

A decorative graphic consisting of two vertical lines, one blue and one red, positioned to the left of the title.

Language Modeling

Dr. G. Bharadwaja Kumar

Language Models

A language model

- an abstract representation of a (natural) language phenomenon.
- A collection of prior knowledge about a language
- Can be expressed in terms of which words or word sequences are possible or how frequently they occur.

Statistical language models:

A statistical **language model** assigns a probability to a sequence of m words $P(w_1, w_2, \dots, w_m)$ by means of a probability distribution.

The model is based on counting events in a large text corpus, for example, how frequent a certain word or word sequence occurs

Knowledge based models:

If the knowledge comes from a human expert the model is called knowledge-based language model.

If this knowledge is defined by rules, such models are also called rule-based models.

Natural language can be viewed as a stochastic process. Every sentence, document, or other contextual unit of text is treated as a random variable with some probability distribution

Another view of statistical language modeling is grounded in information theory.

Language is considered an information source L , which emits a sequence of symbols w_i from a finite alphabet (the vocabulary).

The emission of the next symbol is highly dependent on the identity of the previous ones—the source L is a high-order Markov chain.

In this view, the trigram amounts to modeling the source as a second-order Markov chain.

They do not capture the meaning of the text. As a result, nonsensical sentences may be deemed “reasonable” by these models (i.e. they may be assigned an unduly high probability).

Statistical models require large amounts of training data, which are not always available. Porting the model to other languages or other domain is thus not always possible.

Statistical models often do not make use of explicit linguistic and domain knowledge. Notwithstanding the comment above regarding overestimating experts’ ability, some useful knowledge can and should be obtained from linguists or domain experts

Why is This Useful?

Speech recognition

Handwriting recognition

Spelling correction

Machine translation systems

Optical character recognizers

For Spell Checkers

Collect list of commonly substituted words
piece/peace, whether/weather, their/there ...

Example:

“On Tuesday, the **whether** ...”

“On Tuesday, the **weather** ...”

Word-based Language Models

A model that enables one to compute the probability, or likelihood, of a sentence S , $P(S)$.

Simple: Every word follows every other word w/ equal probability (0-gram)

Assume $|V|$ is the size of the vocabulary V

Likelihood of sentence S of length n is $= 1/|V| \times 1/|V| \dots \times 1/|V|$

If English has 100,000 words, probability of each next word is $1/100000 =$
.00001

Relative Frequencies and Conditional Probabilities

Relative word frequencies are better than equal probabilities for all words

In a corpus with 10K word types, each word would have

$$P(w) = 1/10K$$

Does not match our intuitions that different words are more likely to occur (e.g. the)

Conditional probability more useful than individual relative word frequencies

dog may be relatively rare in a corpus

But if we see **barking**, $P(\text{dog}|\text{barking})$ may be very large

Word Prediction: Simple vs. Smart

- Smarter: probability of each next word is related to word frequency (unigram)
 - Likelihood of sentence $S = P(w_1) \times P(w_2) \times \dots \times P(w_n)$
 - Assumes probability of each word is independent of probabilities of other words.
- Even smarter: Look at probability *given* previous words
 - Assumes probability of each word is dependent on probabilities of other words.

N-grams: Motivation

- ❑ An **N-gram** is a contiguous sequence of n items of from a given sequence of text
 - Approximation of language characteristics: information in n-grams tells us something about language, but doesn't capture the structure
 - Efficient: finding and using every, e.g., two-word collocation in a text is quick and easy to do.
- ❑ **N-grams** can help in a variety of NLP applications:
 - Word prediction = n-grams can be used to aid in predicting the next word of an utterance, based on the previous $n-1$ words
 - Useful for context-sensitive spelling correction

N-gram Language Model

N-gram model is a type of probabilistic language model compute the probability of a word based on previous N-1 words:

N=1 (Unigram)

N=2 (Bigram)

N=3 (Trigram)

Probabilities are estimated from a corpus of training data (text data).

Once model is known, new sentences can be randomly generated by the model!

Syntax roughly encoded by model, but ungrammatical and semantically “strange” sentences can be produced

-
- We want to compute $P(W) = P(w_1, w_2, w_3, \dots, w_M)$

- From the multiplication rule for three or more variables,

$$P(W) = P(w_1) \cdot P(w_2 | w_1) \cdot P(w_3 | w_1, w_2) \cdots P(w_M | w_1, \dots, w_{M-2}, w_{M-1})$$

- Or, equivalently,

$$P(W) = \prod_{m=1}^M P(w_m | w_1, \dots, w_{m-2}, w_{m-1})$$

- We can call $w_1, \dots, w_{m-2}, w_{m-1}$ the *history* up until word position m , or h_m .
- But, computing $P(w_m | h_m)$ is impractical; we need to compute it for every word position m , and as m increases, it quickly becomes impossible to find data (sentences) containing a history of all words in each sentence position for computing the probability.

-
- Instead, we'll approximate $P(w_m | w_1, \dots, w_{m-2}, w_{m-1})$
$$P(w_m | w_1, \dots, w_{m-2}, w_{m-1}) \approx P(w_m | w_{m-N+1}, \dots, w_{m-2}, w_{m-1})$$

where N is the order of the resulting N -gram language model.

- If $N = 1$, then we have a **unigram** language model, which is the *a priori* probability of word w_m : $P(w_m | w_1, \dots, w_{m-2}, w_{m-1}) \approx P(w_m)$

$$P(W) \approx \prod_{m=1}^M P(w_m)$$

- If $N = 2$, we have a **bigram** language model:

$$P(W) \approx P(w_1) \cdot \prod_{m=2}^M P(w_m | w_{m-1})$$
$$P(w_m | w_1, \dots, w_{m-2}, w_{m-1}) \approx P(w_m | w_{m-1})$$

-
- Given an order for the language model (e.g. trigram), how do we compute $P(w_m \mid w_{m-2}, w_{m-1})$?
 - In theory, this is easy... we count occurrences of word combinations in a database, and compute probabilities by dividing the number of occurrences of a word sequence $(w_{m-N+1}, \dots, w_{m-2}, w_{m-1}, w_m)$ by the number of occurrences of word sequence $(w_{m-N+1}, \dots, w_{m-2}, w_{m-1})$

$$P(w_m \mid w_{m-N+1}, \dots, w_{m-2}, w_{m-1}) = \frac{C(w_{m-N+1}, \dots, w_{m-2}, w_{m-1}, w_m)}{C(w_{m-N+1}, \dots, w_{m-2}, w_{m-1})}$$

Simple N-Grams

An **N-gram** model uses the previous N-1 words to predict the next one:

$$P(w_n \mid w_{n-N+1} w_{n-N+2} \dots w_{n-1})$$

unigrams: $P(\text{dog})$

bigrams: $P(\text{dog} \mid \text{big})$

trigrams: $P(\text{dog} \mid \text{the big})$

quadrigrams: $P(\text{dog} \mid \text{chasing the big})$

-
- In practice, it's more difficult. For example, a 10,000 word vocabulary has 10^{12} (one trillion) trigrams, requiring a *very large* corpus of word sequences to robustly estimate all trillion probabilities.
 - In addition, the success of a language model depends on how similar the text used to estimate language-model parameters is to data seen during evaluation.

Counting Words in Corpora

What is a word?

e.g., are **cat** and **cats** the same word?

September and **Sept**?

zero and **oh**?

Is **seventy-two** one word or two? **AT&T**?

Punctuation?

How many words are there in English?

Where do we find the things to count?

N-grams Issues

Sparse data


Not all N-grams found in training data, need smoothing

Change of domain


Train on WSJ, attempt to identify Shakespeare – won't work

N-grams more reliable than (N-1)-grams

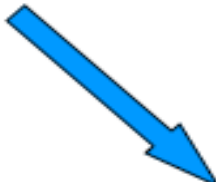
But even more sparse

Larger n-grams  more parameters to estimate

Possible ways to reduce the vocabulary for n-gram models



stemming
(removing the inflectional endings from words)



grouping words into semantic classes
(by pre-existing thesaurus or by induced clustering)

Advantages of n-gram model: simple, easy to calculate, work well to predict words (trigrams, for example).

n-gram models work best when trained on large amounts of data

N-grams Issues

Determine reliable sentence probability estimates
should have smoothing capabilities (avoid the zero-counts)
apply back-off strategies: if N-grams are not possible, back-off to (N-1) grams

$P(\text{"And nothing but the truth"}) \approx 0.001$

$P(\text{"And nuts sing on the roof"}) \approx 0$

Smoothing

Words follow a Zipfian distribution

Small number of words occur very frequently

A large number are seen only once

Zipf's law: *a word's frequency is approximately inversely proportional to its rank in the word distribution list*

Zero probabilities on one bigram cause a zero probability on the entire sentence

So....how do we estimate the likelihood of unseen n-grams?

Smoothing

One major problem with standard N-gram models is that they must be trained from some corpus, and because of that any particular training corpus is finite, some perfectly accepted N-grams are bound to be missing from it.

The bigram matrix for any given training corpus is **sparse**; it is bound to have a very large number of cases of putative “zero probability bigrams” that should really have some non-zero probability.

Smoothing Techniques

Every N-gram training matrix is sparse, even for very large corpora (Zipf's law)

Solution: estimate the likelihood of unseen N-grams

Smoothing – the task of reevaluating some of the zero-probabilities and very low-probabilities N-grams, and assigning them non-zero values.

Add-one Smoothing

Add 1 to every N-gram count

$$P(w_n | w_{n-1}) = C(w_{n-1} w_n) / C(w_{n-1})$$

$$P(w_n | w_{n-1}) = [C(w_{n-1} w_n) + 1] / [C(w_{n-1}) + V]$$

Other Smoothing Methods: Good-Turing

Imagine you are fishing
You have caught 10 Carp, 3 Cod,
2 tuna, 1 trout, 1 salmon, 1 eel.
How likely is it that next species
is new? $3/18$
How likely is it that next is tuna?
Less than $2/18$



Smoothing: Good Turing

How many species (words) were seen once? Estimate for how many are unseen.

All other estimates are adjusted (down) to give probabilities for unseen



$$p_0 = \frac{n_1}{N}$$

$$r^* = (r+1) \frac{n_{r+1}}{n_r}$$

Smoothing: Good Turing Example

10 Carp, 3 Cod, 2 tuna, 1 trout, 1 salmon, 1 eel.

How likely is new data (p_o).

Let n_1 be number occurring
once (3), N be total (18). $p_o = 3/18$

How likely is eel? 1^*

$$n_1 = 3, n_2 = 1$$

$$1^* = 2 \times 1/3 = 2/3$$

$$P(\text{eel}) = 1^* / N = (2/3) / 18 = 1/27$$

Notes:

- p_o refers to the probability of seeing *any* new data. Probability to see a specific unknown item is much smaller, $p_o / \text{all_unknown_items}$ and use the assumption that all unknown events occur with equal probability
- for the words with the highest number of occurrences, use the actual probability (no smoothing)
- for the words for which n_{r+1} is 0, go to the next rank n_{r+2}

Back-off Methods

Notice that:

N-grams are more precise than (N-1)grams (remember the Shakespeare example)

But also, N-grams are more sparse than (N-1) grams

How to combine things?

Attempt N-grams and back-off to (N-1) if counts are not available

E.g. attempt prediction using 4-grams, and back-off to trigrams (or bigrams, or unigrams) if counts are not available

Backoff

In **backoff** model we build an N-gram model based on (N-1)-gram model. We “back off” to a lower order N-gram if we have zero evidence for a higher-order N-gram.

The trigram version of backoff might be represented as follows:

$$\hat{P}(w_i | w_{i-2} w_{i-1}) = P(w_i | w_{i-2} w_{i-1}), \quad \text{if } c(w_{i-2} w_{i-1} w_i) > 0$$

$$\hat{P}(w_i | w_{i-2} w_{i-1}) = \alpha_1 P(w_i | w_{i-1}), \quad \text{if } c(w_{i-2} w_{i-1} w_i) = 0 \ \& \ c(w_{i-1} w_i) > 0$$

$$\hat{P}(w_i | w_{i-2} w_{i-1}) = \alpha_2 P(w_i), \quad \text{if otherwise}$$

Interpolation

- ❖ Linearly combine estimates of N-gram models of increasing order.

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

$$\text{Where: } \sum_i \lambda_i = 1$$

- Learn proper values for λ_i by training to (approximately) maximize the likelihood of an independent *development* (a.k.a. *tuning*) corpus.

Kneser–Ney smoothing

However, in Kneser–Ney smoothing, the lower level probability is a smoothed probability calculated not by computing the raw probability of the word following the context, but by computing the number of different contexts that the word follows in the lower order model. The modified Kneser–Ney algorithm is further extended by using three discounting parameters instead of the single parameter used in standard Kneser–Ney smoothing and absolute discounting.

-
- Better estimate for probabilities of lower-order unigrams!
 - Shannon game: *I can't see without my reading* Fransisco ?
 - “Francisco” is more common than “glasses”
 - ... but “Francisco” always follows “San”
 - The unigram is useful exactly when we haven't seen this bigram!
 - Instead of $P(w)$: “How likely is w ”
 - $P_{\text{continuation}}(w)$: “How likely is w to appear as a novel continuation?”
 - For each word, count the number of bigram types it completes
 - Every bigram type was a novel continuation the first time it was seen

$$P_{\text{CONTINUATION}}(w) \propto |\{w_{i-1} : c(w_{i-1}, w) > 0\}|$$

-
- Alternative metaphor: The number of # of word types seen to precede w

$$|\{w_{i-1} : c(w_{i-1}, w) > 0\}|$$

- normalized by the # of words preceding all words:

$$P_{CONTINUATION}(w) = \frac{|\{w_{i-1} : c(w_{i-1}, w) > 0\}|}{\sum_{w'} |\{w'_{i-1} : c(w'_{i-1}, w') > 0\}|}$$

- A frequent word (Francisco) occurring in only one context (San) will have a low continuation probability

$$P_{KN}(w_i | w_{i-1}) = \frac{\max(c(w_{i-1}, w_i) - d, 0)}{c(w_{i-1})} + \lambda(w_{i-1})P_{CONTINUATION}(w_i)$$

λ is a normalizing constant; the probability mass we've discounted

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

the normalized discount

The number of word types that can follow w_{i-1}
 = # of word types we discounted
 = # of times we applied normalized discount

Kneser-Ney Smoothing: Recursive formulation

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

$$c_{KN}(\bullet) = \begin{cases} \text{count}(\bullet) & \text{for the highest order} \\ \text{continuationcount}(\bullet) & \text{for lower order} \end{cases}$$

Continuation count = Number of unique single word contexts for •

Smoothing for Webscale Ngrams

- “Stupid backoff” (Brants *et al.* 2007)
- No discounting, just use relative frequencies

$$S(w_i | w_{i-k+1}^{i-1}) = \begin{cases} \frac{\text{count}(w_{i-k+1}^i)}{\text{count}(w_{i-k+1}^{i-1})} & \text{if } \text{count}(w_{i-k+1}^i) > 0 \\ 0.4S(w_i | w_{i-k+2}^{i-1}) & \text{otherwise} \end{cases}$$

$$S(w_i) = \frac{\text{count}(w_i)}{N}$$

-
- How to deal with, e.g., Google N-gram corpus
 - Pruning
 - Only store N-grams with count $>$ threshold.
 - Remove singletons of higher-order n-grams
 - Entropy-based pruning
 - Efficiency
 - Efficient data structures like tries
 - Bloom filters: approximate language models
 - Store words as indexes, not strings
 - Use Huffman coding to fit large numbers of words into two bytes
 - Quantize probabilities (4-8 bits instead of 8-byte float)

Evaluation of Language Models

Perplexity is a per –word average of the probability with which the language model generates the (previously unseen) test data set , where the average is over the no of words in the test data set.

A very common measure is the perplexity of the model on unseen (held -out) data, the test data.

There are m sentences (s_m) in the test data. Then perplexity is defined as the negative of this raised to the power of 2:

$$\text{Perplexity} = 2^{-\frac{1}{M} \sum_{i=1}^m \log p(s_i)}$$

We take the negative of the average log probability, and raise it to the power of 2. Perplexity is thus a positive number. The *smaller the perplexity, the better the language model is at modeling unseen data*.

-
- For example, a digit recognizer has a vocabulary size of 10 and any digit is equally likely to follow any other digit. Therefore, if we evaluate over a word sequence of length 1000, for each word, $P(w_3 | w_1, w_2) = 0.10$

$$H(W) = -\frac{1}{1000} \sum_{i=1}^{1000} \log_2(.10) = -\log_2(.10) = 3.322$$

$$PP(W) = 2^{H(W)} = 2^{3.322} = 10$$

- If average $P(w_3 | w_1, w_2)$ increases to .20 due to some structure in the sequence of digits, $H(W) = 2.322$ and $PP(W) = 5$

Perplexity: Is lower better?

Remarkable fact: the true model for data has the lowest possible perplexity

Lower the perplexity, the closer we are to true model.

Typically, perplexity correlates well with speech recognition word error rate

Correlates better when both models are trained on same data

Doesn't correlate well when training data changes

Skip - Grams

skip-grams are a generalization of n-grams in which the components (typically words) need not be consecutive in the text under consideration, but may leave gaps that are skipped over.

They provide one way of overcoming the data sparsity problem found with conventional n-gram analysis.

“Insurgents killed in ongoing fighting.”

Bi-grams = {insurgents killed, killed in, in ongoing, ongoing fighting}.

2-skip-bi-grams = {insurgents killed, insurgents in, insurgents ongoing, killed in, killed ongoing, killed fighting, in ongoing, in fighting, ongoing fighting}

Tri-grams = {insurgents killed in, killed in ongoing, in ongoing fighting}.

2-skip-tri-grams = {insurgents killed in, insurgents killed ongoing, insurgents killed fighting, insurgents in ongoing, insurgents in fighting, insurgents ongoing fighting, killed in ongoing, killed in fighting, killed ongoing fighting, in ongoing fighting}.

skip-grams have been used as an alternative to increasing the size of training data

skip-grams can be surprisingly helpful when test documents are similar to training documents.

when training data is limited better go for skip-grams to expand contextual information

The disadvantage of skip-gram modeling is the sheer size of the training model that can be produced.

N-gram Pruning

count cutoff is widely used to prune language models.

The cutoff method deletes from the LM those n-grams that occur infrequently in the training data.

The cutoff method assumes that if an n-gram is infrequent in training data, it is also infrequent in testing data.

Due to the memory limitation in realistic applications, only a finite set of word pairs have conditional probabilities $P(w_n | w_{n-1})$ explicitly represented in the model, especially when the model is trained on a large corpus. The remaining word pairs are assigned a probability by back-off (i.e. unigram estimates).

The goal of bigram pruning is to remove uncommon explicit bigram estimates $P(w_n | w_{n-1})$ from the model to reduce the number of parameters, while minimizing the performance loss.

Let $p(\cdot|\cdot)$ denote the conditional probabilities assigned by the original model, and $p'(\cdot|\cdot)$ the probabilities in the pruned model. Then, the relative entropy between the two models is

$$D(p||p') = - \sum_{w_i, h_j} p(w_i, h_j) [\log p'(w_i|h_j) - \log p(w_i|h_j)]$$

where the summation is over all words w_i and histories (contexts) h_j .

To choose pruning thresholds, it is helpful to look at a more intuitive interpretation of $D(p||p')$ in terms of perplexity

$$\frac{PP' - PP}{PP} = e^{D(p||p')} - 1$$

$$PP = e^{-\sum_{h,w} p(h,w) \log p(w|h)},$$

$$PP' = e^{-\sum_{h,w} p(h,w) \log p'(w|h)}$$

Final Pruning Algorithm

A simple thresholding algorithm for N-gram pruning:

- Select a threshold τ .
- Compute the relative perplexity increase due to pruning each N-gram individually.
- Remove all N-grams that raise the perplexity by less than τ , and recompute backoff weights.

Drawbacks

N-gram language models fail to model long distance dependencies by definition

- Missing syntactic information
 - The students who participated in the game are tired
 - The student who participated in the game is tired
- Missing semantic information
 - The pizza that I had last night was tasty
 - The class that I had last night was interesting