estimate the parameters $\hat{\boldsymbol{\theta}}^{(g)}$ for a g-model from the training dataset $\boldsymbol{x}_{1:n}$.
   (b) Calculate perplexity$(p_{\hat{\boldsymbol{\theta}}^{(g)}}; \dot{\boldsymbol{x}}_{1:d})$, the perplexity of the g-gram model on the development dataset.
4. Choose the g^* with the lowest development-dataset perplexity, and let $\mathsf{n} = g^*$.

n-gram models are fairly easy to build, requiring only a pass over the data to gather all of the appropriate counts, then simple operations to obtain relative frequency estimates. When working with very large training datasets some engineering is required to work with them efficiently. Many of the key tricks are implemented in open-source implementations like KenLM[3] and SRILM.[4]

5.5 Data Sparsity in n-Gram Model Estimation

n-models suffer from a curse-of-dimensionality problem. As n increases, the number of parameters to be estimated increases exponentially. This means that getting good estimates for each and every n-gram's $\hat{\theta}_{v|\boldsymbol{h}}$ requires much more data. In concrete terms, the vast majority of n-grams will never be observed, even if they are linguistically plausible. This is sometimes referred to as **data sparseness**, because the count values will be mostly zeroes. Imagine a vector holding $c_{\boldsymbol{x}_{1:n}}(\boldsymbol{h}v)$ for a single history, and all $v \in \mathcal{V}$; as n goes up, vectors will get sparser and sparser, since each n-in the corpus now has a longer and more detailed history to associate with.

(Data sparseness can be mitigated somewhat by mapping more and more words to UNK, but this comes at a cost of expressive power. A model that maps most words to UNK will miss a lot of the nuance in natural language!)

We presented relative frequency estimation (counting n-grams and their $(\mathsf{n}-1)$-length histories, then dividing) uncritically. In fact, relative frequency estimation is motivated not just by intuition, but by the **maximum likelihood principle**. This is an idea from statistics, telling us that we should choose parameter estimates that make the training dataset as likely as possible. In notation:

$$\hat{\boldsymbol{\theta}}_{\text{MLE}} = \operatorname*{argmax}_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}}(\boldsymbol{x}_{1:n}) \tag{28}$$

It is not difficult to prove that relative frequency estimation gives the maximum likelihood estimate (MLE) for conditional categorical distributions like those our n-gram models are built out of.

One of the reasons statisticians like the MLE in general is theoretical. In the limit as $n \to \infty$, the MLE will converge on the true distribution that generated the data, provided that the model family (here, an n-model for a particular value of n) includes the true distribution. In NLP, we almost never believe that the model family includes the true distribution; we accept that our models are imperfect and simply try to make them as useful as possible.

And MLE n-models are not very useful, because, for history $\boldsymbol{h}$, the probability of any word not observed immediately after that history will be 0. Further, for any *history* that wasn't observed, the distribution over words that can immediately follow it is completely undefined (since its frequency is 0). This spells disaster for any perplexity calculations!

If you've taken a class in machine learning, you've likely learned about the problem of **overfitting**, that is, when a model's performance on training data is very good, but it doesn't generalize well to test (or

[3] https://kheafield.com/code/kenlm/
[4] http://www.speech.sri.com/projects/srilm/

development) data. The inability of MLE n-gram models to deal well with data sparsity can be seen as an example of overfitting.

5.6 Smoothing

In order to make n-gram models useful, we must apply a **smoothing** transformation that eliminates zero-counts, both for histories and for n-grams. There are many methods available for smoothing, and some of them are quite complicated. To understand why, consider the original definition of a language model, which requires that the sum of probabilities over all sequences in $\mathcal{V}^\dagger$ be one (Equation 2. It is straightforward to show that this constraint is met by MLE with relative frequency estimation; it is harder to guarantee when we start manipulating the estimation procedure.

Instead of explaining specific smoothing algorithms in detail, we will consider some "safe" transformations that provide building blocks for smoothing.

5.6.1 Linear Interpolation

Noting that different values of n lead to models with different strengths and weaknesses, we can build a language model that interpolates among them. Suppose we have two language models, a unigram model $p_{\boldsymbol{\theta}^{(1)}}$ and a bigram model $p_{\boldsymbol{\theta}^{(2)}}$. Let the interpolated model be given by:

$$\hat{\theta}^{(\text{interp})}_{v|v'} = \alpha\hat{\theta}^{(1)}_{v} + (1-\alpha)\hat{\theta}^{(2)}_{v|v'} \tag{29}$$

where $\alpha \in [0, 1]$. This is a bigram model whose estimate is given partly by MLE and partly by a unigram MLE estimate. Unlike the unsmoothed bigram estimate, this inteprolated estimate will never assign zero to a word that has a nonzero unigram count, as long as $\alpha > 0$. Unlike the unsmoothed unigram estimate, the interpolated estimate pays attention to the context, as long as $\alpha < 1$.

The above idea generalizes to any number (K) of models. Let $\boldsymbol{\alpha} \in \mathbb{R}^K_+$ be a vector of positive interpolation coefficients that sums to one. Then let

$$\hat{\theta}^{(\text{interp})}_{v|\boldsymbol{h}} = \sum_{k=1}^{K} \alpha_k \hat{\theta}^{(k)}_{v|\boldsymbol{h}} \tag{30}$$

(We abuse notation slightly; some models may not define $\theta^{(k)}_{v|\boldsymbol{h}}$ for the full history $\boldsymbol{h}$. In such cases, we truncate the history as needed to fit the model.)

This idea leads naturally to the question of how to choose $\boldsymbol{\alpha}$. For the unigram/bigram example above, it's straightforward to show that if we follow the maximum likelihood principle and choose α to make the training dataset as likely as possible, we will pick $\alpha = 0$; a bigram model has more flexibility than a unigram model and can always make the training dataset more likely.

We therefore use development data to choose $\boldsymbol{\alpha}$, simulating the test-time experiment where the model is exposed to new data. $\boldsymbol{\alpha}$ is an example of a hyperparameter, so a model selection method like the one presented in Section 5.4 (for choosing n) can be applied.

5.6.2 Additive Smoothing

Another way to avoid zeroes transform the counts directly before estimating $\boldsymbol{\theta}$. Consider, for example, the estimation formula for a trigram probability:

$$\hat{\theta}^{(\text{MLE})}_{v|v''v'} = \frac{c_{\boldsymbol{x}_{1:n}}(v''v'v)}{\sum_{u\in\mathcal{V}} c_{\boldsymbol{x}_{1:n}}(v''v'u)} \tag{31}$$

If, prior to this calculation, we augmented every count by a fixed quantity λ, we'd have:

$$\hat{\theta}^{(\text{add-}\lambda)}_{v|v''v'} = \frac{\lambda + c_{\boldsymbol{x}_{1:n}}(v''v'v)}{\sum_{u\in\mathcal{V}}(\lambda + c_{\boldsymbol{x}_{1:n}}(v''v'u))} = \frac{\lambda + c_{\boldsymbol{x}_{1:n}}(v''v'v)}{V\lambda + \sum_{u\in\mathcal{V}} c_{\boldsymbol{x}_{1:n}}(v''v'u)} \tag{32}$$

For $\lambda > 0$, the resulting estimate is always strictly positive.

How to choose λ? Again, we treat λ as a hyperparameter and apply model selection. The value $\lambda = 1$ is often presented first, and has a special name ("Laplace smoothing"), but in practice the best values are usually smaller than one (i.e., fractional). It is also possible to choose different values of λ for different histories.

5.6.3 Discounting

A third tool, similar in spirit to adding λ, is **discounting**, where we subtract away from each count before normalizing into probabilities. Let $\delta \in (0, 1)$ be the discount value. For a given history $\boldsymbol{h}$, we divide the vocabulary $\mathcal{V}$ into two disjoint sets:

$$\mathcal{A}(\boldsymbol{h}) = \{v \in \mathcal{V} : c_{\boldsymbol{x}_{1:n}}(\boldsymbol{h}v) > 0\} \tag{33}$$

$$\mathcal{B}(\boldsymbol{h}) = \{v \in \mathcal{V} : c_{\boldsymbol{x}_{1:n}}(\boldsymbol{h}v) = 0\} \tag{34}$$

For $v \in \mathcal{A}(\boldsymbol{h})$, define:

$$\hat{\theta}^{(\text{disc})}_{v|\boldsymbol{h}} = \frac{c_{\boldsymbol{x}_{1:n}}(\boldsymbol{h}v) - \delta}{\sum_{u\in\mathcal{V}} c_{\boldsymbol{x}_{1:n}}(\boldsymbol{h}u)} \tag{35}$$

The result of this transformation will be that the probabilities sum up to a value less than one. Let the "missing mass" be denoted by

$$q(\boldsymbol{h}) = 1 - \sum_{v\in\mathcal{A}(\boldsymbol{h})} \hat{\theta}^{(\text{disc})}_{v|\boldsymbol{h}} \tag{36}$$

This mass, $q(\boldsymbol{h})$, will be divided up among all of the words in $\mathcal{B}(\boldsymbol{h})$. One common way to do it is to divide up proportionally according to a shorter-n n-gram model. This is known as **backoff**. A simpler way is to simply divide it up uniformly across $\mathcal{B}(\boldsymbol{h})$. The value of δ can be chosen using model selection.

Note that, in general, δ need not be a fixed value for every history length, every history frequency, or even every history. One famous method for discounting, **Good-Turing discounting** takes the total count mass for words with observed count c and redistributes it among words with observed count $c - 1$. This means that, for a given history, zero-frequency words are given a probability similar to what we would normally assign to words that occurred once, which are given a probability similar to what we would normally assign to words that occurred twice, and so on.

6 Algorithms

There are three useful language model-related algorithms you should be able to figure out:

1. Given a language model p and a sentence $\boldsymbol{x}$, calculate $p(\boldsymbol{x})$, or its logarithm.[5] You should be able to do this in $O(\ell)$ runtime, where ℓ is the length of the sentence. Because language model probabilities are tiny and can lead to underflow, the following equation is extremely useful:

$$\log p(\boldsymbol{x}) = \sum_{j=1}^{\ell} \log p(x_j \mid x_{0:j-1}) \tag{37}$$

[5]The logarithm is sometimes preferred to avoid underflow, and it's what you need for perplexity (Equation 6).

That is, summing logarithms of probabilities (log-probabilities) is a smart way to avoid the underflow that would likely result if we multiplied together probabilities. It doesn't really matter which logarithm base you use, as long as you exponentiate with the same base (the default in most programming languages is base e).

2. Estimate a language model's parameters from a training dataset. You should be able to do this in $O(N)$ runtime, where N is the number of words in the training data, and also $O(N)$ space.[6] Importantly, you should not require $O(V^{\mathsf{n}})$ space, which is what you would need if you naïvely stored every n-gram's probability! Depending on your smoothing method, many n-grams will have the same probability, and you will want to store that value just once if you can. In particular, an n-gram that had a training dataset frequency of zero should probably take a "default" probability value, depending on some properties of the history it contains.
3. Randomly sample a sentence $\boldsymbol{x}$ from a language model p. You should be able to sample each word in $O(V)$ time.

7 Log-Linear Language Models

Another solution to the data sparseness problem is to rethink the way we parameterize language models. Instead of associating a parameter $\theta_{v|\boldsymbol{h}}$ with every n-gram, the approach we consider next converts the n-gram into a collection of attributes (known as **features**) and uses those to assign it a probability.

See Collins [2011b] or Smith [2004] for general introductions to log-linear models. Here we present them for the language modeling case, specifically.

To define a log-linear language model, we first introduce a function $\boldsymbol{\phi}$ that maps a history and a word (in $\mathcal{V}$) to a real-valued vector in $\mathbb{R}^d$. Associated with the kth dimension of this vector is a coefficient or weight w_k, and together these are stacked into $\mathbf{w} \in \mathbb{R}^d$, which are the parameters of the model. The form of the language model is given by:

$$p_{\mathbf{w}}(\boldsymbol{X} = \boldsymbol{x}) = \prod_{j=1}^{\ell} p_{\mathbf{w}}(X_j = x_j \mid X_{0:j-1} = x_{0:j-1}) \tag{38}$$

$$= \prod_{j=1}^{\ell} \frac{\exp \mathbf{w} \cdot \boldsymbol{\phi}(x_{0:j-1}, x_j)}{Z_{\mathbf{w}}(x_{0:j-1})} \tag{39}$$

$$\overset{\text{assumption}}{=} \prod_{j-1}^{\ell} \frac{\exp \mathbf{w} \cdot \boldsymbol{\phi}(x_{j-\mathsf{n}+1:j-1}, x_j)}{Z_{\mathbf{w}}(x_{j-\mathsf{n}+1:j-1})} \tag{40}$$

$$= \prod_{j=1}^{\ell} \frac{\exp \mathbf{w} \cdot \boldsymbol{\phi}(\boldsymbol{h}_j, x_j)}{Z_{\mathbf{w}}(\boldsymbol{h}_j)} \tag{41}$$

The assumption in Equation 40 is the standard $(\mathsf{n}-1)$-order Markov assumption, and the result is that the feature vector $\boldsymbol{\phi}$ maps n-grams ($\boldsymbol{h}v$) to vectors. The $Z_{\mathbf{w}}$ function in the denominator is shorthand for a summation that guarantees that the distribution over words given history $\boldsymbol{h}$ will sum to one. Its full form is:

$$Z_{\mathbf{w}}(\boldsymbol{h}) = \sum_{v \in \mathcal{V}} \exp \mathbf{w} \cdot \boldsymbol{\phi}(\boldsymbol{h}, v) \tag{42}$$

The form of the log-linear language model is a bit daunting at first. It's helpful to consider how it's built from the inside out, which we'll do in the next few sections.

[6] If you use interpolation or backoff, you might need $O(N\mathsf{n})$ runtime and space.

7.1 Features

When we consider an n-gram $\boldsymbol{h}v$, we start by mapping it to its feature vector $\boldsymbol{\phi}(\boldsymbol{h}v)$. Some kinds of features conventionally used include:

- Indicators for n-grams. For example, a feature for the trigram *do not disturb* would return 1 if and only if the last two words in $\boldsymbol{h}$ are *do not* and v is *disturb*; otherwise it returns 0. There are V^3 trigram features (and V^{n} features for general n-length n-grams).
- Indicators for *gappy* n-grams. This allows, for example, *the old man*, *the healthy man*, *the bearded man*, and so on, to all have probabilities that rise or fall together, through a feature that returns 1 if and only if the second-to-last word in $\boldsymbol{h}$ is *the* and v is *man*.
- Spelling features, such as whether a word begins with a vowel or a capital letter, or whether a word contains a digit. These can be conjoined with a history word; for example, a feature that returns 1 if and only if the last word of $\boldsymbol{h}$ is *an* and v starts with a vowel could let the language model learn to prefer *an* (to *a*) before words that start with vowels.
- Class features, that use some external resource to define sets of words that share membership in some class like "protein names" or "geographic place names" or "transitive verbs." Such a feature returns 1 if v is in the class, and 0 if it isn't; as with the spelling features, it can be conjoined with some feature of the history.

You can define any features as you'd like, as long as they can be calculated by considering only $\boldsymbol{h}$ and v.[7] It's common in NLP for features to be binary indicators, like those above, and to form new features by conjoining the values of simpler binary indicator features.

The first challenge of log-linear language models is in choosing good features. If you choose too many, your model will be prone to overfitting the training dataset. If you don't choose enough features that capture the properties of the language, your model will not learn to generalize.

7.2 Coefficients

Once we've mapped $\boldsymbol{h}v$ to its feature vector representation $\boldsymbol{\phi}(\boldsymbol{h}, v)$, we take an inner product with the coefficients $\mathbf{w}$:

$$\mathbf{w} \cdot \boldsymbol{\phi}(\boldsymbol{h}, v) = \sum_{k=1}^{d} w_k \phi_k(\boldsymbol{h}, v) \tag{43}$$

This linear mapping can be understood as projecting our n-gram to a linear score. For a binary feature ϕ_k, the value of the corresponding coefficient w_k tells us immediately how the presence of the feature changes the score:

- If $w_k > 0$, then the score of any n-gram that "has" this feature (i.e., where $\phi_k(\boldsymbol{h}, v) = 1$) increases by w_k. This will make the n-gram more likely.
- If $w_k < 0$, then the score of any n-gram that "has" this feature (i.e., where $\phi_k(\boldsymbol{h}, v) = 1$) decreases by $-w_k$. This will make the n-gram less likely.
- If $w_k = 0$, then the score of any n-gram that "has" this feature (i.e., where $\phi_k(\boldsymbol{h}, v) = 1$) is unaffected.

[7] It can be shown that a feature that depends only on $\boldsymbol{h}$ and *not* v will have no effect on the probability distribution for history $\boldsymbol{h}$. This is left as an exercise.

7.3 Softmax: Exponentiate and Normalize

Once we map n-grams to scores, we can transform the scores into probability distributions for each history $\boldsymbol{h}$, by applying a transformation often called the **softmax**:

$$\text{softmax}\left(\langle a_1, a_2, \ldots, a_V \rangle\right) = \left\langle \frac{e^{a_1}}{\sum_{k=1}^{V} e^{a_k}}, \frac{e^{a_2}}{\sum_{k=1}^{V} e^{a_k}}, \ldots, \frac{e^{a_V}}{\sum_{k=1}^{V} e^{a_k}} \right\rangle \tag{44}$$

The softmax takes every value in a vector—here, the collection of scores for words in $\mathcal{V}$ paired with a single history $\boldsymbol{h}$—and exponentiates them before renormalizing them. This transformation has some desirable properties:

- It preserves monotonicity (i.e., if $a_i > a_j$ then the ith value of the new vector will be larger than the jth value of the new vector). This means that the higher an n-gram's score is, the higher its probability will be.
- It gives a proper probability distribution over $\mathcal{V}$, i.e., one comprised of nonnegative values that sum to one.
- If the scores are all finite, then every n-gram will have positive probability; there will be no zeroes.

7.4 Parameter Estimation

Unfortunately, there is no closed-form solution for the maximum likelihood estimate of log-linear models, or for any principled method that seeks to avoid overfitting. Instead, the parameter estimation problem is typically solved using a convex optimization algorithm.

We let i index the N words in our training dataset, rather than the n sentences. Each word's history is defined as before (i.e., if it is the first word in a sentence, then its history is $(\mathsf{n}-1)$ copies of $\bigcirc$). Here is the form of the maximum likelihood estimation problem for log-linear language models, written in terms of log-likelihood:[8]

$$\max_{\mathbf{w} \in \mathbb{R}^d} \sum_{i=1}^{N} \log p_{\mathbf{w}}(x_i \mid \boldsymbol{h}_i) \tag{45}$$

$$= \max_{\mathbf{w} \in \mathbb{R}^d} \sum_{i=1}^{N} \log \frac{\exp \mathbf{w} \cdot \boldsymbol{\phi}(\boldsymbol{h}_i, v)}{Z_{\mathbf{w}}(\boldsymbol{h}_i)} \tag{46}$$

$$= \max_{\mathbf{w} \in \mathbb{R}^d} \sum_{i=1}^{N} \mathbf{w} \cdot \boldsymbol{\phi}(\boldsymbol{h}_i, x_i) - \log Z_{\mathbf{w}}(\boldsymbol{h}_i) \tag{47}$$

This problem is *concave* in $\mathbf{w}$ and *differentiable* with respect to $\mathbf{w}$, which means that we can negate it and apply standard methods for convex optimization to solve it with high precision. Two popular methods are:

- L-BFGS, a "batch" method that iteratively calculates the gradient of the total log-likelihood with respect to $\mathbf{w}$ and updates in the gradient direction, with clever math for deciding how far to go in that direction; and
- stochastic gradient descent, a method that calculates the gradient of the log-likelihood with respect to one (or a small number) of instances i, often chosen by random shuffling of the data, and updating $\mathbf{w}$ after each such calculation.

There is an intuitive way to understand Equation 47. For each word x_i in the training set, the goal is to increase the score that it receives with its observed history $\boldsymbol{h}$ (the first term inside the summation over i).

[8] Since the logarithm function is monotonic, the parameters that maximize log-likelihood will be identical to those that maximize likelihood.

We can't stop there, though; if we did, then the trivial solution would drive every w_k to $+\infty$. We must give the optimization method something to *decrease*, and that's where the second term, the logarithm of $Z_{\mathbf{w}}(\boldsymbol{h})$ comes in. We sometimes refer to these two components as the "hope" and "fear" parts of the objective. Letting $s(v)$ denote the linear score of a word with history $\boldsymbol{h}$ (suppressed for clarity), we have:

$$\text{increase } \underbrace{s(x_i)}_{\text{hope}} \text{ while decreasing } \underbrace{\log \sum_{v \in \mathcal{V}} \exp s(v)}_{\text{fear}} \tag{48}$$

Note that the "$\log \sum \exp$" transformation is a smooth upper bound on the $\max$ function. It arises frequently in machine learning. Here, the intuition comes from imagining that it (approximately) picks out the score of the currently-most probable v given history $\boldsymbol{h}$, and tries to push it down. In fact, the scores of *all* v get pushed down, in aggregate. The only way to achieve the maximum likelihood solution is to achieve balance between hope and fear, so that the score of x_i is as close as possible to the (approximately) most-probable next word.

Of course, in real data, many words may share this history, so really what we're trying to achieve for history $\boldsymbol{h}$ is:

$$\text{increase } \sum_{i:\boldsymbol{h}_i = \boldsymbol{h}} \underbrace{s(x_i)}_{\text{hope}} \text{ while decreasing } |\{i : \boldsymbol{h_i} = \boldsymbol{h}\}| \cdot \underbrace{\log \sum_{v \in \mathcal{V}} \exp s(v)}_{\text{fear}} \tag{49}$$

So, in aggregate, we want the scores of words that actually follow $\boldsymbol{h}$ to come close to the (approximately) most probable word's scores. Words that occur with $\boldsymbol{h}$ more often will be counted more times in the "hope" term, so we'll have more to gain by increasing their probabilities—just as in relative frequency estimation for n-gram models.

Of course, because the coefficients $\mathbf{w}$ are shared across all histories, we can't solve the above problem separately for each history by itself. The larger maximum likelihood problem considers all of the data together, balancing between hope and fear across all histories.

Notice that calculating $Z_{\mathbf{w}}$ requires summing up scores over the vocabulary size; simply calculating the objective function will require $O(VNd)$ runtime. This implies that solving Equation 47 will be a fairly expensive operation, especially compared to classical n-gram models. This is one reason that log-linear models never became truly mainstream.

A second challenge with log-linear language models is that they tend to overfit. The overfitting problem here has less to do with data sparsity and more with the rich expressive power of models that use many features. It's straightforward to show that, under certain conditions, a feature's coefficient might tend toward positive or negative infinity as we approach the maximum likelihood solution. One way to prevent this is to **regularize** the log-likelihood objective by adding a penalty that grows as coefficients take on more extreme values. The simplest version of this is a quadratic penalty:

$$\max_{\mathbf{w} \in \mathbb{R}^d} \left(\sum_{i=1}^{N} \mathbf{w} \cdot \boldsymbol{\phi}(\boldsymbol{h}_i, x_i) - \log Z_{\mathbf{w}}(\boldsymbol{h}_i) \right) - \lambda \sum_{k=1}^{d} w_k^2 \tag{50}$$

where λ is a hyperparameter chosen via model selection. This technique is sometimes called (squared) ℓ_2 regularization, because the penalty is proportional to the squared ℓ_2-norm of the $\mathbf{w}$ vector, written as $\|\mathbf{w}\|_2^2$. The ℓ_1 norm can also be used; it has the interesting and arguably desirable property of driving many of the w_k to 0, so that they can be safely eliminated from the model.

References

Michael Collins. Course notes for COMS w4705: Language modeling, 2011a. URL http://www.cs.columbia.edu/~mcollins/courses/nlp2011/notes/lm.pdf.

Michael Collins. Log-linear models, MEMMs, and CRFs, 2011b. URL http://www.cs.columbia.edu/~mcollins/crf.pdf.

Daniel Jurafsky and James H. Martin. N-grams (draft chapter), 2016. URL https://web.stanford.edu/~jurafsky/slp3/4.pdf.

Noah A. Smith. Log-linear models, 2004. URL http://homes.cs.washington.edu/~nasmith/papers/smith.tut04.pdf.