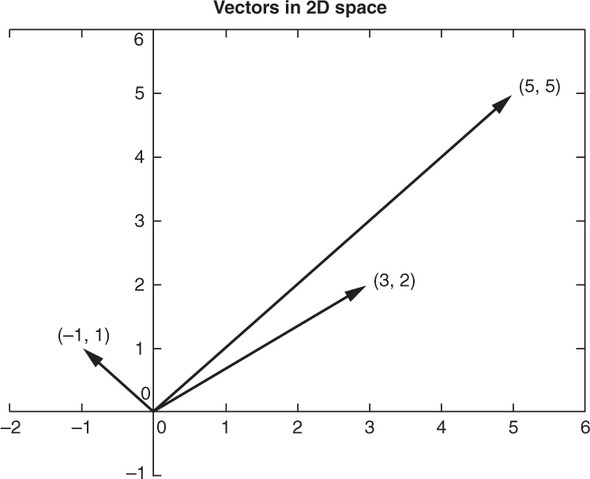
Vector- Is an ordered sequence of numbers

Every vector lives in a multidimensional vector space, as a single point. Embeddings are systematic, well-crafted procedures for projecting ('embedding') input data into such a space.

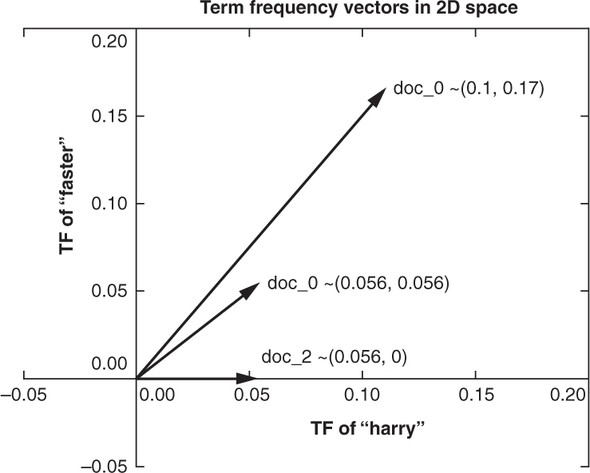
is one way to draw the 2D vectors (5, 5), (3, 2), and (-1, 1). The head of a vector (represented by the pointy tip of an arrow) is used to identify a location in a vector space. So the vector heads in this diagram will be at those three pairs of coordinates. The tail of a position vector (represented by the “rear” of the arrow) is always at the origin, or (0, 0)



The most simple representational embedding for mapping text to vectors is the one-hot embedding. Given a lexicon of N entries (like characters or words), we represent every word as a vector of N-1 zeros and a single, distinct 'one':

For a natural language document vector space, the dimensionality of your vector space is the count of the number of distinct words that appear in the entire corpus. For TF (and TF-IDF to come), sometimes we call this dimensionality capital letter “K.” This number of distinct words is also the vocabulary size of your corpus, so in an academic paper it’ll usually be called “|V|.” You can then describe each document within this K-dimensional vector space by a K-dimensional vector. K = 18 in your three-document corpus about Harry and Jill. Because humans can’t easily visualize spaces of more than three dimensions, let’s set aside most of those dimensions and look at two for a moment, so you can have a visual representation of the vectors on this flat page you’re reading. So in [figure 3.2](https://learning.oreilly.com/library/view/natural-language-processing/9781617294631/kindle_split_013.html#ch03fig02), K is reduced to two for a two-dimensional view of the 18-dimensional Harry and Jill vector space.

##### Figure 3.2. 2D term frequency vectors



K-dimensional vectors work the same way, just in ways you can’t easily visualize. Now that you have a representation of each document and know they share a common space, you have a path to compare them. You could measure the Euclidean distance between the vectors by subtracting them and computing the length of the distance between them, which is called the 2-norm distance. It’s the distance a “crow” would have to fly (in a straight line) to get from a location identified by the tip (head) of one vector and the location of the tip of the other vector. Check out [appendix C](https://learning.oreilly.com/library/view/natural-language-processing/9781617294631/kindle_split_040.html#app03) on linear algebra to see why this is a bad idea for word count (term frequency) vectors.

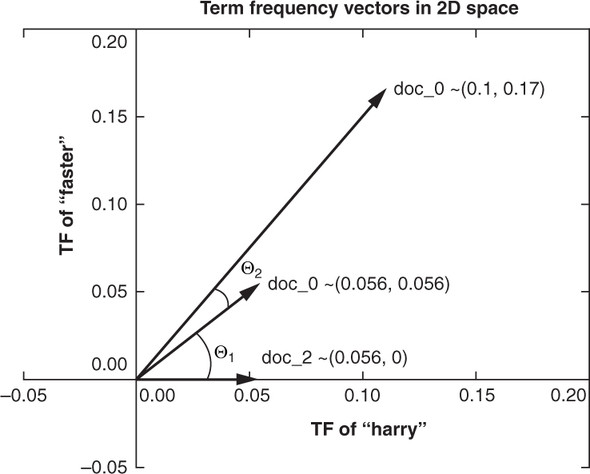
Two vectors are “similar” if they share similar direction. They might have similar magnitude (length), which would mean that the word count (term frequency) vectors are for documents of about the same length. But do you care about document length in your similarity estimate for vector representations of words in documents? Probably not. You’d like your estimate of document similarity to find use of the same words about the same number of times in similar proportions. This accurate estimate would give you confidence that the documents they represent are probably talking about similar things.

Cosine similarity is merely the cosine of the angle between two vectors (theta), shown in [figure 3.3](https://learning.oreilly.com/library/view/natural-language-processing/9781617294631/kindle_split_013.html#ch03fig03), which can be calculated from the Euclidian dot product using

* A ⋅ B = |*A*| |*B*| \* cos Θ

Cosine similarity is efficient to calculate because the dot product doesn’t require evaluation of any trigonometric functions. In addition, cosine similarity has a convenient range for most machine learning problems: -1 to +1.

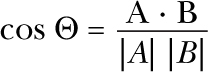
##### Figure 3.3. 2D thetas



In Python this would be

a.dot(b) == np.linalg.norm(a) \* np.linalg.norm(b) / np.cos(theta)

Solving this relationship for cos(theta), you can derive the cosine similarity using



Or you can do it in pure Python without numpy, as in the following listing.

##### Listing 3.1. Compute cosine similarity in python

>>> import math

>>> def cosine\_sim(vec1, vec2):

... """ Let's convert our dictionaries to lists for easier matching."""

... vec1 = [val for val in vec1.values()]

... vec2 = [val for val in vec2.values()]

...

... dot\_prod = 0

... for i, v in enumerate(vec1):

... dot\_prod += v \* vec2[i]

...

... mag\_1 = math.sqrt(sum([x\*\*2 for x in vec1]))

... mag\_2 = math.sqrt(sum([x\*\*2 for x in vec2]))

...

... return dot\_prod / (mag\_1 \* mag\_2)

So you need to take the dot product of two of your vectors in question—multiply the elements of each vector pairwise—and then sum up those products. You then divide by the norm (magnitude or length) of each vector. The vector norm is the same as its Euclidean distance from the head to the tail of the vector—the square root of the sum of the squares of its elements. This *normalized dot product*, like the output of the cosine function, will be a value between -1 and 1. It’s the cosine of the angle between these two vectors. This value is the same as the portion of the longer vector that’s covered by the shorter vector’s perpendicular projection onto the longer one. It gives you a value for how much the vectors point in the same direction.

A cosine similarity of *1* represents identical normalized vectors that point in exactly the same direction along all dimensions. The vectors may have different lengths or magnitudes, but they point in the same direction. Remember you divided the dot product by the norm of each vector, and this can happen before or after the dot product. So the vectors are normalized so they both have a length of 1 as you do the dot product. So the closer a cosine similarity value is to 1, the closer the two vectors are in angle. For NLP document vectors that have a cosine similarity close to 1, you know that the documents are using similar words in similar proportion. So the documents whose document vectors are close to each other are likely talking about the same thing.

A cosine similarity of *0* represents two vectors that share no components. They are orthogonal, perpendicular in all dimensions. For NLP TF vectors, this situation occurs only if the two documents share no words in common. Because these documents use completely different words, they must be talking about completely different things. This doesn’t necessarily mean they have different meanings or topics, just that they use completely different words.

A cosine similarity of *-1* represents two vectors that are anti-similar, completely opposite. They point in opposite directions. This can never happen for simple word count (term frequency) vectors or even normalized TF vectors (which we talk about later). Counts of words can never be negative. So word count (term frequency) vectors will always be in the same “quadrant” of the vector space. None of the term frequency vectors can sneak around into one of the quadrants behind the tail of the other vectors. None of your term frequency vectors can have components (word frequencies) that are the negative of another term frequency vector, because term frequencies just can’t be negative.

You won’t see any negative cosine similarity values for pairs of vectors for natural language documents in this chapter. But in the next chapter, we develop a concept of words and topics that are “opposite” to each other. And this will show up as documents, words, and topics that have cosine similarities of less than zero, or even *-1*.

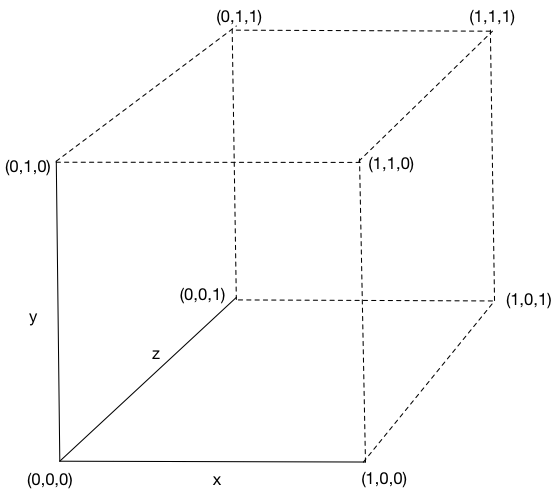
There’s an interesting consequence of the way you calculated cosine similarity. If two vectors or documents have a cosine similarity of *-1* (are opposites) to a third vector, they must be perfectly similar to each other. They must be exactly the same vectors. But the documents those vectors represent may not be exactly the same. Not only might the word order be shuffled, but one may be much longer than the other, if it uses the same words in the same proportion.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| the | Cat | sat | on | a | mat |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 |

Ideally, words that are related together should lie close to each other in such a vector space. 'Close' means here: near each other, as measured by a distance function. One such function is the Euclidean distance. This distance measure computes the length of a straight line between two points in an Euclidean space, using the Pythagorean algorithm. Euclidean space is a space with a finite number of dimensions. Points in Euclidean space are specified by coordinates, for every dimension. So, a vector

* (0,1,0)

is a unique coordinate of a point in three dimensions. The following figure displays a 3-dimensional Euclidean space. Notice how the 23 (8) different coordinates correspond to the corners of the cube.



*Euclidean distance* is the distance you are talking about for 2D vectors when you say “as the crow flies.” It’s the straight line distance between the two points defined by your vectors (the “tips” or “heads” of those vectors).

Euclidean distance is also called L2 norm, because it’s the length of the vector difference between two vectors. The “L” in L2 stands for length. The “2” in L2 represents the exponent (squaring) of the dimensions of the difference vector before these values ares summed (and before the square root of the sum).

Imagine these vectors represented blocks and floors in Manhattan for two people: the difference would be the exact directions you’d need to go from one to the other. If you were on the third floor of an apartment on the corner of 1st Street and 2nd Ave, your coordinates in street, avenue, floor coordinates would be [1, 2, 3], just like in the example. And if your Python mentor was on the sixth floor of an apartment on the corner of 4th Street and 5th Ave, her coordinates would be [4, 5, 6]. So the difference between those vectors ([3, 3, 3]) would mean that you’d have to walk three blocks north, three blocks east, and three floors up to reach her apartment. Actually, vectors and math don’t care about pesky details like gravity

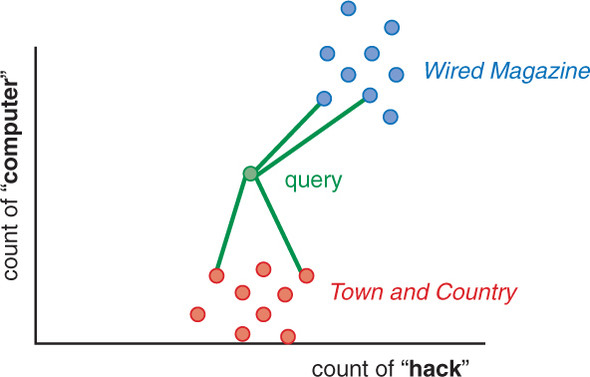
Euclidean distance is also called the RSS distance, which stands for the root sum square distance or difference, which means:

euclidean\_distance = np.sqrt(((vector1 - vector2) \*\* 2).sum())

Let’s say we have 2D term frequency (bag-of-word) vectors that count the occurrences of the words “hack” and “computer” in articles from two publications, *Wired Magazine* and *Town and Country*. And we want to be able to query that set of articles while researching something to find some articles about a particular topic. The query string has both the words “hacking” and “computers” in it. Our query string word vector is [1, 1] for the words “hack” and “computer” because our query tokenized and stemmed the words that way (see [chapter 2](https://learning.oreilly.com/library/view/natural-language-processing/9781617294631/kindle_split_012.html#ch02)).

Now which articles would you say are closest to our query in Euclidean distance? Euclidean distance is the length of the four lines in [figure C.1](https://learning.oreilly.com/library/view/natural-language-processing/9781617294631/kindle_split_041.html#app03fig01). They look pretty similar don’t they. How would you fix this problem so that your search engine returns some useful articles for this query?

**Figure C.1. Measuring Euclidean distance**



Another adjustment to our distance calculation makes our distance value even more useful. Cosine distance is the inverse of the cosine similarity (cosine\_distance = 1 - cosine\_similarity). Cosine similarity is the cosine of the angle between two vectors. So in this example, the angle between the TF vector for this query string and the vector for Wired Magazine articles would be much smaller than the angle between the query and the Town and Country articles. This is what we want. Because a query about “hacking computers” should give us Wired Magazine articles and not articles about recreational activities like horse riding (“hacking”)[2], duck hunting, dinner parties, and rustic interior design. 2 See the equestrian use of the word “hack” in the Wikipedia article “Hack (horse)” (https://en.wikipedia.org/wiki/Hack\_%28horse%29). This is efficiently computed as the dot product of two normalized vectors, vectors whose values have all been divided by the length of the vector, as shown in the following listing.

This is efficiently computed as the dot product of two normalized vectors, vectors whose values have all been divided by the length of the vector,

The cosine similarity between our query TF vector and these other two TF vectors (cosine of the angle between them) is

The cosine distance between our query and these two TF vectors is one minus the cosine similarity.

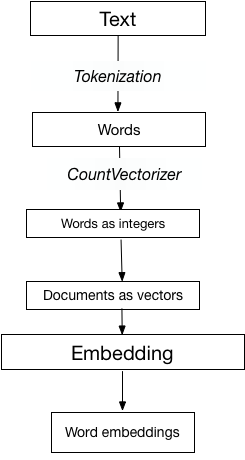
This is why cosine similarity is used for TF vectors in NLP: It’s easy to compute (just multiplication and addition). It has a convenient range (-1 to +1). Its inverse (cosine distance) is easy to compute (1 - cosine\_similarity). Its inverse (cosine distance) is bounded (0 to +2). However, cosine distance has one disadvantage compared to Euclidean distance: it isn’t a real distance metric because the triangle inequality doesn’t hold.[3] That means that if the word vector for “red” has a cosine distance of 0.5 from “car” and 0.3 from “apple,” “apple” might be much further away than 0.8 from “car.” The triangle inequality is mainly important when you want to use cosine distances to try to prove something about some vectors. That’s rarely the case in real-world NLP.

**Word Embeddings**

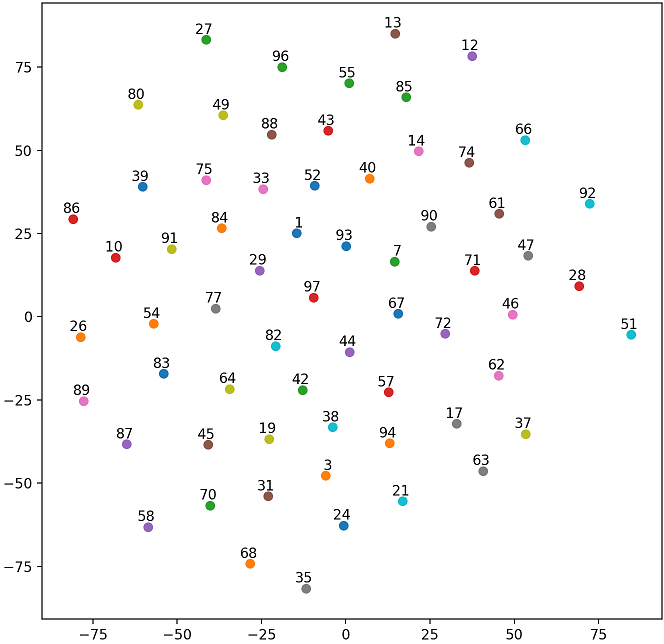
**Embeddings**

Embeddings are procedures for converting input data into vector representations

Word embeddings are basically a form of word representation that bridges the human understanding of language to that of a machine. Word embeddings are distributed representations of text in an n-dimensional space

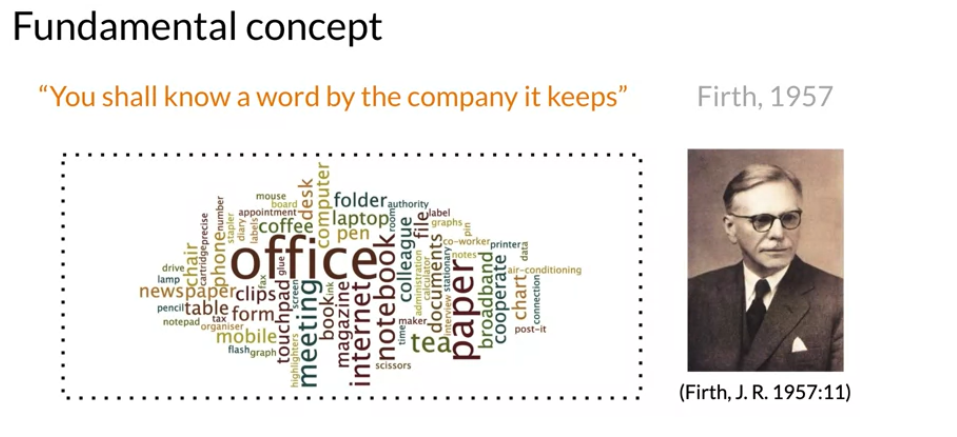


Representing a set of documents as vectors of integers, we can create a standard Embedding of these documents similar to the Embedding for random integers.



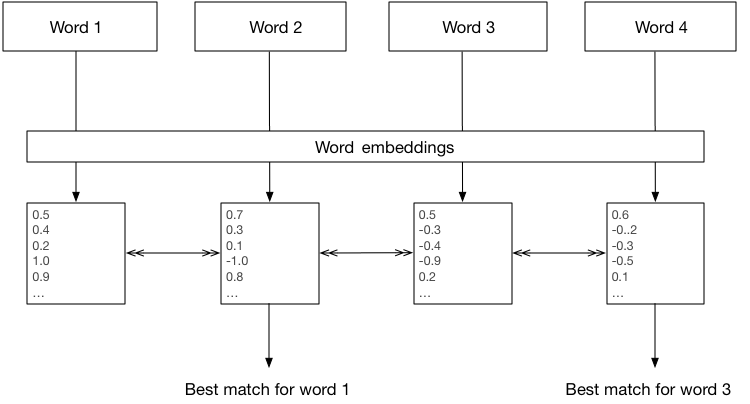
Word2Vec

The word2vec algorithm, discussed in Chapter 2, takes word context information into account for establishing relations between words. The algorithm we implemented in the previous section did not remotely do such a thing. It optimized the vector representation of words such that the accuracy on the sentiment labeling task was maximized. But it did not care a bit about establishing relations between words that shared similar contexts. The vector representations we obtain with word2vec should, hopefully, express a more interpretable form of similarity between words, based on shared contexts: according to word2vec, two words are more similar if they share similar contexts, which is a modern incarnation of the tenet of the British linguist John Rupert Firth



“Word2vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located in close proximity to one another in the space.”

the word2vec algorithm has two more or less equivalent implementations: predicting words from contexts, or contexts from words.



* the restaurant has a terrible ambiance and the food is awful

we want to predict if the words 'terrible' for 'restaurant' and 'awful' for 'food' are valid context words. Words with similar context predictions are judged to be similar by word2vec: they produce comparable vector representations.

WordNet, a thesaurus containing lists of synonym sets and hypernyms (“is a” relationships)

noun: good

noun: good, goodness

noun: good, goodness

noun: commodity, trade\_good, good

adj: good

adj (sat): full, good

adj: good

adj (sat): estimable, good, honorable, respectable

adj (sat): beneficial, good

adj (sat): good

adj (sat): good, just, upright

…

adverb: well, good

adverb: thoroughly, soundly, good

Problems with WordNet

Great as a resource but missing nuance

e.g. “proficient” is listed as a synonym for “good”. This is only correct in some contexts.

Missing new meanings of words e.g., wicked, badass, nifty, wizard, genius, ninja, bombast

Impossible to keep up-to-date!

Vector dimension = number of words in vocabulary (e.g., 500,000)

Applications of Word2Vec

# **Analyzing Survey Responses**

Word2Vec can be used to get actionable metrics from thousands of customers reviews. Word embeddings prove invaluable in such cases. Vector representation of words trained on (or adapted to) survey data-sets can help embed complex relationship between the responses being reviewed and the specific context within which the response was made. Machine learning algorithms can leverage this information to identify actionable insights for your business/product

# **Music/Video Recommendation System**

Word2Vec. The algorithm interprets a user’s listening queue as a sentence with each song considered as a word in the sentence. When a Word2Vec model is trained on a such a dataset what we mean is that each song that the user has listened to in the past and the song one is listening to at present somehow belong to the same context. Word2Vec accurately represents each song with a vector of coordinates that maps the context in which the song or the video is played.