CHAPTER 16 Semantics with Dense Vectors

In the previous chapter we saw how to represent a word as a sparse vector with dimensions corresponding to the words in the vocabulary, and whose values were some function of the count of the word co-occurring with each neighboring word. Each word is thus represented with a vector that is both **long** (length $|V|$, with vocabularies of 20,000 to 50,000) and **sparse**, with most elements of the vector for each word equal to zero.

In this chapter we turn to an alternative family of methods of representing a word: the use of vectors that are **short** (of length perhaps 50-1000) and **dense** (most values are non-zero).

Short vectors have a number of potential advantages. First, they are easier to include as features in machine learning systems; for example if we use 100-dimensional word embeddings as features, a classifier can just learn 100 weights to represent a function of word meaning, instead of having to learn tens of thousands of weights for each of the sparse dimensions. Because they contain fewer parameters than sparse vectors of explicit counts, dense vectors may generalize better and help avoid overfitting. And dense vectors may do a better job of capturing synonymy than sparse vectors. For example, *car* and *automobile* are synonyms; but in a typical sparse vectors representation, the *car* dimension and the *automobile* dimension are distinct dimensions. Because the relationship between these two dimensions is not modeled, sparse vectors may fail to capture the similarity between a word with *car* as a neighbor and a word with *automobile* as a neighbor.

We will introduce three methods of generating very dense, short vectors: (1) using dimensionality reduction methods like **SVD**, (2) using neural nets like the popular **skip-gram** or **CBOW** approaches. (3) a quite different approach based on neighboring words called **Brown clustering**.

16.1 Dense Vectors via SVD

SVD

Latent Semantic Analysis

We begin with a classic method for generating dense vectors: **singular value decomposition**, or **SVD**, first applied to the task of generating embeddings from term-document matrices by Deerwester et al. (1988) in a model called **Latent Semantic Indexing** or **Latent Semantic Analysis** (**LSA**).

Singular Value Decomposition (SVD) is a method for finding the most important dimensions of a data set, those dimensions along which the data varies the most. It can be applied to any rectangular matrix. SVD is part of a family of methods that can approximate an N-dimensional dataset using fewer dimensions, including **Principle Components Analysis (PCA)**, **Factor Analysis**, and so on.

In general, dimensionality reduction methods first rotate the axes of the original dataset into a new space. The new space is chosen so that the highest order dimension captures the most variance in the original dataset, the next dimension captures the next most variance, and so on. Fig. 16.1 shows a visualization. A set of points (vectors) in two dimensions is rotated so that the first new dimension captures the most variation in the data. In this new space, we can represent data with a smaller number of dimensions (for example using one dimension instead of two) and still capture much of the variation in the original data.

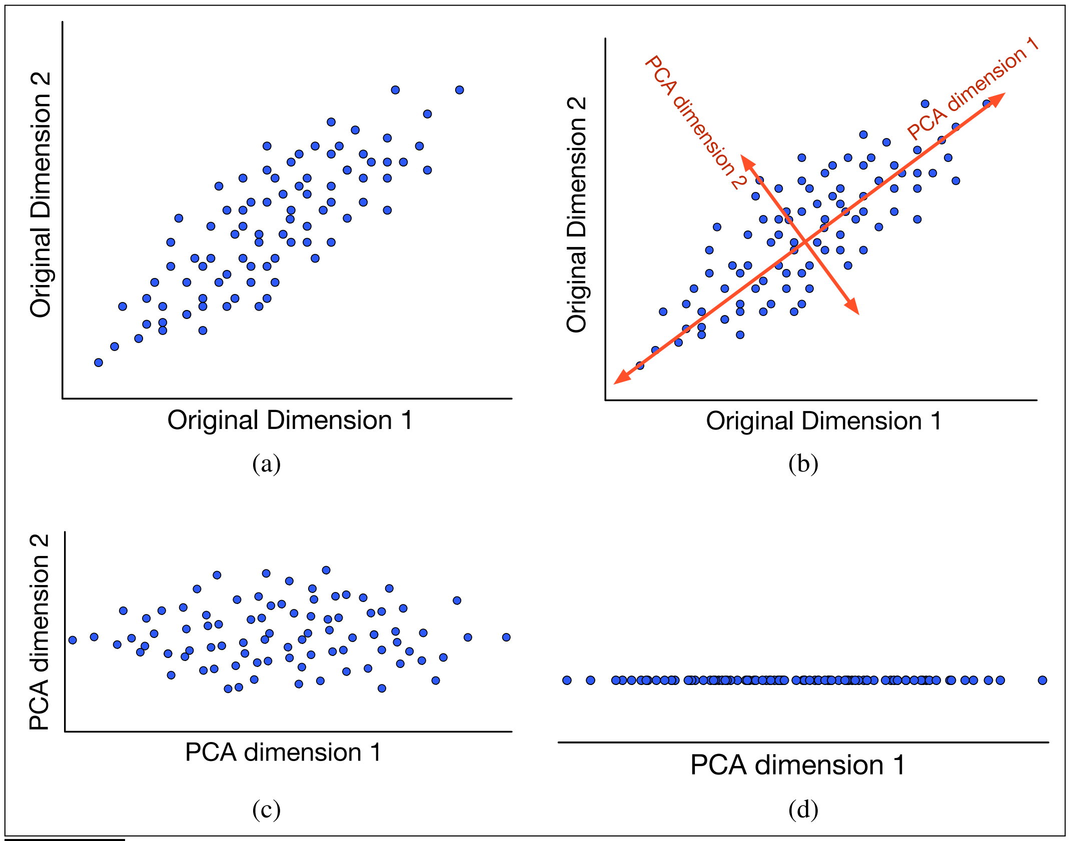

Figure 16.1 Visualizing principle components analysis: Given original data (a) find the rotation of the data (b) such that the first dimension captures the most variation, and the second dimension is the one orthogonal to the first that captures the next most variation. Use this new rotated space (c) to represent each point on a single dimension (d). While some information about the relationship between the original points is necessarily lost, the remaining dimension preserves the most that any one dimension could.

16.1.1 Latent Semantic Analysis

The use of SVD as a way to reduce large sparse vector spaces for word meaning, like the vector space model itself, was first applied in the context of information retrieval, briefly called **latent semantic indexing** (LSI) (Deerwester et al., 1988) but most frequently referred to as **LSA** (**latent semantic analysis**) (Deerwester et al., 1990).

LSA

LSA is a particular application of SVD to a $|V| \times c$ term-document matrix X representing $|V|$ words and their co-occurrence with c documents or contexts. SVD factorizes any such rectangular $|V| \times c$ matrix X into the product of three matrices W, Σ, and C^T. In the $|V| \times m$ matrix W, each of the w rows still represents a word, but the columns do not; each column now represents one of m dimensions in a latent space, such that the m column vectors are orthogonal to each other and the columns are ordered by the amount of variance in the original dataset each accounts for. The number of such dimensions m is the **rank** of X (the rank of a matrix is the number of linearly independent rows). Σ is a diagonal $m \times m$ matrix, with **singular values** along the diagonal, expressing the importance of each dimension. The $m \times c$ matrix C^T still represents documents or contexts, but each row now represents one of the new latent dimensions and the m row vectors are orthogonal to each other.

By using only the first k dimensions, of W, Σ, and C instead of all m dimensions, the product of these 3 matrices becomes a least-squares approximation to the original X. Since the first dimensions encode the most variance, one way to view the reconstruction is thus as modeling the most important information in the original dataset.

SVD applied to co-occurrence matrix X:

$$\underset{|V| \times c}{\begin{bmatrix} \\ \\ & X & \\ \\ \\ \end{bmatrix}} = \underset{|V| \times m}{\begin{bmatrix} \\ \\ & W & \\ \\ \\ \end{bmatrix}} \underset{m \times m}{\begin{bmatrix} \sigma_1 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \sigma_m \end{bmatrix}} \underset{m \times c}{\begin{bmatrix} \\ & C & \\ \\ \end{bmatrix}}$$

Taking only the top $k, k \leq m$ dimensions after the SVD is applied to the co-occurrence matrix X:

$$\underset{|V| \times c}{\begin{bmatrix} \\ \\ & X & \\ \\ \\ \end{bmatrix}} = \underset{|V| \times k}{\begin{bmatrix} \\ \\ & W_k & \\ \\ \\ \end{bmatrix}} \underset{k \times k}{\begin{bmatrix} \sigma_1 & 0 & 0 & \dots & 0 \\ 0 & \sigma_2 & 0 & \dots & 0 \\ 0 & 0 & \sigma_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \sigma_k \end{bmatrix}} \begin{bmatrix} & C & \\ & k \times c & \end{bmatrix}$$

Figure 16.2 SVD factors a matrix X into a product of three matrices, W, Σ, and C. Taking the first k dimensions gives a $|V| \times k$ matrix W_k that has one k-dimensioned row per word that can be used as an embedding.

Using only the top k dimensions (corresponding to the k most important singular values) leads to a reduced $|V| \times k$ matrix W_k, with one k-dimensioned row per word. This row now acts as a dense k-dimensional vector (embedding) representing that word, substituting for the very high-dimensional rows of the original X.

LSA embeddings generally set k=300, so these embeddings are relatively short by comparison to other dense embeddings.

Instead of PPMI or tf-idf weighting on the original term-document matrix, LSA implementations generally use a particular weighting of each co-occurrence cell that multiplies two weights called the **local** and **global** weights for each cell (i, j)—term i in document j. The local weight of each term i is its log frequency: $\log f(i, j) + 1$ The global weight of term i is a version of its entropy: $1 + \frac{\sum_j p(i,j) \log p(i,j)}{\log D}$, where D

is the number of documents.

LSA has also been proposed as a cognitive model for human language use (Landauer and Dumais, 1997) and applied to a wide variety of NLP applications; see the end of the chapter for details.

16.1.2 SVD applied to word-context matrices

Rather than applying SVD to the term-document matrix (as in the LSA algorithm of the previous section), an alternative that is widely practiced is to apply SVD to the word-word or word-context matrix. In this version the context dimensions are words rather than documents, an idea first proposed by Schütze (1992).

The mathematics is identical to what is described in Fig. 16.2: SVD factorizes the word-context matrix X into three matrices W, Σ, and C^T. The only difference is that we are starting from a PPMI-weighted word-word matrix, instead of a term-document matrix.

Once again only the top k dimensions are retained (corresponding to the k most important singular values), leading to a reduced $|V| \times k$ matrix W_k, with one k-dimensioned row per word. Just as with LSA, this row acts as a dense k-dimensional vector (embedding) representing that word. The other matrices (Σ and C) are simply thrown away. [1]

truncated SVD

This use of just the top dimensions, whether for a term-document matrix like LSA, or for a term-term matrix, is called **truncated SVD**. Truncated SVD is parameterized by k, the number of dimensions in the representation for each word, typically ranging from 500 to 5000. Thus SVD run on term-context matrices tends to use many more dimensions than the 300-dimensional embeddings produced by LSA. This difference presumably has something to do with the difference in granularity; LSA counts for words are much coarser-grained, counting the co-occurrences in an entire document, while word-context PPMI matrices count words in a small window. Generally the dimensions we keep are the highest-order dimensions, although for some tasks, it helps to throw out a small number of the most high-order dimensions, such as the first 1 or even the first 50 (Lapesa and Evert, 2014).

Fig. 16.3 shows a high-level sketch of the entire SVD process. The dense embeddings produced by SVD sometimes perform better than the raw PPMI matrices on semantic tasks like word similarity. Various aspects of the dimensionality reduction seem to be contributing to the increased performance. If low-order dimensions represent unimportant information, the truncated SVD may be acting to removing noise. By removing parameters, the truncation may also help the models generalize better to unseen data. When using vectors in NLP tasks, having a smaller number of dimensions may make it easier for machine learning classifiers to properly weight the dimensions for the task. And as mentioned above, the models may do better at capturing higher order co-occurrence.

Nonetheless, there is a significant computational cost for the SVD for a large co-occurrence matrix, and performance is not always better than using the full sparse PPMI vectors, so for many applications the sparse vectors are the right approach. Alternatively, the neural embeddings we discuss in the next section provide a popular efficient solution to generating dense embeddings.

[1] Some early systems weighted W_k by the singular values, using the product $W_k \cdot \Sigma_k$ as an embedding instead of just the matrix W_k, but this weighting leads to significantly worse embeddings and is not generally used (Levy et al., 2015).

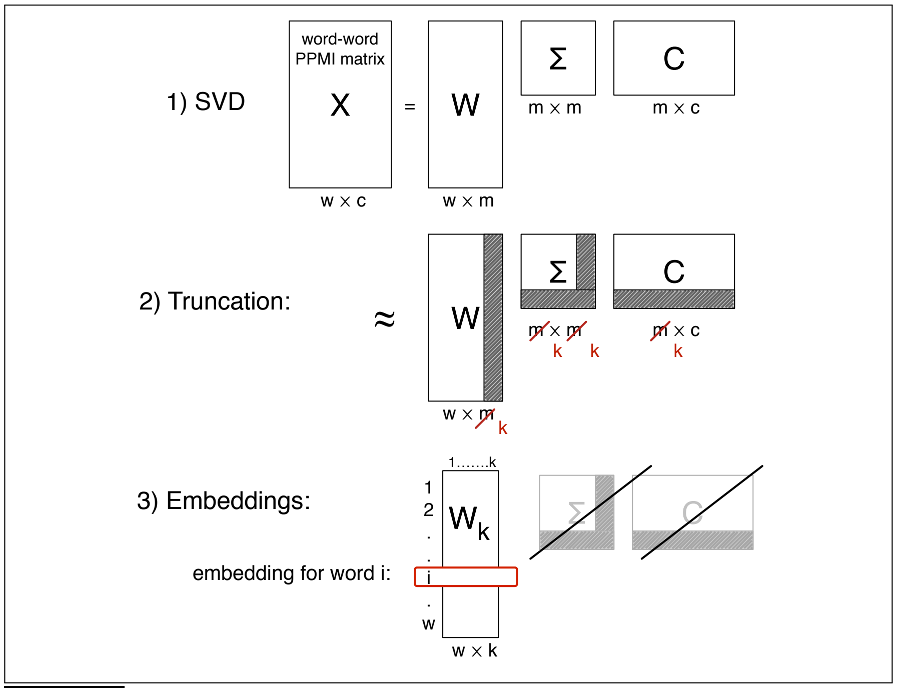

Figure 16.3 Sketching the use of SVD to produce a dense embedding of dimensionality k from a sparse PPMI matrix of dimensionality c. The SVD is used to factorize the word-word PPMI matrix into a W, Σ, and C matrix. The Σ and C matrices are discarded, and the W matrix is truncated giving a matrix of k-dimensionality embedding vectors for each word.

16.2 Embeddings from prediction: Skip-gram and CBOW

A second method for generating dense embeddings draws its inspiration from the neural network models used for language modeling. Recall from Chapter 8 that neural network language models are given a word and predict context words. This prediction process can be used to learn embeddings for each target word. The intuition is that words with similar meanings often occur near each other in texts. The neural models therefore learn an embedding by starting with a random vector and then iteratively shifting a word's embeddings to be more like the embeddings of neighboring words, and less like the embeddings of words that don't occur nearby.

Although the metaphor for this architecture comes from word prediction, we'll see that the process for learning these neural embeddings actually has a strong relationship to PMI co-occurrence matrices, SVD factorization, and dot-product similarity metrics.

word2vec **skip-gram** **CBOW**

The most popular family of methods is referred to as **word2vec**, after the software package that implements two methods for generating dense embeddings: **skip-gram** and **CBOW** (**continuous bag of words**) (Mikolov et al. 2013, Mikolov et al. 2013a). Like the neural language models, the word2vec models learn embeddings by training a network to predict neighboring words. But in this case the prediction task is not the main goal; words that are semantically similar often occur near each other in text, and so embeddings that are good at predicting neighboring words are also good at representing similarity. The advantage of the word2vec methods is that they are fast, efficient to train, and easily available online with code and pretrained embeddings.

We'll begin with the skip-gram model. Like the SVD model in the previous

word embedding **context embedding**

section, the skip-gram model actually learns two separate embeddings for each word w: the **word embedding** v and the **context embedding** c. These embeddings are encoded in two matrices, the **word matrix** W and the **context matrix** C. We'll discuss in Section 16.2.1 how W and C are learned, but let's first see how they are used. Each row i of the word matrix W is the $1 \times d$ vector embedding v_i for word i in the vocabulary. Each column i of the context matrix C is a $d \times 1$ vector embedding c_i for word i in the vocabulary. In principle, the word matrix and the context matrix could use different vocabularies V_w and V_c. For the remainder of the chapter, however we'll simplify by assuming the two matrices share the same vocabulary, which we'll just call V.

Let's consider the prediction task. We are walking through a corpus of length T and currently pointing at the tth word $w^{(t)}$, whose index in the vocabulary is j, so we'll call it w_j $(1 < j < |V|)$. The skip-gram model predicts each neighboring word in a context window of $2L$ words from the current word. So for a context window $L = 2$ the context is $[w^{t-2}, w^{t-1}, w^{t+1}, w^{t+2}]$ and we are predicting each of these from word w_j. But let's simplify for a moment and imagine just predicting one of the $2L$ context words, for example $w^{(t+1)}$, whose index in the vocabulary is k $(1 < k < |V|)$. Hence our task is to compute $P(w_k|w_j)$.

The heart of the skip-gram computation of the probability $p(w_k|w_j)$ is computing the dot product between the vectors for w_k and w_j, the *context vector* for w_k and the *target vector* for w_j. For simplicity, we'll represent this dot product as $c_k \cdot v_j$, (although more correctly, it should be $c_k^T v_j$), where c_k is the context vector of word k and v_j is the target vector for word j. As we saw in the previous chapter, the higher the dot product between two vectors, the more similar they are. (That was the intuition of using the cosine as a similarity metric, since cosine is just a normalized dot product). Fig. 16.4 shows the intuition that the similarity function requires selecting out a target vector v_j from W, and a context vector c_k from C.

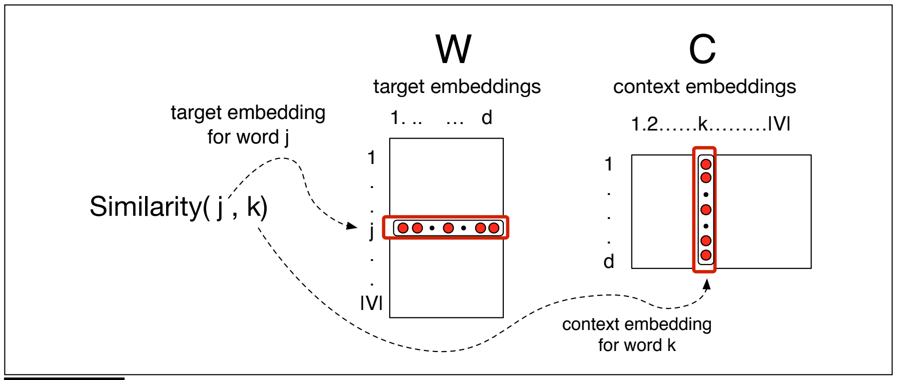

Figure 16.4

Of course, the dot product $c_k \cdot v_j$ is not a probability, it's just a number ranging from $-\infty$ to ∞. We can use the *softmax* function from Chapter 8 to normalize the dot product into probabilities. Computing this denominator requires computing the dot product between each other word w in the vocabulary with the target word w_i:

$$p(w_k|w_j) = \frac{exp(c_k \cdot v_j)}{\sum_{i \in |V|} exp(c_i \cdot v_j)} \quad (16.1)$$

In summary, the skip-gram computes the probability $p(w_k|w_j)$ by taking the dot product between the word vector for j (v_j) and the context vector for k (c_k), and turning this dot product $v_j \cdot c_k$ into a probability by passing it through a softmax function.

This version of the algorithm, however, has a problem: the time it takes to compute the denominator. For each word w^t, the denominator requires computing the dot product with all other words. As we'll see in the next section, we generally solve this by using an approximation of the denominator.

CBOW The CBOW (**continuous bag of words**) model is roughly the mirror image of the skip-gram model. Like skip-grams, it is based on a predictive model, but this time predicting the current word w_t from the context window of $2L$ words around it, e.g. for $L = 2$ the context is $[w_{t-2}, w_{t-1}, w_{t+1}, w_{t+2}]$

While CBOW and skip-gram are similar algorithms and produce similar embeddings, they do have slightly different behavior, and often one of them will turn out to be the better choice for any particular task.

16.2.1 Learning the word and context embeddings

We already mentioned the intuition for learning the word embedding matrix W and the context embedding matrix C: iteratively make the embeddings for a word more like the embeddings of its neighbors and less like the embeddings of other words.

In the version of the prediction algorithm suggested in the previous section, the probability of a word is computed by normalizing the dot-product between a word and each context word by the dot products for all words. This probability is optimized when a word's vector is closest to the words that occur near it (the numerator), and further from every other word (the denominator). Such a version of the algorithm is very expensive; we need to compute a whole lot of dot products to make the denominator.

Instead, the most commonly used version of skip-gram, *skip-gram with negative sampling*, approximates this full denominator.

This section offers a brief sketch of how this works. In the training phase, the algorithm walks through the corpus, at each target word choosing the surrounding context words as positive examples, and for each positive example also choosing k **noise** samples or **negative samples**: non-neighbor words. The goal will be to move the embeddings toward the neighbor words and away from the noise words.

negative samples

For example, in walking through the example text below we come to the word *apricot*, and let $L = 2$ so we have 4 context words c1 through c4:

```
lemon,  a [tablespoon of apricot preserves or] jam
             c1          c2    w      c3       c4
```

The goal is to learn an embedding whose dot product with each context word is high. In practice skip-gram uses a sigmoid function σ of the dot product, where $\sigma(x) = \frac{1}{1+e^{-x}}$. So for the above example we want $\sigma(c1 \cdot w) + \sigma(c2 \cdot w) + \sigma(c3 \cdot w) + \sigma(c4 \cdot w)$ to be high.

In addition, for each context word the algorithm chooses k random noise words according to their unigram frequency. If we let $k = 2$, for each target/context pair, we'll have 2 noise words for each of the 4 context words:

```
[cement metaphysical dear coaxial    apricot attendant whence forever puddle]
 n1     n2           n3   n4                 n5        n6     n7        n8
```

We'd like these noise words n to have a low dot-product with our target embedding w; in other words we want $\sigma(n1 \cdot w) + \sigma(n2 \cdot w) + ... + \sigma(n8 \cdot w)$ to be low.

More formally, the learning objective for one word/context pair (w,c) is

$$\log \sigma(c \cdot w) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim p(w)} [\log \sigma(-w_i \cdot w)] \tag{16.2}$$

That is, we want to maximize the dot product of the word with the actual context words, and minimize the dot products of the word with the k negative sampled non-neighbor words. The noise words w_i are sampled from the vocabulary V according to their weighted unigram probability; in practice rather than $p(w)$ it is common to use the weighting $p^{\frac{3}{4}}(w)$.

The learning algorithm starts with randomly initialized W and C matrices, and then walks through the training corpus moving W and C so as to maximize the objective in Eq. 16.2. An algorithm like stochastic gradient descent is used to iteratively shift each value so as to maximize the objective, using error backpropagation to propagate the gradient back through the network as described in Chapter 8 (Mikolov et al., 2013a).

In summary, the learning objective in Eq. 16.2 is not the same as the $p(w_k|w_j)$ defined in Eq. 16.3. Nonetheless, although negative sampling is a different objective than the probability objective, and so the resulting dot products will not produce optimal predictions of upcoming words, it seems to produce good embeddings, and that's the goal we care about.

Visualizing the network Using error backpropagation requires that we envision the selection of the two vectors from the W and C matrices as a network that we can propagate backwards across. Fig. 16.5 shows a simplified visualization of the model; we've simplified to predict a single context word rather than 2L context words, and simplified to show the softmax over the entire vocabulary rather than just the k noise words.

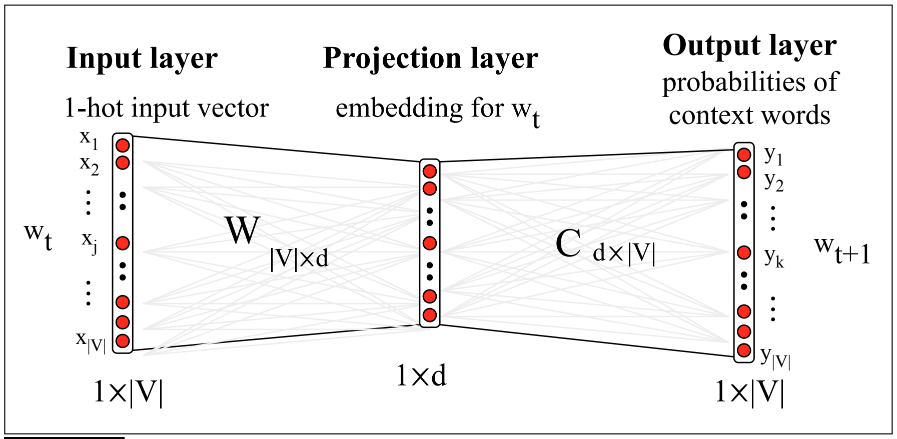

Figure 16.5 The skip-gram model viewed as a network (Mikolov et al. 2013, Mikolov et al. 2013a).

It's worth taking a moment to envision how the network is computing the same probability as the dot product version we described above. In the network of Fig. 16.5, we begin with an input vector x, which is a **one-hot** vector for the current word w_j. A one-hot vector is just a vector that has one element equal to 1, and all the other elements are set to zero. Thus in a one-hot representation for the word w_j, $x_j = 1$, and $x_i = 0 \ \forall i \neq j$, as shown in Fig. 16.6.

one-hot

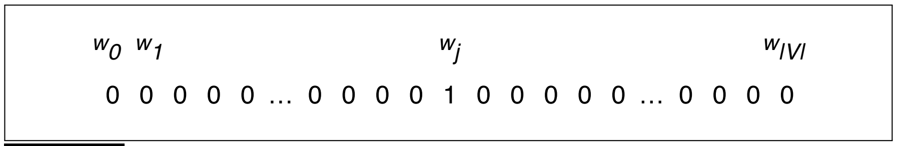

Figure 16.6 A one-hot vector, with the dimension corresponding to word w_j set to 1.

We then predict the probability of each of the $2L$ output words—in Fig. 16.5 that means the one output word w_{t+1}— in 3 steps:

projection layer

1. **Select the embedding from W**: x is multiplied by W, the input matrix, to give the hidden or **projection layer**. Since each row of the input matrix W is just an embedding for word w_t, and the input is a one-hot columnvector for w_j, the projection layer for input x will be $h = W * w_j = v_j$, the input embedding for w_j.
2. **Compute the dot product** $c_k \cdot v_j$: For each of the $2L$ context words we now multiply the projection vector h by the context matrix C. The result for each context word, $o = Ch$, is a $1 \times |V|$ dimensional output vector giving a score for each of the $|V|$ vocabulary words. In doing so, the element o_k was computed by multiplying h by the *output embedding* for word w_k: $o_k = c_k \cdot h = c_k \cdot v_j$.
3. **Normalize the dot products into probabilities**: For each context word we normalize this vector of dot product scores, turning each score element o_k into a probability by using the soft-max function:

$$p(w_k|w_j) = y_k = \frac{exp(c_k \cdot v_j)}{\sum_{i \in |V|} exp(c_i \cdot v_j)} \tag{16.3}$$

16.2.2 Relationship between different kinds of embeddings

There is an interesting relationship between skip-grams, SVD/LSA, and PPMI. If we multiply the two context matrices WC, we produce a $|V| \times |V|$ matrix X, each entry x_{ij} corresponding to some association between input word i and context word j. Levy and Goldberg (2014b) prove that skip-gram's optimal value occurs when this learned matrix is actually a version of the PMI matrix, with the values shifted by k:

$$WC = X^{\text{PMI}} - \log k \tag{16.4}$$

In other words, skip-gram is implicitly factorizing a (shifted version of the) PMI matrix into the two embedding matrices W and C, just as SVD did, albeit with a different kind of factorization. See Levy and Goldberg (2014b) for more details.

Once the embeddings are learned, we'll have two embeddings for each word w_i: v_i and c_i. We can choose to throw away the C matrix and just keep W, as we did with SVD, in which case each word i will be represented by the vector v_i.

Alternatively we can add the two embeddings together, using the summed embedding $v_i + c_i$ as the new d-dimensional embedding, or we can concatenate them into an embedding of dimensionality $2d$.

As with the simple count-based methods like PPMI, the context window size L effects the performance of skip-gram embeddings, and experiments often tune the parameter L on a dev set. As as with PPMI, window sizing leads to qualitative differences: smaller windows capture more syntactic information, larger ones more semantic and relational information. One difference from the count-based methods is that for skip-grams, the larger the window size the more computation the algorithm requires for training (more neighboring words must be predicted). See the end of the chapter for a pointer to surveys which have explored parameterizations like window-size for different tasks.

16.3 Properties of embeddings

We'll discuss in Section **??** how to evaluate the quality of different embeddings. But it is also sometimes helpful to visualize them. Fig. 16.7 shows the words/phrases that are most similar to some sample words using the phrase-based version of the skip-gram algorithm (Mikolov et al., 2013a).

target:	Redmond	Havel	ninjutsu	graffiti	capitulate
	Redmond Wash.	Vaclav Havel	ninja	spray paint	capitulation
	Redmond Washington	president Vaclav Havel	martial arts	graffiti	capitulated
	Microsoft	Velvet Revolution	swordsmanship	taggers	capitulating

Figure 16.7 Examples of the closest tokens to some target words using a phrase-based extension of the skip-gram algorithm (Mikolov et al., 2013a).

One semantic property of various kinds of embeddings that may play in their usefulness is their ability to capture relational meanings

Mikolov et al. (2013b) demonstrates that the *offsets* between vector embeddings can capture some relations between words, for example that the result of the expression vector(*'king'*) - vector(*'man'*) + vector(*'woman'*) is a vector close to vector(*'queen'*); the left panel in Fig. 16.8 visualizes this by projecting a representation down into 2 dimensions. Similarly, they found that the expression vector(*'Paris'*) - vector(*'France'*) + vector(*'Italy'*) results in a vector that is very close to vector(*'Rome'*). Levy and Goldberg (2014a) shows that various other kinds of embeddings also seem to have this property.

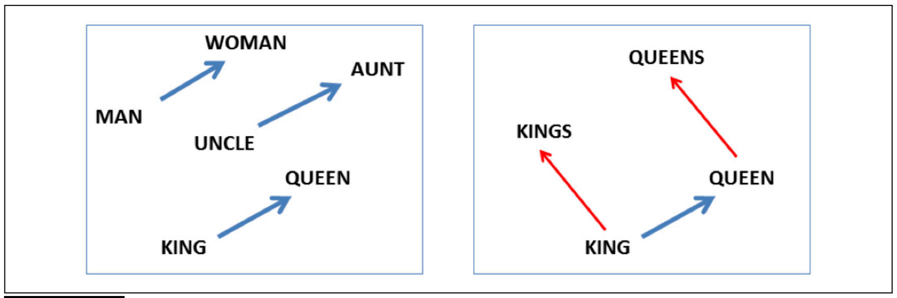

Figure 16.8 Vector offsets showing relational properties of the vector space, shown by projecting vectors onto two dimensions using PCA. In the left panel, 'king' - 'man' + 'woman' is close to 'queen'. In the right, we see the way offsets seem to capture grammatical number (Mikolov et al., 2013b).

16.4 Brown Clustering

Brown clustering

Brown clustering (Brown et al., 1992) is an agglomerative clustering algorithm for deriving vector representations of words by clustering words based on their associations with the preceding or following words.

class-based language model

The algorithm makes use of the **class-based language model** (Brown et al., 1992), a model in which each word $w \in V$ belongs to a class $c \in C$ with a probability $P(w|c)$. Class based LMs assigns a probability to a pair of words w_{i-1} and w_i by modeling the transition between classes rather than between words:

$$P(w_i|w_{i-1}) = P(c_i|c_{i-1})P(w_i|c_i) \tag{16.5}$$

The class-based LM can be used to assign a probability to an entire corpus given a particularly clustering C as follows:

$$P(\text{corpus}|C) = \prod_{i-1}^{n} P(c_i|c_{i-1})P(w_i|c_i) \tag{16.6}$$

Class-based language models are generally not used as a language model for applications like machine translation or speech recognition because they don't work as well as standard n-grams or neural language models. But they are an important component in Brown clustering.

Brown clustering is a hierarchical clustering algorithm. Let's consider a naive (albeit inefficient) version of the algorithm:

1. Each word is initially assigned to its own cluster.
2. We now consider consider merging each pair of clusters. The pair whose merger results in the smallest decrease in the likelihood of the corpus (according to the class-based language model) is merged.
3. Clustering proceeds until all words are in one big cluster.

Two words are thus most likely to be clustered if they have similar probabilities for preceding and following words, leading to more coherent clusters. The result is that words will be merged if they are contextually similar.

By tracing the order in which clusters are merged, the model builds a binary tree from bottom to top, in which the leaves are the words in the vocabulary, and each intermediate node in the tree represents the cluster that is formed by merging its children. Fig. 16.9 shows a schematic view of a part of a tree.

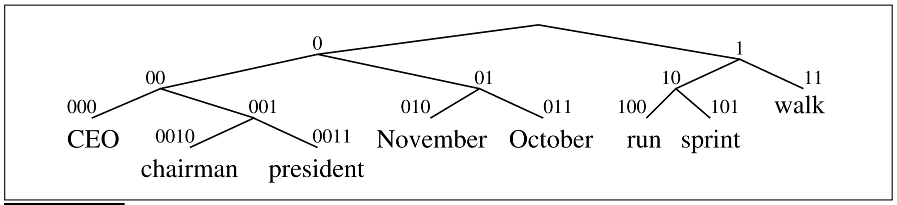

Figure 16.9 Brown clustering as a binary tree. A full binary string represents a word; each binary prefix represents a larger class to which the word belongs and can be used as a vector representation for the word. After Koo et al. (2008).

After clustering, a word can be represented by the binary string that corresponds to its path from the root node; 0 for left, 1 for right, at each choice point in the binary tree. For example in Fig. 16.9, the word *chairman* is the vector `0010` and *October* is `011`. Since Brown clustering is a **hard clustering** algorithm (each word has only one cluster), there is just one string per word.

hard clustering

Now we can extract useful features by taking the binary prefixes of this bit string; each prefix represents a cluster to which the word belongs. For example the string `01` in the figure represents the cluster of month names $\{$*November, October*$\}$, the string

`0001` the names of common nouns for corporate executives {*chairman, president*}, `1` is verbs {*run, sprint, walk*}, and `0` is nouns. These prefixes can then be used as a vector representation for the word; the shorter the prefix, the more abstract the cluster. The length of the vector representation can thus be adjusted to fit the needs of the particular task. Koo et al. (2008) improving parsing by using multiple features: a 4-6 bit prefix to capture part of speech information and a full bit string to represent words. Spitkovsky et al. (2011) shows that vectors made of the first 8 or 9-bits of a Brown clustering perform well at grammar induction. Because they are based on immediately neighboring words, Brown clusters are most commonly used for representing the syntactic properties of words, and hence are commonly used as a feature in parsers. Nonetheless, the clusters do represent some semantic properties as well. Fig. 16.10 shows some examples from a large clustering from Brown et al. (1992).

Friday Monday Thursday Wednesday Tuesday Saturday Sunday weekends Sundays Saturdays
June March July April January December October November September August
pressure temperature permeability density porosity stress velocity viscosity gravity tension
anyone someone anybody somebody
had hadn't hath would've could've should've must've might've
asking telling wondering instructing informing kidding reminding bothering thanking deposing
mother wife father son husband brother daughter sister boss uncle
great big vast sudden mere sheer gigantic lifelong scant colossal
down backwards ashore sideways southward northward overboard aloft downwards adrift

Figure 16.10 Some sample Brown clusters from a 260,741-word vocabulary trained on 366 million words of running text (Brown et al., 1992). Note the mixed syntactic-semantic nature of the clusters.

Note that the naive version of the Brown clustering algorithm described above is extremely inefficient — $O(n^5)$: at each of n iterations, the algorithm considers each of n^2 merges, and for each merge, compute the value of the clustering by summing over n^2 terms. because it has to consider every possible pair of merges. In practice we use more efficient $O(n^3)$ algorithms that use tables to pre-compute the values for each merge (Brown et al. 1992, Liang 2005).

16.5 Summary

- **Singular Value Decomposition** (**SVD**) is a dimensionality technique that can be used to create lower-dimensional embeddings from a full term-term or term-document matrix.
- **Latent Semantic Analysis** is an application of SVD to the term-document matrix, using particular weightings and resulting in embeddings of about 300 dimensions.
- Two algorithms inspired by neural language models, **skip-gram** and **CBOW**, are popular efficient ways to compute embeddings. They learn embeddings (in a way initially inspired from the neural word prediction literature) by finding embeddings that have a high dot-product with neighboring words and a low dot-product with noise words.
- **Brown clustering** is a method of grouping words into clusters based on their relationship with the preceding and following words. Brown clusters can be

used to create bit-vectors for a word that can function as a syntactic representation.

Bibliographical and Historical Notes

The use of SVD as a way to reduce large sparse vector spaces for word meaning, like the vector space model itself, was first applied in the context of information retrieval, briefly as **latent semantic indexing** (LSI) (Deerwester et al., 1988) and then afterwards as **LSA** (**latent semantic analysis**) (Deerwester et al., 1990). LSA was based on applying SVD to the term-document matrix (each cell weighted by log frequency and normalized by entropy), and then using generally the top 300 dimensions as the embedding. Landauer and Dumais (1997) summarizes LSA as a cognitive model. LSA was then quickly applied to a wide variety of NLP applications: spell checking (Jones and Martin, 1997), language modeling (Bellegarda 1997, Coccaro and Jurafsky 1998, Bellegarda 2000) morphology induction (Schone and Jurafsky 2000, Schone and Jurafsky 2001), and essay grading (Rehder et al., 1998).

The idea of SVD on the term-term matrix (rather than the term-document matrix) as a model of meaning for NLP was proposed soon after LSA by Schütze (1992). Schütze applied the low-rank (97-dimensional) embeddings produced by SVD to the task of word sense disambiguation, analyzed the resulting semantic space, and also suggested possible techniques like dropping high-order dimensions. See Schütze (1997).

A number of alternative matrix models followed on from the early SVD work, including Probabilistic Latent Semantic Indexing (PLSI) (Hofmann, 1999) Latent Dirichlet Allocation (LDA) (Blei et al., 2003). Nonnegative Matrix Factorization (NMF) (Lee and Seung, 1999).

Neural networks were used as a tool for language modeling by Bengio et al. (2003) and Bengio et al. (2006), and extended to recurrent net language models in Mikolov et al. (2011). Collobert and Weston (2007), Collobert and Weston (2008), and Collobert et al. (2011) is a very influential line of work demonstrating that embeddings could play a role as the first representation layer for representing word meanings for a number of NLP tasks. The idea of simplifying the hidden layer of these neural net language models to create the skip-gram and CBOW algorithms was proposed by Mikolov et al. (2013). The negative sampling training algorithm was proposed in Mikolov et al. (2013a). Both algorithms were made available in the `word2vec` package, and the resulting embeddings widely used in many applications.

The development of models of embeddings is an active research area, with new models including GloVe (Pennington et al., 2014) (based on ratios of probabilities from the word-word co-occurrence matrix), or sparse embeddings based on non-negative matrix factorization (Fyshe et al., 2015). Many survey experiments have explored the parameterizations of different kinds of vector space embeddings and their parameterizations, including sparse and dense vectors, and count-based and predict-based models (Dagan 2000, ?, Curran 2003, Bullinaria and Levy 2007, Bullinaria and Levy 2012, Lapesa and Evert 2014, Kiela and Clark 2014, Levy et al. 2015).

Exercises

Bellegarda, J. R. (1997). A latent semantic analysis framework for large-span language modeling. In *Eurospeech-97*, Rhodes, Greece.

Bellegarda, J. R. (2000). Exploiting latent semantic information in statistical language modeling. *Proceedings of the IEEE*, *89*(8), 1279–1296.

Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. (2003). A neural probabilistic language model. *JMLR*, *3*, 1137–1155.

Bengio, Y., Schwenk, H., Senécal, J.-S., Morin, F., and Gauvain, J.-L. (2006). Neural probabilistic language models. In *Innovations in Machine Learning*, pp. 137–186. Springer.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent Dirichlet allocation. *Journal of Machine Learning Research*, *3*(5), 993–1022.

Brown, P. F., Della Pietra, V. J., de Souza, P. V., Lai, J. C., and Mercer, R. L. (1992). Class-based n-gram models of natural language. *Computational Linguistics*, *18*(4), 467–479.

Bullinaria, J. A. and Levy, J. P. (2007). Extracting semantic representations from word co-occurrence statistics: A computational study. *Behavior research methods*, *39*(3), 510–526.

Bullinaria, J. A. and Levy, J. P. (2012). Extracting semantic representations from word co-occurrence statistics: stop-lists, stemming, and svd. *Behavior research methods*, *44*(3), 890–907.

Coccaro, N. and Jurafsky, D. (1998). Towards better integration of semantic predictors in statistical language modeling. In *ICSLP-98*, Sydney, Vol. 6, pp. 2403–2406.

Collobert, R. and Weston, J. (2007). Fast semantic extraction using a novel neural network architecture. In *ACL-07*, pp. 560–567.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML*, pp. 160–167.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, *12*, 2493–2537.

Curran, J. R. (2003). *From Distributional to Semantic Similarity*. Ph.D. thesis, University of Edinburgh.

Dagan, I. (2000). Contextual word similarity. In Dale, R., Moisl, H., and Somers, H. L. (Eds.), *Handbook of Natural Language Processing*. Marcel Dekker.

Deerwester, S., Dumais, S., Furnas, G., Harshman, R., Landauer, T., Lochbaum, K., and Streeter, L. (1988). Computer information retrieval using latent semantic structure: Us patent 4,839,853..

Deerwester, S. C., Dumais, S. T., Landauer, T. K., Furnas, G. W., and Harshman, R. A. (1990). Indexing by latent semantics analysis. *JASIS*, *41*(6), 391–407.

Fyshe, A., Wehbe, L., Talukdar, P. P., Murphy, B., and Mitchell, T. M. (2015). A compositional and interpretable semantic space. In *NAACL HLT 2015*.

Hofmann, T. (1999). Probabilistic latent semantic indexing. In *SIGIR-99*, Berkeley, CA.

Jones, M. P. and Martin, J. H. (1997). Contextual spelling correction using latent semantic analysis. In *ANLP 1997*, Washington, D.C., pp. 166–173.

Kiela, D. and Clark, S. (2014). A systematic study of semantic vector space model parameters. In *Proceedings of the EACL 2nd Workshop on Continuous Vector Space Models and their Compositionality (CVSC)*, pp. 21–30.

Koo, T., Carreras Pérez, X., and Collins, M. (2008). Simple semi-supervised dependency parsing. In *ACL-08*, pp. 598–603.

Landauer, T. K. and Dumais, S. T. (1997). A solution to Plato's problem: The Latent Semantic Analysis theory of acquisition, induction, and representation of knowledge. *Psychological Review*, *104*, 211–240.

Lapesa, G. and Evert, S. (2014). A large scale evaluation of distributional semantic models: Parameters, interactions and model selection. *TACL*, *2*, 531–545.

Lee, D. D. and Seung, H. S. (1999). Learning the parts of objects by non-negative matrix factorization. *Nature*, *401*(6755), 788–791.

Levy, O. and Goldberg, Y. (2014a). Linguistic regularities in sparse and explicit word representations. In *CoNLL-14*.

Levy, O. and Goldberg, Y. (2014b). Neural word embedding as implicit matrix factorization. In *NIPS 14*, pp. 2177–2185.

Levy, O., Goldberg, Y., and Dagan, I. (2015). Improving distributional similarity with lessons learned from word embeddings. *TACL*, *3*, 211–225.

Liang, P. (2005). *Semi-supervised learning for natural language* Master's thesis, Massachusetts Institute of Technology.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. In *ICLR 2013*.

Mikolov, T., Kombrink, S., Burget, L., Černockỳ, J. H., and Khudanpur, S. (2011). Extensions of recurrent neural network language model. In *ICASSP-11*, pp. 5528–5531.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013a). Distributed representations of words and phrases and their compositionality. In *NIPS 13*, pp. 3111–3119.

Mikolov, T., Yih, W.-t., and Zweig, G. (2013b). Linguistic regularities in continuous space word representations. In *NAACL HLT 2013*, pp. 746–751.

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *EMNLP 2014*, pp. 1532–1543.

Rehder, B., Schreiner, M. E., Wolfe, M. B. W., Laham, D., Landauer, T. K., and Kintsch, W. (1998). Using Latent Semantic Analysis to assess knowledge: Some technical considerations. *Discourse Processes*, *25*(2-3), 337–354.

Schone, P. and Jurafsky, D. (2000). Knowlege-free induction of morphology using latent semantic analysis. In *CoNLL-00*.

Schone, P. and Jurafsky, D. (2001). Knowledge-free induction of inflectional morphologies. In *NAACL 2001*.

Schütze, H. (1992). Dimensions of meaning. In *Proceedings of Supercomputing '92*, pp. 787–796. IEEE Press.

Schütze, H. (1997). *Ambiguity Resolution in Language Learning – Computational and Cognitive Models*. CSLI, Stanford, CA.

Spitkovsky, V. I., Alshawi, H., Chang, A. X., and Jurafsky, D. (2011). Ynsupervised dependency parsing without gold part-of-speech tags. In *EMNLP-11*, pp. 1281–1290.