at which point we can see how the model assigns topics to each user's interests:

```
for document, topic_counts in zip(documents, document_topic_counts):
    print(document)
    for topic, count in topic_counts.most_common():
        if count > 0:
            print(topic_names[topic], count)
    print()
```

which gives:

```
['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
Big Data and programming languages 4 databases 3
['NoSQL', 'MongoDB', 'Cassandra', 'HBase', 'Postgres']
databases 5
['Python', 'scikit-learn', 'scipy', 'numpy', 'statsmodels', 'pandas']
Python and statistics 5 machine learning 1
```

and so on. Given the "ands" we needed in some of our topic names, it's possible we should use more topics, although most likely we don't have enough data to successfully learn them.

Word Vectors

A lot of recent advances in NLP involve deep learning. In the rest of this chapter we'll look at a couple of them using the machinery we developed in Chapter 19.

One important innovation involves representing words as low-dimensional vectors. These vectors can be compared, added together, fed into machine learning models, or anything else you want to do with them. They usually have nice properties; for example, similar words tend to have similar vectors. That is, typically the word vector for *big* is pretty close to the word vector for *large*, so that a model operating on word vectors can (to some degree) handle things like synonymy for free.

Frequently the vectors will exhibit delightful arithmetic properties as well. For instance, in some such models if you take the vector for *king*, subtract the vector for *man*, and add the vector for *woman*, you will end up with a vector that's very close to the vector for *queen*. It can be interesting to ponder what this means about what the word vectors actually "learn," although we won't spend time on that here.

Coming up with such vectors for a large vocabulary of words is a difficult undertaking, so typically we'll *learn* them from a corpus of text. There are a couple of different schemes, but at a high level the task typically looks something like this:

1. Get a bunch of text.
2. Create a dataset where the goal is to predict a word given nearby words (or alternatively, to predict nearby words given a word).
3. Train a neural net to do well on this task.
4. Take the internal states of the trained neural net as the word vectors.

In particular, because the task is to predict a word given nearby words, words that occur in similar contexts (and hence have similar nearby words) should have similar internal states and therefore similar word vectors.

Here we'll measure "similarity" using *cosine similarity*, which is a number between –1 and 1 that measures the degree to which two vectors point in the same direction:

```
from scratch.linear_algebra import dot, Vector
import math

def cosine_similarity(v1: Vector, v2: Vector) -> float:
    return dot(v1, v2) / math.sqrt(dot(v1, v1) * dot(v2, v2))

assert cosine_similarity([1., 1, 1], [2., 2, 2]) == 1, "same direction"
assert cosine_similarity([-1., -1], [2., 2]) == -1,    "opposite direction"
assert cosine_similarity([1., 0], [0., 1]) == 0,       "orthogonal"
```

Let’s learn some word vectors to see how this works.

To start with, we’ll need a toy dataset. The commonly used word vectors are typically derived from training on millions or even billions of words. As our toy library can’t cope with that much data, we’ll create an artificial dataset with some structure to it:

```
colors = ["red", "green", "blue", "yellow", "black", ""]
nouns = ["bed", "car", "boat", "cat"]
verbs = ["is", "was", "seems"]
adverbs = ["very", "quite", "extremely", ""]
adjectives = ["slow", "fast", "soft", "hard"]

def make_sentence() -> str:
    return " ".join([
        "The",
        random.choice(colors),
        random.choice(nouns),
        random.choice(verbs),
        random.choice(adverbs),
        random.choice(adjectives),
        "."
    ])

NUM_SENTENCES = 50

random.seed(0)
sentences = [make_sentence() for _ in range(NUM_SENTENCES)]
```

This will generate lots of sentences with similar structure but different words; for example, “The green boat seems quite slow.” Given this setup, the colors will mostly appear in “similar” contexts, as will the nouns, and so on. So if we do a good job of assigning word vectors, the colors should get similar vectors, and so on.

> **NOTE**
>
> In practical usage, you’d probably have a corpus of millions of sentences, in which case you’d get “enough” context from the sentences as they are. Here, with only 50 sentences, we have to make them somewhat artificial.

As mentioned earlier, we'll want to one-hot-encode our words, which means we'll need to convert them to IDs. We'll introduce a `Vocabulary` class to keep track of this mapping:

```
from scratch.deep_learning import Tensor

class Vocabulary:
    def __init__(self, words: List[str] = None) -> None:
        self.w2i: Dict[str, int] = {}  # mapping word -> word_id
        self.i2w: Dict[int, str] = {}  # mapping word_id -> word

        for word in (words or []):     # If words were provided,
            self.add(word)             # add them.

    @property
    def size(self) -> int:
        """how many words are in the vocabulary"""
        return len(self.w2i)

    def add(self, word: str) -> None:
        if word not in self.w2i:         # If the word is new to us:
            word_id = len(self.w2i)      # Find the next id.
            self.w2i[word] = word_id     # Add to the word -> word_id map.
            self.i2w[word_id] = word     # Add to the word_id -> word map.

    def get_id(self, word: str) -> int:
        """return the id of the word (or None)"""
        return self.w2i.get(word)

    def get_word(self, word_id: int) -> str:
        """return the word with the given id (or None)"""
        return self.i2w.get(word_id)

    def one_hot_encode(self, word: str) -> Tensor:
        word_id = self.get_id(word)
        assert word_id is not None, f"unknown word {word}"

        return [1.0 if i == word_id else 0.0 for i in range(self.size)]
```

These are all things we could do manually, but it's handy to have it in a class. We should probably test it:

```
vocab = Vocabulary(["a", "b", "c"])
assert vocab.size == 3,              "there are 3 words in the vocab"
```

```
assert vocab.get_id("b") == 1,       "b should have word_id 1"
assert vocab.one_hot_encode("b") == [0, 1, 0]
assert vocab.get_id("z") is None,    "z is not in the vocab"
assert vocab.get_word(2) == "c",     "word_id 2 should be c"
vocab.add("z")
assert vocab.size == 4,              "now there are 4 words in the vocab"
assert vocab.get_id("z") == 3,       "now z should have id 3"
assert vocab.one_hot_encode("z") == [0, 0, 0, 1]
```

We should also write simple helper functions to save and load a vocabulary, just as we have for our deep learning models:

```
import json

def save_vocab(vocab: Vocabulary, filename: str) -> None:
    with open(filename, 'w') as f:
        json.dump(vocab.w2i, f)       # Only need to save w2i

def load_vocab(filename: str) -> Vocabulary:
    vocab = Vocabulary()
    with open(filename) as f:
        # Load w2i and generate i2w from it
        vocab.w2i = json.load(f)
        vocab.i2w = {id: word for word, id in vocab.w2i.items()}
    return vocab
```

We'll be using a word vector model called *skip-gram* that takes as input a word and generates probabilities for what words are likely to be seen near it. We will feed it training pairs `(word, nearby_word)` and try to minimize the `SoftmaxCrossEntropy` loss.

> **NOTE**
>
> Another common model, *continuous bag-of-words* (CBOW), takes the nearby words as the inputs and tries to predict the original word.

Let's design our neural network. At its heart will be an *embedding* layer that takes as input a word ID and returns a word vector. Under the covers we can just use a lookup table for this.

We'll then pass the word vector to a `Linear` layer with the same number of outputs as we have words in our vocabulary. As before, we'll use `softmax` to convert these outputs to probabilities over nearby words. As we use gradient descent to train the model, we will be updating the vectors in the lookup table. Once we've finished training, that lookup table gives us our word vectors.

Let's create that embedding layer. In practice we might want to embed things other than words, so we'll construct a more general `Embedding` layer. (Later we'll write a `TextEmbedding` subclass that's specifically for word vectors.)

In its constructor we'll provide the number and dimension of our embedding vectors, so it can create the embeddings (which will be standard random normals, initially):

```
from typing import Iterable
from scratch.deep_learning import Layer, Tensor, random_tensor, zeros_like

class Embedding(Layer):
    def __init__(self, num_embeddings: int, embedding_dim: int) -> None:
        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim

        # One vector of size embedding_dim for each desired embedding
        self.embeddings = random_tensor(num_embeddings, embedding_dim)
        self.grad = zeros_like(self.embeddings)

        # Save last input id
        self.last_input_id = None
```

In our case we'll only be embedding one word at a time. However, in other models we might want to embed a sequence of words and get back a sequence of word vectors. (For example, if we wanted to train the CBOW model described earlier.) So an alternative design would take sequences of word IDs. We'll stick with one at a time, to make things simpler.

```
    def forward(self, input_id: int) -> Tensor:
        """Just select the embedding vector corresponding to the input id"""
        self.input_id = input_id    # remember for use in backpropagation
```

```
        return self.embeddings[input_id]
```

For the backward pass we'll get a gradient corresponding to the chosen embedding vector, and we'll need to construct the corresponding gradient for `self.embeddings`, which is zero for every embedding other than the chosen one:

```
    def backward(self, gradient: Tensor) -> None:
        # Zero out the gradient corresponding to the last input.
        # This is way cheaper than creating a new all-zero tensor each time.
        if self.last_input_id is not None:
            zero_row = [0 for _ in range(self.embedding_dim)]
            self.grad[self.last_input_id] = zero_row

        self.last_input_id = self.input_id
        self.grad[self.input_id] = gradient
```

Because we have parameters and gradients, we need to override those methods:

```
    def params(self) -> Iterable[Tensor]:
        return [self.embeddings]

    def grads(self) -> Iterable[Tensor]:
        return [self.grad]
```

As mentioned earlier, we'll want a subclass specifically for word vectors. In that case our number of embeddings is determined by our vocabulary, so let's just pass that in instead:

```
class TextEmbedding(Embedding):
    def __init__(self, vocab: Vocabulary, embedding_dim: int) -> None:
        # Call the superclass constructor
        super().__init__(vocab.size, embedding_dim)

        # And hang onto the vocab
        self.vocab = vocab
```

The other built-in methods will all work as is, but we'll add a couple more methods specific to working with text. For example, we'd like to be able to retrieve the vector for a given word. (This is not part of the `Layer` interface, but we are always free to add extra methods to specific layers as we like.)

```
    def __getitem__(self, word: str) -> Tensor:
        word_id = self.vocab.get_id(word)
        if word_id is not None:
            return self.embeddings[word_id]
        else:
            return None
```

This dunder method will allow us to retrieve word vectors using indexing:

```
word_vector = embedding["black"]
```

And we'd also like the embedding layer to tell us the closest words to a given word:

```
    def closest(self, word: str, n: int = 5) -> List[Tuple[float, str]]:
        """Returns the n closest words based on cosine similarity"""
        vector = self[word]

        # Compute pairs (similarity, other_word), and sort most similar first
        scores = [(cosine_similarity(vector, self.embeddings[i]), other_word)
                  for other_word, i in self.vocab.w2i.items()]
        scores.sort(reverse=True)

        return scores[:n]
```

Our embedding layer just outputs vectors, which we can feed into a `Linear` layer.

Now we're ready to assemble our training data. For each input word, we'll choose as target words the two words to its left and the two words to its right.

Let's start by lowercasing the sentences and splitting them into words:

```
import re

# This is not a great regex, but it works on our data.
tokenized_sentences = [re.findall("[a-z]+|[.]", sentence.lower())
                       for sentence in sentences]
```

at which point we can construct a vocabulary:

```
# Create a vocabulary (that is, a mapping word -> word_id) based on our text.
vocab = Vocabulary(word
                   for sentence_words in tokenized_sentences
                   for word in sentence_words)
```

And now we can create training data:

```
from scratch.deep_learning import Tensor, one_hot_encode

inputs: List[int] = []
targets: List[Tensor] = []

for sentence in tokenized_sentences:
    for i, word in enumerate(sentence):          # For each word
        for j in [i - 2, i - 1, i + 1, i + 2]:   # take the nearby locations
            if 0 <= j < len(sentence):           # that aren't out of bounds
                nearby_word = sentence[j]        # and get those words.

                # Add an input that's the original word_id
                inputs.append(vocab.get_id(word))

                # Add a target that's the one-hot-encoded nearby word
                targets.append(vocab.one_hot_encode(nearby_word))
```

With the machinery we've built up, it's now easy to create our model:

```
from scratch.deep_learning import Sequential, Linear

random.seed(0)
EMBEDDING_DIM = 5  # seems like a good size

# Define the embedding layer separately, so we can reference it.
embedding = TextEmbedding(vocab=vocab, embedding_dim=EMBEDDING_DIM)

model = Sequential([
    # Given a word (as a vector of word_ids), look up its embedding.
```

```
    embedding,
    # And use a linear layer to compute scores for "nearby words."
    Linear(input_dim=EMBEDDING_DIM, output_dim=vocab.size)
])
```

Using the machinery from Chapter 19, it's easy to train our model:

```
from scratch.deep_learning import SoftmaxCrossEntropy, Momentum,
GradientDescent

loss = SoftmaxCrossEntropy()
optimizer = GradientDescent(learning_rate=0.01)

for epoch in range(100):
    epoch_loss = 0.0
    for input, target in zip(inputs, targets):
        predicted = model.forward(input)
        epoch_loss += loss.loss(predicted, target)
        gradient = loss.gradient(predicted, target)
        model.backward(gradient)
        optimizer.step(model)
    print(epoch, epoch_loss)            # Print the loss
    print(embedding.closest("black"))   # and also a few nearest words
    print(embedding.closest("slow"))    # so we can see what's being
    print(embedding.closest("car"))     # learned.
```

As you watch this train, you can see the colors getting closer to each other, the adjectives getting closer to each other, and the nouns getting closer to each other.

Once the model is trained, it's fun to explore the most similar words:

```
pairs = [(cosine_similarity(embedding[w1], embedding[w2]), w1, w2)
         for w1 in vocab.w2i
         for w2 in vocab.w2i
         if w1 < w2]
pairs.sort(reverse=True)
print(pairs[:5])
```

which (for me) results in:

```
[(0.9980283554864815, 'boat', 'car'),
 (0.9975147744587706, 'bed', 'cat'),
```

```
 (0.9953153441218054, 'seems', 'was'),
 (0.9927107440377975, 'extremely', 'quite'),
 (0.9836183658415987, 'bed', 'car')]
```

(Obviously *bed* and *cat* are not really similar, but in our training sentences they appear to be, and that's what the model is capturing.)

We can also extract the first two principal components and plot them:

```
from scratch.working_with_data import pca, transform
import matplotlib.pyplot as plt

# Extract the first two principal components and transform the word vectors
components = pca(embedding.embeddings, 2)
transformed = transform(embedding.embeddings, components)

# Scatter the points (and make them white so they're "invisible")
fig, ax = plt.subplots()
ax.scatter(*zip(*transformed), marker='.', color='w')

# Add annotations for each word at its transformed location
for word, idx in vocab.w2i.items():
    ax.annotate(word, transformed[idx])

# And hide the axes
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)

plt.show()
```

which shows that similar words are indeed clustering together (Figure 21-3):

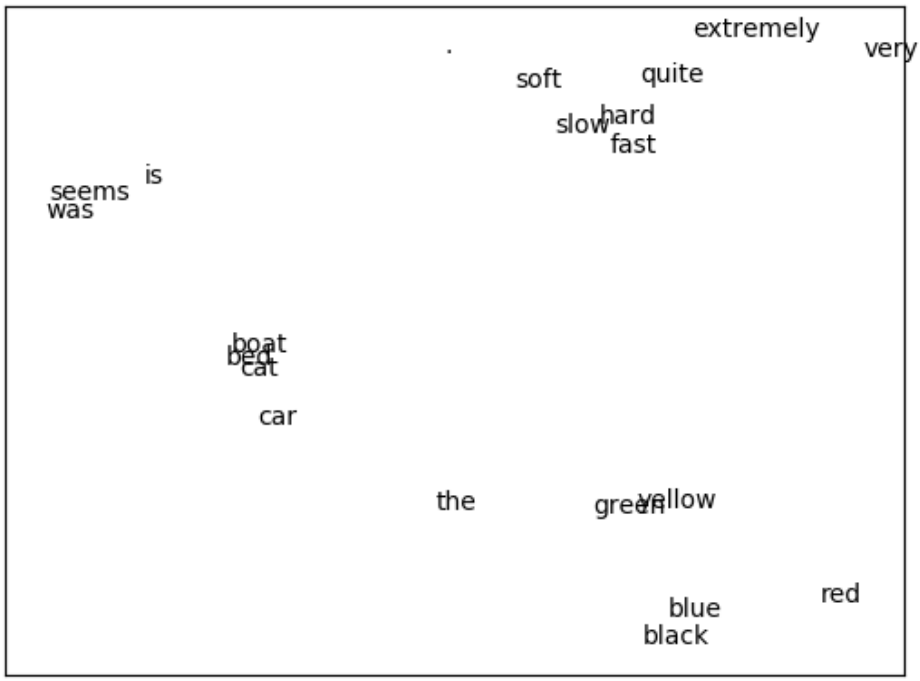

Figure 21-3. Word vectors

If you're interested, it's not hard to train CBOW word vectors. You'll have to do a little work. First, you'll need to modify the `Embedding` layer so that it takes as input a *list* of IDs and outputs a *list* of embedding vectors. Then you'll have to create a new layer (`Sum`?) that takes a list of vectors and returns their sum.

Each word represents a training example where the input is the word IDs for the surrounding words, and the target is the one-hot encoding of the word itself.

The modified `Embedding` layer turns the surrounding words into a list of vectors, the new `Sum` layer collapses the list of vectors down to a single vector, and then a `Linear` layer can produce scores that can be `softmax`ed to get a distribution representing "most likely words, given this context."