Co-occurrence vectors

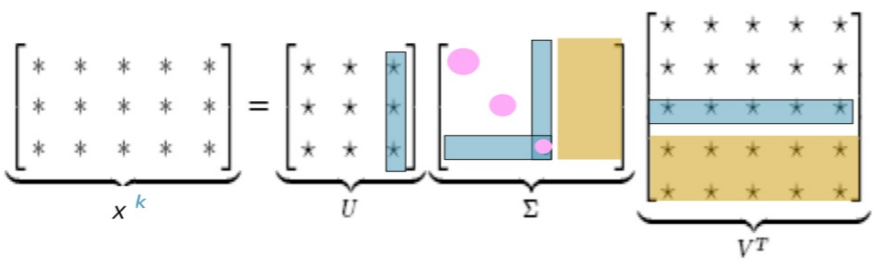
- Simple count co-occurrence vectors
  - Vectors increase in size with vocabulary
  - Very high dimensional: require a lot of storage (though sparse)
  - Subsequent classification models have sparsity issues → Models are less robust

- Low-dimensional vectors
  - Idea: store "most" of the important information in a fixed, small number of dimensions: a dense vector
  - Usually 25–1000 dimensions, similar to word2vec
  - How to reduce the dimensionality?

Classic Method: Dimensionality Reduction on X (HW1)

Singular Value Decomposition of co-occurrence matrix X

Factorizes X into $U\Sigma V^T$, where U and V are orthonormal (unit vectors and orthogonal)



Retain only k singular values, in order to generalize.
$\hat{X}$ is the best rank k approximation to X, in terms of least squares.
Classic linear algebra result. Expensive to compute for large matrices.

Hacks to X (several used in Rohde et al. 2005 in COALS)

- Running an SVD on raw counts doesn't work well!!!

- Scaling the counts in the cells can help ***a lot***
  - Problem: function words (*the, he, has*) are too frequent → syntax has too much impact. Some fixes:
    - log the frequencies
    - min(X, t), with $t \approx 100$
    - Ignore the function words
- Ramped windows that count closer words more than further away words
- Use Pearson correlations instead of counts, then set negative values to 0
- Etc.

Problems with SVD

Computational cost scales quadratically for $n \times m$ matrix:
$O(mn^2)$ flops (when $n < m$)

→ Bad for millions of words or documents.

→ Hard to incorporate new word or documents.

Combining the best from word2vec & SVD

word2vec → Window based

SVD → Matrix factorization

GloVe (Global Vectors for Word Representation)

GloVe (Global vectors for word Embedding)

3. Dense word vectors learned through word2vec or GloVe have many advantages over using sparse one-hot word vectors. Which of the following is a NOT advantage dense vectors have over sparse vectors?

a) Models using dense word vectors generalize better to unseen words than those using sparse vectors.

b) Models using dense word vectors generalize better to rare words than those using sparse vectors.

c) Dense word vectors encode similarity between words while sparse vectors do not.

d) Dense word vectors are easier to include as features in machine learning systems than sparse vectors.

Answer: (a) Models using dense word vectors generalize better to unseen words than those using sparse vectors.

Just like sparse representations, word2vec or GloVe do not have representations for unseen words and hence do not help in generalization.

Table 1: Co-occurrence probabilities for target words *ice* and *steam* with selected context words from a 6 billion token corpus. Only in the ratio does noise from non-discriminative words like *water* and *fashion* cancel out, so that large values (much greater than 1) correlate well with properties specific to ice, and small values (much less than 1) correlate well with properties specific of steam.

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Linear structure" in this context usually just refers to the addition and scalar multiplication of the vector space.

context of word i.

We begin with a simple example that showcases how certain aspects of meaning can be extracted directly from co-occurrence probabilities. Consider two words i and j that exhibit a particular aspect of interest; for concreteness, suppose we are interested in the concept of thermodynamic phase, for which we might take $i = ice$ and $j = steam$. The relationship of these words can be examined by studying the ratio of their co-occurrence probabilities with various probe words, k. For words k related to ice but not steam, say $k = solid$, we expect the ratio P_{ik}/P_{jk} will be large. Similarly, for words k related to steam but not ice, say $k = gas$, the ratio should be small. For words k like *water* or *fashion*, that are either related to both ice and steam, or to neither, the ratio should be close to one. Table 1 shows these probabilities and their ratios for a large corpus, and the numbers confirm these expectations. Compared to the raw probabilities, the ratio is better able to distinguish relevant words (*solid* and *gas*) from irrelevant words (*water* and *fashion*) and it is also better able to discriminate between the two relevant words.

The above argument suggests that the appropriate starting point for word vector learning should be with ratios of co-occurrence probabilities rather than the probabilities themselves. Noting that the ratio P_{ik}/P_{jk} depends on three words i, j, and k, the most general model takes the form,

$$F(w_i, w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}, \tag{1}$$

where $w \in \mathbb{R}^d$ are word vectors and $\tilde{w} \in \mathbb{R}^d$ are separate context word vectors whose role will be discussed in Section 4.2. In this equation, the right-hand side is extracted from the corpus, and F may depend on some as-of-yet unspecified parameters. The number of possibilities for F is vast, but by enforcing a few desiderata we can select a unique choice. First, we would like F to encode the information present the ratio P_{ik}/P_{jk} in the word vector space. Since vector spaces are inherently linear structures, the most natural way to do this is with vector differences. With this aim, we can restrict our consideration to those functions F that depend only on the difference of the two target words, modifying Eqn. (1) to,

$$F(w_i - w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}. \tag{2}$$

Next, we note that the arguments of F in Eqn. (2) are vectors while the right-hand side is a scalar. While F could be taken to be a complicated function parameterized by, e.g., a neural network, doing so would obfuscate the linear structure we are trying to capture. To avoid this issue, we can first take the dot product of the arguments,

$$F\left((w_i - w_j)^T \tilde{w}_k\right) = \frac{P_{ik}}{P_{jk}}, \tag{3}$$

which prevents F from mixing the vector dimensions in undesirable ways. Next, note that for word-word co-occurrence matrices, the distinction between a word and a context word is arbitrary and that we are free to exchange the two roles. To do so consistently, we must not only exchange $w \leftrightarrow \tilde{w}$ but also $X \leftrightarrow X^T$. Our final model should be invariant under this relabeling, but Eqn. (3) is not. However, the symmetry can be restored in two steps. First, we require that F be a homomorphism between the groups $(\mathbb{R}, +)$ and $(\mathbb{R}_{>0}, \times)$, i.e.,

$$F\left((w_i - w_j)^T \tilde{w}_k\right) = \frac{F(w_i^T \tilde{w}_k)}{F(w_j^T \tilde{w}_k)}, \tag{4}$$

which, by Eqn. (3), is solved by,

$$F(w_i^T \tilde{w}_k) = P_{ik} = \frac{X_{ik}}{X_i}. \tag{5}$$

The solution to Eqn. (4) is $F = \exp$, or,

$$w_i^T \tilde{w}_k = \log(P_{ik}) = \log(X_{ik}) - \log(X_i). \tag{6}$$

Next, if we assume F has a certain property (i.e. homomorphism between additive group and the multiplicative group) which gives,
F(w_i* u_k — w_j* u_k) = F(w_i* u_k)/F(w_j* u_k) = P_ik/P_jk
In other words this particular homomorphism ensures that the subtraction F(A-B) can also be represented as a division F(A)/F(B) and get the same result.

Next, we note that Eqn. (6) would exhibit the exchange symmetry if not for the $\log(X_i)$ on the right-hand side. However, this term is independent of k so it can be absorbed into a bias b_i for w_i. Finally, adding an additional bias $\tilde{b}_k$ for $\tilde{w}_k$ restores the symmetry,

$$w_i^T \tilde{w}_k + b_i + \tilde{b}_k = \log(X_{ik}) \,. \tag{7}$$

Eqn. (7) is a drastic simplification over Eqn. (1), but it is actually ill-defined since the logarithm diverges whenever its argument is zero. One resolution to this issue is to include an additive shift in the logarithm, $\log(X_{ik}) \rightarrow \log(1 + X_{ik})$, which maintains the sparsity of X while avoiding the divergences. The idea of factorizing the log of the co-occurrence matrix is closely related to LSA and we will use the resulting model as a baseline in our experiments. A main drawback to this model is that it weighs all co-occurrences equally, even those that happen rarely or never. Such rare co-occurrences are noisy and carry less information than the more frequent ones — yet even just the zero entries account for 75–95% of the data in X, depending on the vocabulary size and corpus.

We propose a new weighted least squares regression model that addresses these problems. Casting Eqn. (7) as a least squares problem and introducing a weighting function $f(X_{ij})$ into the cost function gives us the model

$$J = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2 \,, \tag{8}$$

where V is the size of the vocabulary. The weighting function should obey the following properties:

1. $f(0) = 0$. If f is viewed as a continuous function, it should vanish as $x \rightarrow 0$ fast enough that the $\lim_{x \rightarrow 0} f(x) \log^2 x$ is finite.
2. $f(x)$ should be non-decreasing so that rare co-occurrences are not overweighted.
3. $f(x)$ should be relatively small for large values of x, so that frequent co-occurrences are not overweighted.

Of course a large number of functions satisfy these properties, but one class of functions that we found to work well can be parameterized as,

$$f(x) = \begin{cases} (x/x_{\max})^\alpha & \text{if } x < x_{\max} \\ 1 & \text{otherwise} \end{cases} \,. \tag{9}$$

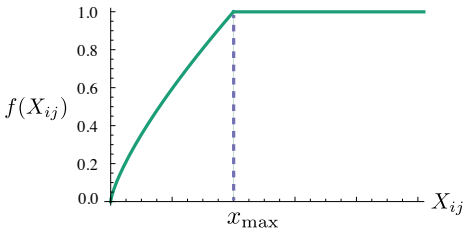


Figure 1: Weighting function f with $\alpha = 3/4$.

The performance of the model depends weakly on the cutoff, which we fix to $x_{\max} = 100$ for all our experiments. We found that $\alpha = 3/4$ gives a modest improvement over a linear version with $\alpha = 1$. Although we offer only empirical motivation for choosing the value 3/4, it is interesting that a similar fractional power scaling was found to give the best performance in (Mikolov et al., 2013a).

3.1 Relationship to Other Models

Because all unsupervised methods for learning word vectors are ultimately based on the occurrence statistics of a corpus, there should be commonalities between the models. Nevertheless, certain models remain somewhat opaque in this regard, particularly the recent window-based methods like skip-gram and ivLBL. Therefore, in this subsection we show how these models are related to our proposed model, as defined in Eqn. (8).

The starting point for the skip-gram or ivLBL methods is a model Q_{ij} for the probability that word j appears in the context of word i. For concreteness, let us assume that Q_{ij} is a softmax,

$$Q_{ij} = \frac{\exp(w_i^T \tilde{w}_j)}{\sum_{k=1}^{V} \exp(w_i^T \tilde{w}_k)} \,. \tag{10}$$

Most of the details of these models are irrelevant for our purposes, aside from the the fact that they attempt to maximize the log probability as a context window scans over the corpus. Training proceeds in an on-line, stochastic fashion, but the implied global objective function can be written as,

$$J = -\sum_{\substack{i \in \text{corpus} \\ j \in \text{context}(i)}} \log Q_{ij} \,. \tag{11}$$

Evaluating the normalization factor of the softmax for each term in this sum is costly. To allow for efficient training, the skip-gram and ivLBL models introduce approximations to Q_{ij}. However, the sum in Eqn. (11) can be evaluated much

more efficiently if we first group together those terms that have the same values for i and j,

$$J = -\sum_{i=1}^{V}\sum_{j=1}^{V} X_{ij} \log Q_{ij}\,, \tag{12}$$

where we have used the fact that the number of like terms is given by the co-occurrence matrix X.

Recalling our notation for $X_i = \sum_k X_{ik}$ and $P_{ij} = X_{ij}/X_i$, we can rewrite J as,

$$J = -\sum_{i=1}^{V} X_i \sum_{j=1}^{V} P_{ij} \log Q_{ij} = \sum_{i=1}^{V} X_i H(P_i, Q_i)\,, \tag{13}$$

where $H(P_i, Q_i)$ is the cross entropy of the distributions P_i and Q_i, which we define in analogy to X_i. As a weighted sum of cross-entropy error, this objective bears some formal resemblance to the weighted least squares objective of Eqn. (8). In fact, it is possible to optimize Eqn. (13) directly as opposed to the on-line training methods used in the skip-gram and ivLBL models. One could interpret this objective as a "global skip-gram" model, and it might be interesting to investigate further. On the other hand, Eqn. (13) exhibits a number of undesirable properties that ought to be addressed before adopting it as a model for learning word vectors.

To begin, cross entropy error is just one among many possible distance measures between probability distributions, and it has the unfortunate property that distributions with long tails are often modeled poorly with too much weight given to the unlikely events. Furthermore, for the measure to be bounded it requires that the model distribution Q be properly normalized. This presents a computational bottleneck owing to the sum over the whole vocabulary in Eqn. (10), and it would be desirable to consider a different distance measure that did not require this property of Q. A natural choice would be a least squares objective in which normalization factors in Q and P are discarded,

$$\hat{J} = \sum_{i,j} X_i (\hat{P}_{ij} - \hat{Q}_{ij})^2 \tag{14}$$

where $\hat{P}_{ij} = X_{ij}$ and $\hat{Q}_{ij} = \exp(w_i^T \tilde{w}_j)$ are the unnormalized distributions. At this stage another problem emerges, namely that X_{ij} often takes very large values, which can complicate the optimization. An effective remedy is to minimize the squared error of the logarithms of $\hat{P}$ and $\hat{Q}$ instead,

$$\begin{aligned}\hat{J} &= \sum_{i,j} X_i (\log \hat{P}_{ij} - \log \hat{Q}_{ij})^2 \\ &= \sum_{i,j} X_i (w_i^T \tilde{w}_j - \log X_{ij})^2\,. \end{aligned} \tag{15}$$

Finally, we observe that while the weighting factor X_i is preordained by the on-line training method inherent to the skip-gram and ivLBL models, it is by no means guaranteed to be optimal. In fact, Mikolov et al. (2013a) observe that performance can be increased by filtering the data so as to reduce the effective value of the weighting factor for frequent words. With this in mind, we introduce a more general weighting function, which we are free to take to depend on the context word as well. The result is,

$$\hat{J} = \sum_{i,j} f(X_{ij}) (w_i^T \tilde{w}_j - \log X_{ij})^2\,, \tag{16}$$

which is equivalent[1] to the cost function of Eqn. (8), which we derived previously.

3.2 Complexity of the model

As can be seen from Eqn. (8) and the explicit form of the weighting function $f(X)$, the computational complexity of the model depends on the number of nonzero elements in the matrix X. As this number is always less than the total number of entries of the matrix, the model scales no worse than $O(|V|^2)$. At first glance this might seem like a substantial improvement over the shallow window-based approaches, which scale with the corpus size, $|C|$. However, typical vocabularies have hundreds of thousands of words, so that $|V|^2$ can be in the hundreds of billions, which is actually much larger than most corpora. For this reason it is important to determine whether a tighter bound can be placed on the number of nonzero elements of X.

In order to make any concrete statements about the number of nonzero elements in X, it is necessary to make some assumptions about the distribution of word co-occurrences. In particular, we will assume that the number of co-occurrences of word i with word j, X_{ij}, can be modeled as a power-law function of the frequency rank of that word pair, r_{ij}:

$$X_{ij} = \frac{k}{(r_{ij})^\alpha}\,. \tag{17}$$

[1] We could also include bias terms in Eqn. (16).

Congratulations! You passed!
TO PASS 80% or higher

Keep Learning

GRADE
100%

Natural Language Processing & Word Embeddings

LATEST SUBMISSION GRADE
100%

1. Suppose you learn a word embedding for a vocabulary of 10000 words. Then the embedding vectors should be 10000 dimensional, so as to capture the full range of variation and meaning in those words. **1 / 1 point**

- ○ True
- ◉ False

Correct
The dimension of word vectors is usually smaller than the size of the vocabulary. Most common sizes for word vectors ranges between 50 and 400.

2. What is t-SNE? **1 / 1 point**

- ○ A linear transformation that allows us to solve analogies on word vectors
- ◉ A non-linear dimensionality reduction technique
- ○ A supervised learning algorithm for learning word embeddings
- ○ An open-source sequence modeling library

Correct
Yes

3. Suppose you download a pre-trained word embedding which has been trained on a huge corpus of text. You then use this word embedding to train an RNN for a language task of recognizing if someone is happy from a short snippet of text, using a small training set. **1 / 1 point**

x (input text)	y (happy?)
I'm feeling wonderful today!	1
I'm bummed my cat is ill.	0
Really enjoying this!	1

Then even if the word "ecstatic" does not appear in your small training set, your RNN might reasonably be expected to recognize "I'm ecstatic" as deserving a label $y = 1$.

- ◉ True
- ○ False

Correct
Yes, word vectors empower your model with an incredible ability to generalize. The vector for "ecstatic would contain a positive/happy connotation which will probably make your model classified the sentence as a "1".

4. Which of these equations do you think should hold for a good word embedding? (Check all that apply) **1 / 1 point**

- ☑ $e_{boy} - e_{girl} \approx e_{brother} - e_{sister}$

Correct
Yes!

- ☐ $e_{boy} - e_{girl} \approx e_{sister} - e_{brother}$
- ☑ $e_{boy} - e_{brother} \approx e_{girl} - e_{sister}$

Correct
Yes!

- ☐ $e_{boy} - e_{brother} \approx e_{sister} - e_{girl}$

5. Let E be an embedding matrix, and let o_{1234} be a one-hot vector corresponding to word 1234. Then to get the embedding of word 1234, why don't we call $E * o_{1234}$ in Python? **1 / 1 point**

- ◉ It is computationally wasteful.
- ○ The correct formula is $E^T * o_{1234}$.

- This doesn't handle unknown words (<UNK>).
- None of the above: calling the Python snippet as described above is fine.

Correct
Yes, the element-wise multiplication will be extremely inefficient.

6. When learning word embeddings, we create an artificial task of estimating $P(target \mid context)$. It is okay if we do poorly on this artificial prediction task; the more important by-product of this task is that we learn a useful set of word embeddings. **1 / 1 point**

- (•) True
- ( ) False

Correct

7. In the word2vec algorithm, you estimate $P(t \mid c)$, where t is the target word and c is a context word. How are t and c chosen from the training set? Pick the best answer. **1 / 1 point**

- ( ) c is the sequence of all the words in the sentence before t.
- ( ) c is the one word that comes immediately before t.
- ( ) c is a sequence of several words immediately before t.
- (•) c and t are chosen to be nearby words.

Correct

8. Suppose you have a 10000 word vocabulary, and are learning 500-dimensional word embeddings. The word2vec model uses the following softmax function: **1 / 1 point**

$$P(t \mid c) = \frac{e^{\theta_t^T e_c}}{\sum_{t'=1}^{10000} e^{\theta_{t'}^T e_c}}$$

Which of these statements are correct? Check all that apply.

- [x] θ_t and e_c are both 500 dimensional vectors.

Correct

- [ ] θ_t and e_c are both 10000 dimensional vectors.
- [x] θ_t and e_c are both trained with an optimization algorithm such as Adam or gradient descent.

Correct

- [ ] After training, we should expect θ_t to be very close to e_c when t and c are the same word.

9. Suppose you have a 10000 word vocabulary, and are learning 500-dimensional word embeddings.The GloVe model minimizes this objective: **1 / 1 point**

$$\min \sum_{i=1}^{10,000} \sum_{j=1}^{10,000} f(X_{ij})(\theta_i^T e_j + b_i + b_j' - log X_{ij})^2$$

Which of these statements are correct? Check all that apply.

- [ ] θ_i and e_j should be initialized to 0 at the beginning of training.
- [x] θ_i and e_j should be initialized randomly at the beginning of training.

Correct

- [x] X_{ij} is the number of times word j appears in the context of word i.

Correct

- [x] The weighting function $f(.)$ must satisfy $f(0) = 0$.

Correct
The weighting function helps prevent learning only from extremely common word pairs. It is not necessary that it satisfies this function.

10. You have trained word embeddings using a text dataset of m_1 words. You are considering using these word embeddings for a language task, for which you have a separate labeled dataset of m_2 words. Keeping in mind that using word embeddings is a form of transfer learning, under which of these circumstance would you expect the word embeddings to be helpful? **1 / 1 point**

- (•) $m_1 >> m_2$

$m_1 << m_2$

Correct

Identifying and quantifying bias in word embeddings

- Assumption: The aspect of bias is known. E.g. gender
- Find the "gender" dimension
  - Collect explicit gender-based word pairs (f, m): (woman, man), (mother, father), (gal, guy), (girl, boy), (she, he)
  - Get the gender dimension as (f-m) [How?]
- Collect a set N of gender neutral words

- Compute the gender component g in elements from N
  - DirectBias = $(1/|N|)\sum_{w \in N}|\cos(w,g)|$

Identifying and quantifying bias in word embeddings

- How to capture indirect bias?
- Direct bias: component along gender dimension
- Indirect bias: Component along its perpendicular

- Need to find the component to the perpendicular of the "gender" dimension
- Component of vector a along vector b:
  - Scalar Component: $comp_b(a) = (a.b)/|b|$
  - Vector component: $comp_b(a).b$
- $w_g = (w.g)g$, $w_\perp = w - w_g$

- IndirectBias $B(w,v) = (w.v - (w_\perp - v_\perp)/(|w_\perp| - |v_\perp|)) / (w.v)$

$$J = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - log(X_{ij})\right)^2 + \lambda \cos(w_i, g) + \gamma \cos(\tilde{w}_j, g)$$

A simple technique for debiasing GloVE

Language Modeling

What is language modeling?

Language modeling (LM) analyzes bodies to text to provide a foundation for word prediction. These models use statistical and probabilistic techniques to determine the probability of a particular word sequence occurring in a sentence.

Language modeling is used in NLP techniques that generate written text as an output. Applications and programs with NLP-concepts rely on language models for tasks like audio-to-text conversion, sentiment analysis, speech recognition, and spelling corrections.

Language models work by determining word probabilities in an analyzed chunk of text data. Data is interpreted after being fed to a machine learning algorithm that looks for contextual rules in that given natural language (i.e. English, Japanese, Spanish).

The model then applies those rules to the input language tasks for generating predictions. It can even produce new sequences or sentences based on what it learned.

Language models are useful for both text classification and text generation. In text classification, we can use the language model's probability calculations to separate texts into different categories.

For example, if we trained a language model on spam email subject titles, the model would likely give the subject "CLICK HERE FOR FREE EASY MONEY" a relatively high probability of being spam.
In text generation, a language model completes a sentence by generating text based on the incomplete input sentence. This is the idea behind the autocomplete feature when texting on a phone or typing in a search engine. The model will give suggestions to complete the sentence based on the words it predicts with the highest probabilities.

Types of Language Models

There are two categories that Language Models fall under:

Statistical Language Models: These models use traditional statistical techniques like N-grams, Hidden Markov Models (HMM), and established linguistic rules to learn the probability distribution of words. Statistical Language Modeling involves the development of probabilistic models that can predict the next word in the sequence given the words that precede it.

Neural Language Models: These models are new players in the NLP world and have surpassed the statistical language models in their effectiveness. They use different kinds of Neural Networks to model language. The use of neural networks in the development of language models has become so popular that it is now the preferred approach for challenging tasks like speech recognition and machine translation.

Note: GPT-3 is an example of a Neural language model. BERT by Google is another popular Neural language model used in the algorithm of the search engine for next word prediction of our search query.

N-gram Language Models

The n-gram model is a probabilistic language model that can predict the next item in a sequence using the (n − 1)–order Markov model. Let's understand that better with an example. Consider the following sentence:

> "I love reading blogs on Educative to learn new concepts"

A 1-gram (or unigram) is a **one-word sequence**. For the above sentence, the unigrams would simply be: "I", "love", "reading", "blogs", "on", "Educative", "and", "learn", "new", "concepts".

A 2-gram (or bigram) is a **two-word sequence of words**, like "I love", "love reading", "on Educative" or "new concepts".

Lastly, a 3-gram (or trigram) is a **three-word sequence of words**, like "I love reading", "blogs on Educative", or "learn new concepts".

An N-gram language model predicts the probability of a given N-gram within any sequence of words in the language. If we have a good N-gram model, we can predict $p(w|h)$, or the probability of seeing the word w given a history of previous words h, where the history contains n-1 words.

> **Example:** "I love reading ___". Here, we want to predict what word will fill the dash based on the probabilities of the previous words.

We must estimate this probability to construct an N-gram model. We compute this probability in two steps:

1. Apply the chain rule of probability
2. We then apply a very strong simplification assumption to allow us to compute p(w1…ws) in an easy manner.

The chain rule of probability is:

$$p(w1...ws) = p(w1).p(w2|w1).p(w3|w1w2).p(w4|w1w2w3).....p(wn|w1...wn-1)$$

> **Definition:** What is the chain rule? It tells us how to compute the joint probability of a sequence by using the conditional probability of a word given previous words.

Here, we do not have access to these conditional probabilities with complex conditions of up to n-1 words. So, how do we proceed? This is where we introduce a **simplification assumption**. We can assume for all conditions, that:

$$p(wk|w1...wk-1) = p(wk|wk-1)$$

Here, we approximate the history (the context) of the word wk by looking only at the last word of the context. This assumption is called the Markov assumption. It is an example of the Bigram model. The same concept can be enhanced further for example for trigram model the formula will be:

$$p(wk|w1...wk-1) = p(wk|wk-2wk-1)$$

These models have a basic problem: they give the probability to zero if an unknown word is seen, so the concept of smoothing is used. In smoothing we assign some probability to the unseen words. There are different types of smoothing techniques such as Laplace smoothing, Good Turing, and Kneser-ney smoothing.

3.1 N-Grams

Let's begin with the task of computing $P(w|h)$, the probability of a word w given some history h. Suppose the history h is "*its water is so transparent that*" and we want to know the probability that the next word is *the*:

$$P(\textit{the}|\textit{its water is so transparent that}). \tag{3.1}$$

One way to estimate this probability is from relative frequency counts: take a very large corpus, count the number of times we see *its water is so transparent that*, and count the number of times this is followed by *the*. This would be answering the question "Out of the times we saw the history h, how many times was it followed by the word w", as follows:

$$P(\textit{the}|\textit{its water is so transparent that}) = \frac{C(\textit{its water is so transparent that the})}{C(\textit{its water is so transparent that})} \tag{3.2}$$

With a large enough corpus, such as the web, we can compute these counts and estimate the probability from Eq. 3.2. You should pause now, go to the web, and compute this estimate for yourself.

While this method of estimating probabilities directly from counts works fine in many cases, it turns out that even the web isn't big enough to give us good estimates in most cases. This is because language is creative; new sentences are created all the time, and we won't always be able to count entire sentences. Even simple extensions

of the example sentence may have counts of zero on the web (such as "*Walden Pond's water is so transparent that the*"; well, *used to* have counts of zero).

Similarly, if we wanted to know the joint probability of an entire sequence of words like *its water is so transparent*, we could do it by asking "out of all possible sequences of five words, how many of them are *its water is so transparent*?" We would have to get the count of *its water is so transparent* and divide by the sum of the counts of all possible five word sequences. That seems rather a lot to estimate!

For this reason, we'll need to introduce more clever ways of estimating the probability of a word w given a history h, or the probability of an entire word sequence W. Let's start with a little formalizing of notation. To represent the probability of a particular random variable X_i taking on the value "the", or $P(X_i = \text{"the"})$, we will use the simplification $P(the)$. We'll represent a sequence of n words either as $w_1 \dots w_n$ or $w_{1:n}$ (so the expression $w_{1:n-1}$ means the string $w_1, w_2, \dots, w_{n-1}$). For the joint probability of each word in a sequence having a particular value $P(X_1 = w_1, X_2 = w_2, X_3 = w_3, \dots, X_n = w_n)$ we'll use $P(w_1, w_2, \dots, w_n)$.

Now, how can we compute probabilities of entire sequences like $P(w_1, w_2, \dots, w_n)$? One thing we can do is decompose this probability using the **chain rule of probability**:

$$\begin{aligned} P(X_1 \dots X_n) &= P(X_1)P(X_2|X_1)P(X_3|X_{1:2}) \dots P(X_n|X_{1:n-1}) \\ &= \prod_{k=1}^{n} P(X_k|X_{1:k-1}) \end{aligned} \tag{3.3}$$

Applying the chain rule to words, we get

$$\begin{aligned} P(w_{1:n}) &= P(w_1)P(w_2|w_1)P(w_3|w_{1:2}) \dots P(w_n|w_{1:n-1}) \\ &= \prod_{k=1}^{n} P(w_k|w_{1:k-1}) \end{aligned} \tag{3.4}$$

The chain rule shows the link between computing the joint probability of a sequence and computing the conditional probability of a word given previous words. Equation 3.4 suggests that we could estimate the joint probability of an entire sequence of words by multiplying together a number of conditional probabilities. But using the chain rule doesn't really seem to help us! We don't know any way to compute the exact probability of a word given a long sequence of preceding words, $P(w_n|w_{1:n-1})$. As we said above, we can't just estimate by counting the number of times every word occurs following every long string, because language is creative and any particular context might have never occurred before!

The intuition of the n-gram model is that instead of computing the probability of a word given its entire history, we can **approximate** the history by just the last few words.

bigram

The **bigram** model, for example, approximates the probability of a word given all the previous words $P(w_n|w_{1:n-1})$ by using only the conditional probability of the preceding word $P(w_n|w_{n-1})$. In other words, instead of computing the probability

$$P(\text{the}|\text{Walden Pond's water is so transparent that}) \tag{3.5}$$

we approximate it with the probability

$$P(\text{the}|\text{that}) \tag{3.6}$$

When we use a bigram model to predict the conditional probability of the next word, we are thus making the following approximation:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-1}) \tag{3.7}$$

Markov

The assumption that the probability of a word depends only on the previous word is called a **Markov** assumption. Markov models are the class of probabilistic models that assume we can predict the probability of some future unit without looking too far into the past. We can generalize the bigram (which looks one word into the past) to the trigram (which looks two words into the past) and thus to the **n-gram** (which looks $n-1$ words into the past).

n-gram

Let's see a general equation for this n-gram approximation to the conditional probability of the next word in a sequence. We'll use N here to mean the n-gram size, so $N = 2$ means bigrams and $N = 3$ means trigrams. Then we approximate the probability of a word given its entire context as follows:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-N+1:n-1}) \tag{3.8}$$

Given the bigram assumption for the probability of an individual word, we can compute the probability of a complete word sequence by substituting Eq. 3.7 into Eq. 3.4:

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \tag{3.9}$$

maximum likelihood estimation

normalize

How do we estimate these bigram or n-gram probabilities? An intuitive way to estimate probabilities is called **maximum likelihood estimation** or **MLE**. We get the MLE estimate for the parameters of an n-gram model by getting counts from a corpus, and **normalizing** the counts so that they lie between 0 and 1.[1]

For example, to compute a particular bigram probability of a word w_n given a previous word w_{n-1}, we'll compute the count of the bigram $C(w_{n-1}w_n)$ and normalize by the sum of all the bigrams that share the same first word w_{n-1}:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)} \tag{3.10}$$

We can simplify this equation, since the sum of all bigram counts that start with a given word w_{n-1} must be equal to the unigram count for that word w_{n-1} (the reader should take a moment to be convinced of this):

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \tag{3.11}$$

Let's work through an example using a mini-corpus of three sentences. We'll first need to augment each sentence with a special symbol <s> at the beginning of the sentence, to give us the bigram context of the first word. We'll also need a special end-symbol. </s>[2]

```
<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>
```

[1] For probabilistic models, normalizing means dividing by some total count so that the resulting probabilities fall between 0 and 1.

[2] We need the end-symbol to make the bigram grammar a true probability distribution. Without an end-symbol, instead of the sentence probabilities of all sentences summing to one, the sentence probabilities for all sentences *of a given length* would sum to one. This model would define an infinite set of probability distributions, with one distribution per sentence length. See Exercise 3.5.

Here are the calculations for some of the bigram probabilities from this corpus

$P(\texttt{I}|\texttt{<s>}) = \frac{2}{3} = .67$ $P(\texttt{Sam}|\texttt{<s>}) = \frac{1}{3} = .33$ $P(\texttt{am}|\texttt{I}) = \frac{2}{3} = .67$

$P(\texttt{</s>}|\texttt{Sam}) = \frac{1}{2} = 0.5$ $P(\texttt{Sam}|\texttt{am}) = \frac{1}{2} = .5$ $P(\texttt{do}|\texttt{I}) = \frac{1}{3} = .33$

For the general case of MLE n-gram parameter estimation:

$$P(w_n|w_{n-N+1:n-1}) = \frac{C(w_{n-N+1:n-1}\ w_n)}{C(w_{n-N+1:n-1})} \tag{3.12}$$

relative frequency

Equation 3.12 (like Eq. 3.11) estimates the n-gram probability by dividing the observed frequency of a particular sequence by the observed frequency of a prefix. This ratio is called a **relative frequency**. We said above that this use of relative frequencies as a way to estimate probabilities is an example of maximum likelihood estimation or MLE. In MLE, the resulting parameter set maximizes the likelihood of the training set T given the model M (i.e., $P(T|M)$). For example, suppose the word *Chinese* occurs 400 times in a corpus of a million words like the Brown corpus. What is the probability that a random word selected from some other text of, say, a million words will be the word *Chinese*? The MLE of its probability is $\frac{400}{1000000}$ or .0004. Now .0004 is not the best possible estimate of the probability of *Chinese* occurring in all situations; it might turn out that in some other corpus or context *Chinese* is a very unlikely word. But it is the probability that makes it *most likely* that Chinese will occur 400 times in a million-word corpus. We present ways to modify the MLE estimates slightly to get better probability estimates in Section 3.5.

Matching genres and dialects is still not sufficient. Our models may still be subject to the problem of **sparsity**. For any n-gram that occurred a sufficient number of times, we might have a good estimate of its probability. But because any corpus is limited, some perfectly acceptable English word sequences are bound to be missing from it. That is, we'll have many cases of putative "zero probability n-grams" that should really have some non-zero probability. Consider the words that follow the bigram *denied the* in the WSJ Treebank3 corpus, together with their counts:

denied the allegations:	5
denied the speculation:	2
denied the rumors:	1
denied the report:	1

But suppose our test set has phrases like:

denied the offer
denied the loan

Our model will incorrectly estimate that the $P(\text{offer}|\text{denied the})$ is 0!

These **zeros**—things that don't ever occur in the training set but do occur in the test set—are a problem for two reasons. First, their presence means we are underestimating the probability of all sorts of words that might occur, which will hurt the performance of any application we want to run on this data.

Second, if the probability of any word in the test set is 0, the entire probability of the test set is 0. By definition, perplexity is based on the inverse probability of the

given 3 sentences

"JOHN READ MOBY DICK"

"MARY READ A DIFFERENT BOOK"

"SHE READ A BOOK BY CHER"

Let's Calculate

$$P(\text{JOHN READ A BOOK})$$

$$P(\text{JOHN} \mid \langle BOS \rangle) = \frac{c(\langle BOS \rangle \text{ JOHN})}{\sum_w c(\langle BOS \rangle\ w)} = \frac{1}{3}$$

$$P(\text{READ} \mid \text{JOHN}) = \frac{c(\text{JOHN READ})}{\sum_w c(\text{JOHN } w)} = \frac{1}{1}$$

$$P(\text{A} \mid \text{READ}) = \frac{c(\text{READ A})}{\sum_w c(\text{READ } w)} = \frac{2}{3}$$

$$P(\text{BOOK} \mid \text{A}) = \frac{c(\text{A BOOK})}{\sum_w c(\text{A } w)} = \frac{1}{2}$$

$$P(\langle EOS \rangle \mid \text{BOOK}) = \frac{c(\text{BOOK} \langle EOS \rangle)}{\sum_w c(\text{BOOK } w)} = \frac{1}{2}$$

$$P(\text{JOHN READ A BOOK}) = P(\text{JOHN} \mid \langle BOS \rangle)\, P(\text{READ} \mid \text{JOHN})\, P(\text{A} \mid \text{READ})\, P(\text{BOOK} \mid \text{A})\, P(\langle EOS \rangle \mid \text{BOOK})$$

$$= \frac{1}{3} \times 1 \times \frac{2}{3} \times \frac{1}{2} \times \frac{1}{2}$$

$$\approx 0.06$$

1.2 Smoothing

Now, consider the sentence CHER READ A BOOK. We have

$$p(\text{READ} \mid \text{CHER}) = \frac{c(\text{CHER READ})}{\sum_w c(\text{CHER } w)} = \frac{0}{1}$$

giving us $p(\text{CHER READ A BOOK}) = 0$. Obviously, this is an underestimate for the probability of CHER READ A BOOK as there is *some* probability that the sentence occurs. To show why it is important that this probability should be given a nonzero value, we turn to the primary application for language models, *speech recognition*. In speech recognition, one attempts to find the sentence s that maximizes $p(s|A) = \frac{p(A|s)p(s)}{p(A)}$ for a given acoustic signal A. If $p(s)$ is zero, then $p(s|A)$ will be zero and the string s will never be considered as a transcription, regardless of how unambiguous the acoustic signal is. Thus, whenever a string s such that $p(s) = 0$ occurs during a speech recognition task, an error will be made. Assigning all strings a nonzero probability helps prevent errors in speech recognition.

Smoothing is used to address this problem. The term *smoothing* describes techniques for adjusting the maximum likelihood estimate of probabilities (as in equations (2) and (4)) to produce more accurate probabilities. The name *smoothing* comes from the fact that these techniques tend to make distributions more uniform, by adjusting low probabilities such as zero probabilities upward, and high probabilities downward. Not only do smoothing methods generally prevent zero probabilities, but they also attempt to improve the accuracy of the model as a whole. Whenever a probability is estimated from few counts, smoothing has the potential to significantly improve estimation.

To give an example, one simple smoothing technique is to pretend each bigram occurs once more than it actually does (Lidstone, 1920; Johnson, 1932; Jeffreys, 1948), yielding

$$p(w_i|w_{i-1}) = \frac{1 + c(w_{i-1}w_i)}{\sum_{w_i}[1 + c(w_{i-1}w_i)]} = \frac{1 + c(w_{i-1}w_i)}{|V| + \sum_{w_i} c(w_{i-1}w_i)} \quad (5)$$

Smoothing

What do we do with words that are in our vocabulary (they are not unknown words) but appear in a test set in an unseen context (for example they appear after a word they never appeared after in training)? To keep a language model from assigning zero probability to these unseen events, we'll have to shave off a bit of probability mass from some more frequent events and give it to the events we've never seen. This modification is called smoothing or discounting. In this section and the fol- lowing ones we'll introduce a variety of ways to do smoothing: Laplace (add-one) smoothing, add-k smoothing, stupid backoff, and Kneser-Ney smoothing.

Laplace Smoothing:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})} \tag{3.23}$$

For add-one smoothed bigram counts, we need to augment the unigram count by the number of total word types in the vocabulary V:

$$P_{\text{Laplace}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+1}{\sum_w (C(w_{n-1}w)+1)} = \frac{C(w_{n-1}w_n)+1}{C(w_{n-1})+V} \tag{3.24}$$

Example: In the Berkeley Restaurant Project, we have the following counts:

$$C(\texttt{want}) = 927$$
$$C(\texttt{want to}) = 608$$
$$C(\texttt{want want}) = 0$$
$$|\mathcal{V}| = 1446$$

Estimate the probabilities for $P(\texttt{to}|\texttt{want})$ and $P(\texttt{want}|\texttt{want})$ with and without add-one smoothing.

$$P_{\text{MLE}}(\texttt{to}|\texttt{want}) = \frac{608}{927} = 0.6559$$
$$P_{\text{MLE}}(\texttt{want}|\texttt{want}) = \frac{0}{927} = 0$$
$$P_{+1}(\texttt{to}|\texttt{want}) = \frac{608+1}{927+1446} = 0.2566$$
$$P_{+1}(\texttt{want}|\texttt{want}) = \frac{1}{927+1446} = 0.0004$$

Problem: We often steal way too much! This is because the $|\mathcal{V}|$ in the denominator can completely overpower $C(w_{t-1})$. In the example above, we had a probability go from 0.6559 to 0.2566!

Disadvantage of add-one smoothing is

→ takes away too much probability mass from seen events.

→ Assign too much total probability mass to unseen events.

3.5.2 Add-k smoothing

One alternative to add-one smoothing is to move a bit less of the probability mass from the seen to the unseen events. Instead of adding 1 to each count, we add a fractional count k (.5? .05? .01?). This algorithm is therefore called **add-k smoothing**.

$$P^*_{\text{Add-k}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+k}{C(w_{n-1})+kV} \tag{3.26}$$

Add-k smoothing requires that we have a method for choosing k; this can be done, for example, by optimizing on a **devset**. Although add-k is useful for some tasks (including text classification), it turns out that it still doesn't work well for language modeling, generating counts with poor variances and often inappropriate discounts (Gale and Church, 1994).

3.5.3 Backoff and Interpolation

The discounting we have been discussing so far can help solve the problem of zero frequency n-grams. But there is an additional source of knowledge we can draw on. If we are trying to compute $P(w_n|w_{n-2}w_{n-1})$ but we have no examples of a particular trigram $w_{n-2}w_{n-1}w_n$, we can instead estimate its probability by using the bigram probability $P(w_n|w_{n-1})$. Similarly, if we don't have counts to compute $P(w_n|w_{n-1})$, we can look to the unigram $P(w_n)$.

In other words, sometimes using **less context** is a good thing, helping to generalize more for contexts that the model hasn't learned much about. There are two ways to use this n-gram "hierarchy". In **backoff**, we use the trigram if the evidence is sufficient, otherwise we use the bigram, otherwise the unigram. In other words, we only "back off" to a lower-order n-gram if we have zero evidence for a higher-order n-gram. By contrast, in **interpolation**, we always mix the probability estimates from all the n-gram estimators, weighting and combining the trigram, bigram, and unigram counts.

In simple linear interpolation, we combine different order n-grams by linearly interpolating them. Thus, we estimate the trigram probability $P(w_n|w_{n-2}w_{n-1})$ by mixing together the unigram, bigram, and trigram probabilities, each weighted by a

λ:

$$\begin{aligned}\hat{P}(w_n|w_{n-2}w_{n-1}) &= \lambda_1 P(w_n) \\ &\quad +\lambda_2 P(w_n|w_{n-1}) \\ &\quad +\lambda_3 P(w_n|w_{n-2}w_{n-1})\end{aligned} \tag{3.27}$$

The λs must sum to 1, making Eq. 3.27 equivalent to a weighted average:

$$\sum_i \lambda_i = 1 \tag{3.28}$$

Back-off

In back-off we use the highest-order model if the count is not zero, otherwise we back off to a lower-order model.

We need to discount the higher-order models in order to spread mass to the lower-order models. And then we need to make sure the probabilities sum to 1.

Back-off N-gram model:

$$P_{\text{BO}}(w_t|w_{t-N+1:t-1}) = \begin{cases} P_d(w_t|w_{t-N+1:t-1}) & \text{if } C(w_{t-N+1:t}) > 0 \\ \alpha(w_{t-N+1:t}) P_{\text{BO}}(w_t|w_{t-N+2:t-1}) & \text{if } C(w_{t-N+1:t}) = 0 \end{cases}$$

where

- $P_d(w_t|w_{t-N+1:t-1})$ is some discounted N-gram model.
- The back-off weights $\alpha(w_{t-N+1:t})$ are such that the probability sum to 1.

Good Turing smoothing:

<s> I am here </s>
<s> who am I </s>
<s> I would like </s>

Computing N_c

I	3	frequency of frequency
am	2	$N_1 = 4$
here	1	$N_2 = 1$
who	1	$N_3 = 1$
would	1	
like	1	

Idea:

→ Reallocating the probability mass of n-gram that occurs r+1 times in the training data to the n-grams that occurs r times.

→ In particular, reallocate the probability mass of n-grams that were seen once to the n-grams that were never seen.