N-gram Language Models

We will introduce models that assign a probability to each possible next word given a sequence of words. These models will also assign probabilities to entire sentences.
This is useful in applications like speech recognition, spelling correction, and machine translation.

Models that do this are called LMs; a simple type of LM is the n-gram model. An n-gram is a sequence of n words.

N-Grams

Suppose we want to compute $P(w \mid h)$, the probability of a word w occurring given the history h. We could just count the occurrences and do it, but sometimes we don't have enough in the corpus.
Similarly, if we wanted to know $P(w_{1:n})$, we would have to count all possible n-word sequences. This is infeasible.

However, one step we can take is decomposing the probabilities using the chain rule:

$$P(w_{1:n}) = \prod_{k=1}^{n} P(w_k \mid w_{1:k-1}).$$

The intuition of n-grams is that we can approximate the history ofa word by its last $n-1$ words, *i.e.*,

$$P(w_n \mid w_{1:n-1}) \approx P(w_n \mid w_{n-N+1:n-1}).$$

This is called a Markov assumption.

Now, to estimate these probabilities, one possible method is MLE (getting the counts and normalising them to lie between 0 and 1). Thus, for example,

$$P(w_n \mid w_{n-1}) = \frac{c(w_{n-1}w_n)}{\sum_w c(w_{n-1}w)}.$$

Note that we usually work with the log of the probabilities, adding them up instead of multiplying, and taking the exponent only if they have to be reported at the end.

Evaluating Language Models

Language models can be evaluated extrinsically or intrinsically. For intrinsic evaluation, we need an unseen test set, apart from the training set.
A language model that assigns a higher probability to the test set is considered better.

Perplexity

The perplexity of a model on a test set is the inverse probability, normalised by the number of words, *i.e.*,

$$\text{PP}(W) = P(w_1 \dots w_N)^{-frac1N}.$$

Thus, the higher the conditional probability, the lower the perplexity – we aim to minimise perplexity.
The perplexity can also be considered the *weighted average branching factor* of a language. Suppose we had to recognise the digits in English, given that they all have equal probability. Then the perplexity would evaluate to

$$\left(\frac{1}{10^N}\right)^{-\frac{1}{N}},$$

which is 1.

The perplexities of different n-gram models show that higher values of n usually yield better results.

Sampling Sentences from a Language Model

Sampling from a language model means generating sentences from it, choosing sentences according to their likelihood. Thus, we start with $n-1$ start tokens and a random word, and generate tokens until the end token is generated.

Generalisation and Zeroes

As n-gram models are dependent on the training corpus, probabilities often encode specific facts about the corpus. Further, higher-order n-grams perform better.

We need to be sure that the training corpus has a similar genre to the task at hand. We should also get the appropriate dialect or variety of the language.

The problem of sparsity also arises.

Unknown Words

Words which the model has not seen before are called OOV words. The percentage of OOV words occurring in the test set is called the OOV rate. An open vocabulary system is one in which we model these words by adding a word `<UNK>`.

There are two ways to train the probabilities with this. One is to choose a fixed vocabulary and convert all OOV tokens in the training set to `<UNK>` in a text normalisation step, and treat it like any other word. The other is to replace words in the training data whose frequencies are under some threshold.
The choice of strategy can affect the perplexity of the model.

Smoothing

We are still left with the problem of unseen n-grams, however (even if the words are known). Smoothing enables us to take out some probabiliity from higher-frequency events and assign it to zero-count events.

Laplace (Add-One) Smoothing

This is the simplest way to do smoothing – add one to all the counts, including the zeroes, and then normalise them. It does not perform that well, but it is nevertheless useful for some tasks.

We know that the unsmoothed probability is given by

$$P(w_i) = \frac{c_i}{N}.$$

Laplace smoothing just adds one to each count, and renormalises:

$$P_L(w_i) = \frac{c_i + 1}{N + V}.$$

It is convenient to define an adjusted count c^*, to describe how the algorithm affects just the numerator. Thus we have

$$c_i^* = (c_i + 1)\frac{N}{N + V},$$

which can be turned into a probability P_i^* by normalising by N.

Smoothing can also be viewed as discounting nonzero counts, in which case we define the relative discount as

$$d_c = \frac{c^*}{c}.$$

The effect that this method has on the probabilities of, say, bigrams is:

$$P_L(w_n \mid w_{n-1}) = \frac{c(w_{n-1}w_n) + 1}{\sum_w (c(w_{n-1}w) + 1)} = \frac{c(w_{n-1}w_n) + 1}{c(w_{n-1}) + V}.$$

From this, the changed bigram counts can be calculated

$$c^*(w_{n-1}w_n) = \frac{(c(w_{n-1}w_n) + 1) \cdot c(w_{n-1})}{c(w_{n-1} + V)}.$$

This method causes too much probability mass to be moved to the zero count elements.

Add-k Smoothing

An alternative is add k to the counts, rather than 1, in the above process. k here is an additional parameter which we can fine-tune on the devset. However, it doesn't work much better.

Backoff and Interpolation

There is a source of information we have not been using: $P(w_n \mid w_{n-2}w_{n-1})$ can be estimated by computing $P(w_n \mid w_{n-1})$, a lower-order n-gram. Backoff and interpolation are two processes that make use of this.
In backoff, we use the lower-order n-grams *if* the higher-order ones are insufficient or abset. By contrast, interpolation makes use of n-grams of all orders.

For example, simple linear interpolation has the form

$$\hat{P}(w_n \mid w_{n-2}w_{n-1}) = \lambda_1 P(w_n) + \lambda_2 P(w_n \mid w_{n-1}) + \lambda_3 P(w_n \mid w_{n-2}w_{n-1}),$$

where

$$\sum_i \lambda_i = 1.$$

In a more sophisticated version, the coefficients depend on the context:

$$\hat{P}(w_n \mid w_{n-2}w_{n-1}) = \lambda_1(w_{n-2:n-1})P(w_n) + \lambda_2(w_{n-2:n-1})P(w_n \mid w_{n-1}) + \lambda_3(w_{n-2:n-1})P(w_n \mid w_{n-2}w_{n-1}).$$

In both these, the values of λ are learnt from an additional corpus called the held-out corpus. The EM algorithm is one way to find the optimal values.

In a backoff model, if an n-gram has zero counts, we approximate it by backing off to the $(n-1)$-gram, and repeat till we find a nonzero count. However, in order to account for the extra probability mass brought in by this process, we need to discount high-frequency events and a function α to distribute the mass. This kind of backoff is called Katz backoff.

$$P_K(w_n \mid w_{n-N+1:n-1}) = \begin{cases} P^*(w_n \mid w_{n-N+1:n-1}), & c(w_{n-N+1:n}) > 0 \\ \alpha(w_{n-N+1:n-1})P_K(w_n \mid w_{n-N+2:n-1}), & \text{otherwise.} \end{cases}$$

Katz backoff is often combined with Good-Turing smoothing.

Kneser-Ney Smoothing

The basis of Kneser-Ney smoothing lies in absolute discounting. In this method, a fixed discount is subtracted from each count. This does not affect the higher counts, and mainly modifies lower counts (which are unreliable), possibly improving them.
Interpolated absolute discounting for bigrams thus has the form

$$P_A D(w_i \mid w_{i-1}) = \frac{c(w_{i-1}w_i) - d}{\sum_v c(w_{i-1}v)} + \lambda(w_{i-1})P(w_i).$$

Kneser-Ney discounting augments this method with a way to handle the lower-order distribution. Instead of creating a unigram model to answer the question "how likely is a word to appear?", it creates one to see how likely a word is to appear *as a novel continuation.* This P_{cont} is based on the number of different contexts the word has appeared in. Thus

$$P_{\text{cont}}(w) \propto |\{v : c(vw) > 0\}|,$$

which can be normalised to become a true distribution:

$$P_{\text{cont}}(w) = \frac{|\{v : c(vw) > 0\}|}{|\{(u', w') : c(u'w') > 0\}|}.$$

We could equivalently have used the number of words the word is preceded by.

Thus a frequent word occurring in only one context will have a low P_{cont} value. We use this in the final interpolated Kneser-Ney smoothing expression:

$$P_{KN}(w_i \mid w_{i-1}) = \frac{\max(c(w_{i-1}w_i) - d, 0)}{c(w_{i-1})} + \lambda(w_{i-1})P_{\text{cont}}(w_i).$$

λ is a constant that allows us to normalise; its value is given by

$$\lambda(w_{i-1}) = \frac{d}{\sum_v c(w_{i-1}v)} |\{w : c(w_{i-1}w) > 0\}|.$$

For general n-grams,

$$P_{KN}(w_i \mid w_{i-n+1:i-1}) = \frac{\max(c_{KN}(w_{i-n+1:i}) - d, 0)}{\sum_v c_{KN}(w_{i-n+1:i-1}v)} + \lambda(w_{i-n+1:i-1})P_{KN}(w_i \mid w_{i-n+2:i-1}),$$

where c_{KN} is the ordinary count for the highest-order n-gram, and the continuation count for the lower order n-grams (the number of unique single-word contexts).

As a base case, unigrams are interpolated with the uniform distribution:

$$P_{KN}(w) = \frac{\max(c_{KN}(w) - d, 0)}{\sum_{w'} c_{KN}(w')} + \lambda(\epsilon)\frac{1}{V}.$$

There is a modified version of this method which performs better.

Huge Language Models and Stupid Backoff

Efficiency is important when the sets of n-grams are large. Words are stored as 8-byte hash numbers in memory, probabilities with 4-8 bits, and n-grams in reverse tries.
n-grams may also be stored only if their counts are greater than some threshold.

While Kneser-Ney smoothing is possible with these algorithms, a simpler algorithm, called stupid backoff, may be sufficient. This method avoids trying to create a true probability distribution via backoff. If a higher-order n-gram has a zero count, we simply weight a lower-order count.

$$S(w_i \mid w_{i-N+1:i-1}) = \begin{cases} \frac{c(w_{i-N+1:i})}{c(w_{i-N+1:i-1})}, & c(w_{i-N+1:i}) > 0 \\ \lambda S(w_i \mid w_{i-N+2:i-1}), & \text{otherwise.} \end{cases}$$

The recursion terminates at the unigram level:

$$S(w) = \frac{c(w)}{N}.$$

Perplexity's Relation to Entropy

Perplexity arises from cross-entropy in information theory and its relationship to entropy, which is a measure of information. The entropy of a random variable X is

$$H(X) = -\sum_{x \in \chi} p(x) \log_2 p(x).$$

Entropy is a lower bound on the number of bits it would take to encode a decision in the optimal coding scheme.

We can analogously define the entropy of a sequence, and the entropy rate (the per-word entropy):

$$\frac{1}{n} H(w_{1:n}) = -\frac{1}{n} \sum_{w_{1:n} \in L} p(w_{1:n}) \log p(w_{1:n}).$$

The entropy of a language is then defined as the limit of this as $n \to \infty$.$ However, we can also take the entropy of a single sequence instead of summing, if the language is stationary (the probability distribution is time-invariant), which Markov models are, and ergodic.

Cross-entropy is useful when we don't know the actual distribution p, but only a model m of it. The cross-entropy of m on p is

$$H(p, m) = \lim_{n \to \infty} -\frac{1}{n} \sum_{W \in L} p(w_{1:n}) \log m(w_{1:n}).$$

Again, for a stationary ergodic process:

$$H(p, m) = \lim_{n \to \infty} -\frac{1}{n} \log m(w_{1:n}).$$

The cross-entropy is an upper bound on the entropy, *i.e.*, $H(p) \leq H(p, m)$ for any m.

We take an approximation to the cross-entropy of a model, relying on a fixed-length sequence. This approximation is

$$H(W) = -\frac{1}{N} \log P(w_{1:N}).$$

The perplexity of a model is then defined as

$$PP(W) = 2^{H(W)}$$