Vector Space

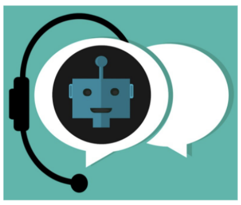
Vector Space Models

Vector spaces are fundamental in many applications in NLP. If you were to represent a word, document, tweet, or any form of text, you will probably be encoding it as a vector. These vectors are important in tasks like information extraction, machine translation, and chatbots. Vector spaces could also be used to help you identify relationships between words as follows:

- You eat *cereal* from a *bowl*
- You *buy* something and someone else *sells* it

Information Extraction

Machine Translation



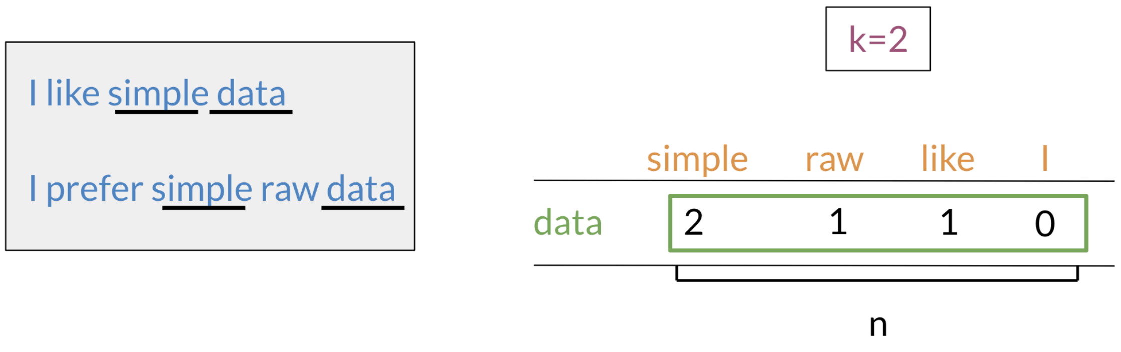
Chatbots

The famous quote by Firth says, **"You shall know a word by the company it keeps".** When learning these vectors, you usually make use of the neighboring words to extract meaning and information about the center word. If you were to cluster these vectors together, as you will see later in this specialization, you will see that adjectives, nouns, verbs, etc. tend to be near one another. Another cool fact, is that synonyms and antonyms are also very close to one another. This is because you can easily interchange them in a sentence and they tend to have similar neighboring words!

Word by Word and Word by Doc.

Word by Word Design

We will start by exploring the word by word design. Assume that you are trying to come up with a vector that will represent a certain word. One possible design would be to create a matrix where each row and column corresponds to a word in your vocabulary. Then you can iterate over a document and see the number of times each word shows up next each other word. You can keep track of the number in the matrix. In the video I spoke about a parameter *K*. You can think of *K* as the bandwidth that decides whether two words are next to each other or not.

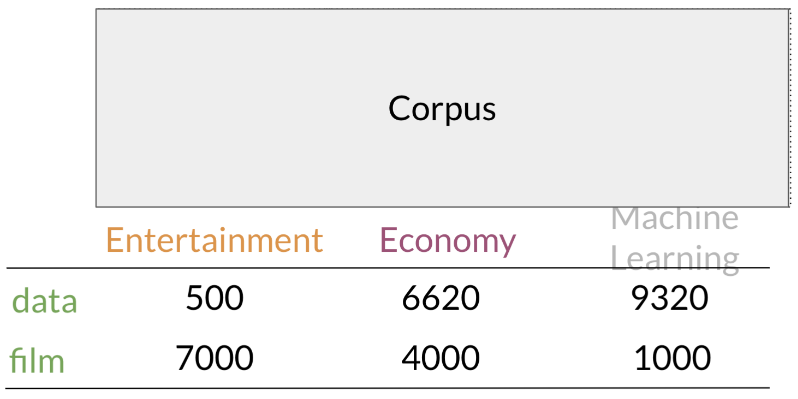


	simple	raw	like	I
data	2	1	1	0

In the example above, you can see how we are keeping track of the number of times words occur together within a certain distance *k*. At the end, you can represent the word data, as a vector *v*=[2,1,1,0].

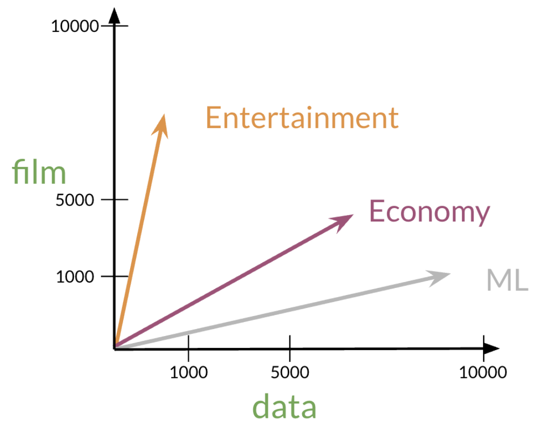
Word by Document Design

You can now apply the same concept and map words to documents. The rows could correspond to words and the columns to documents. The numbers in the matrix correspond to the number of times each word showed up in the document.



	Entertainment	Economy	Machine Learning
data	500	6620	9320
film	7000	4000	1000

You can represent the entertainment category, as a vector *v*=[500,7000]. You can then also compare categories as follows by doing a simple plot.

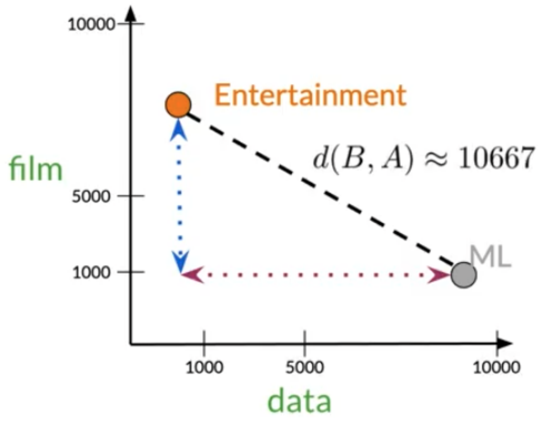


	Entertainment	Economy	ML
data	500	6620	9320
film	7000	4000	1000

Measures of "similarity:"
Angle
Distance

Later this week, you will see how you can use the angle between two vectors to measure similarity.

Euclidean distance



Corpus A: (500,7000)

Corpus B: (9320,1000)

$$d(B,A) = \sqrt{\underline{(B_1 - A_1)^2} + \underline{(B_2 - A_2)^2}}$$

$$c^2 = a^2 + b^2$$

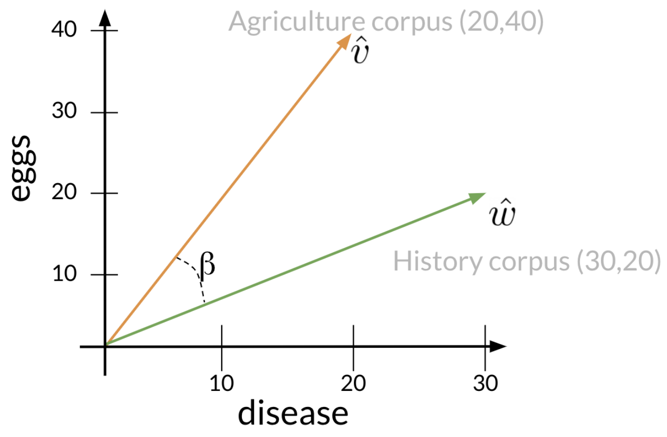
$$d(B,A) = \sqrt{(8820)^2 + (-6000)^2}$$

	data	$\vec{w}$ boba	$\vec{v}$ ice-cream
AI	6	0	1
drinks	0	4	6
food	0	6	8

$$= \sqrt{(1-0)^2 + (6-4)^2 + (8-6)^2}$$

$$= \sqrt{1+4+4} = \sqrt{9} = 3$$

Cosine Similarity



$$\hat{v} \cdot \hat{w} = \|\hat{v}\|\|\hat{w}\| \cos(\beta)$$

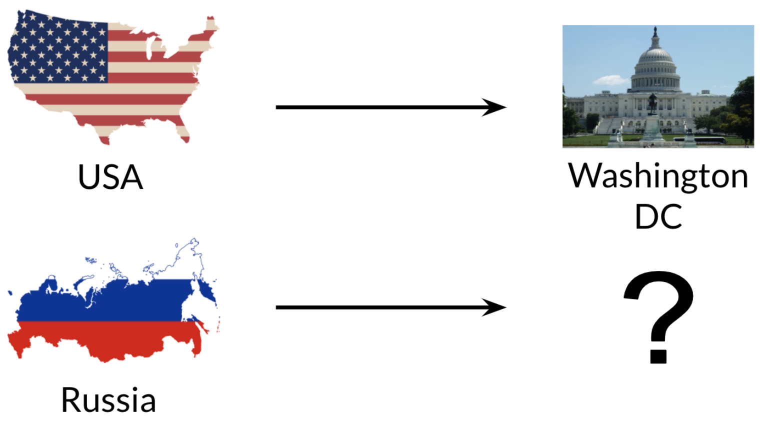
$$\boxed{\cos(\beta) = \frac{\hat{v} \cdot \hat{w}}{\|\hat{v}\|\|\hat{w}\|}}$$

$$= \frac{(20 \times 30) + (40 \times 20)}{\sqrt{20^2 + 40^2} \times \sqrt{30^2 + 20^2}}$$

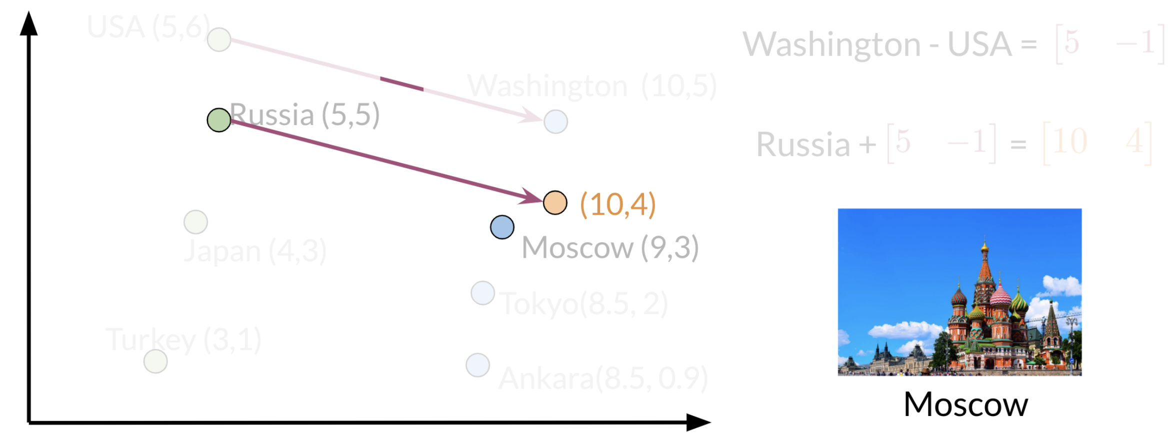
$$= 0.87$$

Manipulating Words in Vector Spaces

You can use word vectors to actually extract patterns and identify certain structures in your text. For example:



You can use the word vector for Russia, USA, and DC to actually compute a **vector** that would be very similar to that of Moscow. You can then use cosine similarity of the **vector** with all the other word vectors you have and you can see that the vector of Moscow is the closest. Isn't that cool?



Note that the distance (and direction) between a country and its capital is relatively the same. Hence manipulating word vectors allows you identify patterns in the text.

Visualization and PCA

Principal component analysis is an unsupervised learning algorithm which can be used to reduce the dimension of your data. As a result, it allows you to visualize your data. It tries to combine variances across features. Here is a concrete example of PCA:

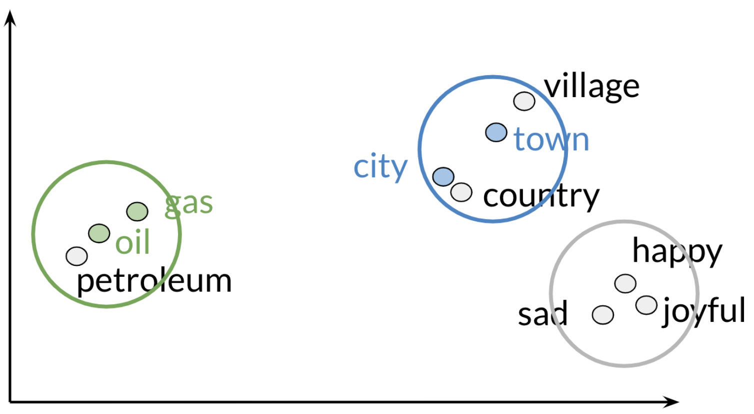
Visualization of word vectors

$$\overbrace{\hspace{10em}}^{d > 2}$$

oil	0.20	...	0.10
gas	2.10	...	3.40
city	9.30	...	52.1
town	6.20	...	34.3

How can you visualize if your representation captures these relationships?

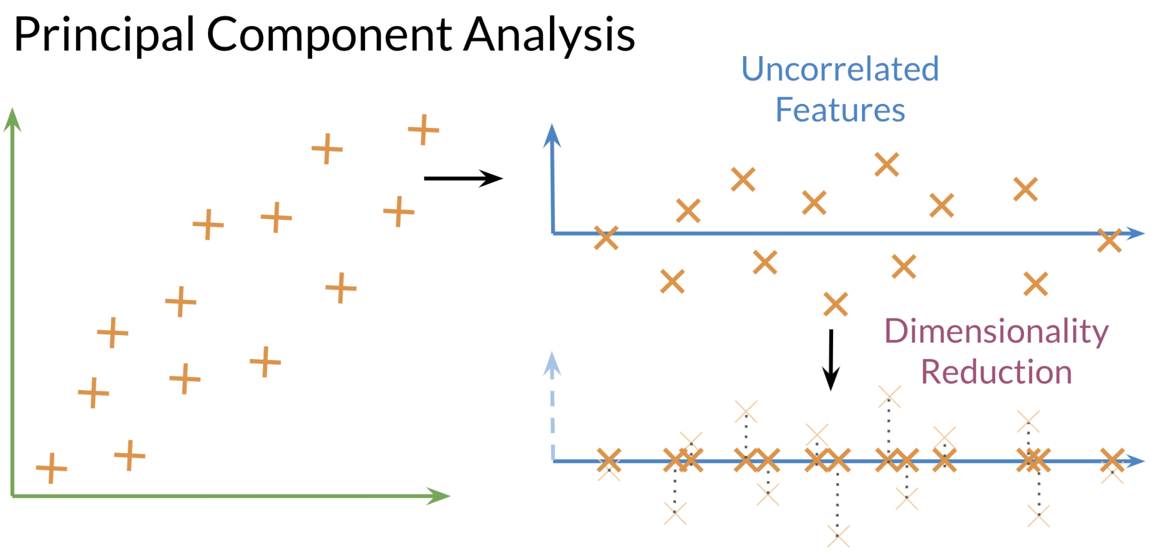
Note that when doing PCA on this data, you will see that oil & gas are close to one another and town & city are also close to one another. To plot the data you can use PCA to go from *d*>2 dimensions to *d*=2.



Those are the results of plotting a couple of vectors in two dimensions. Note that words with similar part of speech (POS) tags are next to one another. This is because many of the training algorithms learn words by identifying the neighboring words. Thus, words with similar POS tags tend to be found in similar locations. An interesting insight is that synonyms and antonyms tend to be found next to each other in the plot. Why is that the case?

PCA algorithm

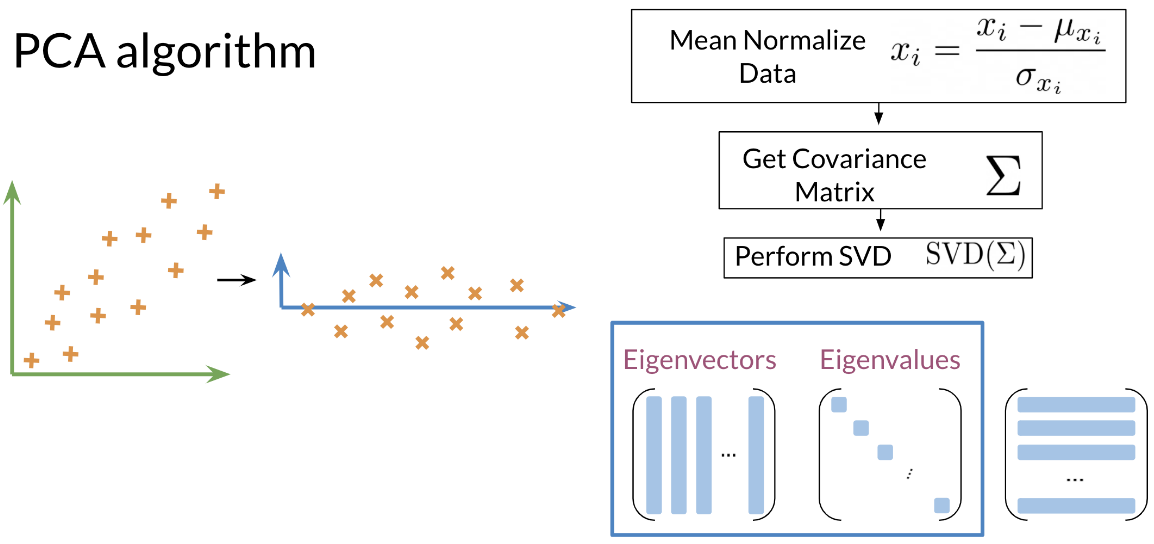
PCA is commonly used to reduce the dimension of your data. Intuitively the model collapses the data across principal components. You can think of the first principal component (in a 2D dataset) as the line where there is the most amount of variance. You can then collapse the data points on that line. Hence you went from 2D to 1D. You can generalize this intuition to several dimensions.



Eigenvector: the resulting vectors, also known as the uncorrelated features of your data

Eigenvalue: the amount of information retained by each new feature. You can think of it as the variance in the eigenvector.

Also each **eigenvalue** has a corresponding eigenvector. The eigenvalue tells you how much variance there is in the eigenvector. Here are the steps required to compute PCA:


PCA algorithm
Mean Normalize Data $x_i = \frac{x_i - \mu_{x_i}}{\sigma_{x_i}}$
Get Covariance Matrix Σ
Perform SVD $\mathrm{SVD}(\Sigma)$
Eigenvectors
Eigenvalues
...

<u>**Steps to Compute PCA:**</u>

- Mean normalize your data
- Compute the covariance matrix
- Compute SVD on your covariance matrix. This returns $[USV]=svd(\Sigma)$. The three matrices U, S, V are drawn above. U is labelled with eigenvectors, and S is labelled with eigenvalues.
- You can then use the first n columns of vector U, to get your new data by multiplying $XU[:,0:n]$

```
def compute_pca(X, n_components=2):
    """
    Input:
        X: of dimension (m,n) where each row corresponds to a
        n_components: Number of components you want to keep.
    Output:
        X_reduced: data transformed in 2 dims/columns + regen
    pass in: data as 2D NumPy array
    """
    # mean center the data
    X_demeaned = X - np.mean(X, axis=0)

    # calculate the covariance matrix
    covariance_matrix = np.cov(X_demeaned, rowvar=False)

    # calculate eigenvectors & eigenvalues of the covariance 
    eigen_vals, eigen_vecs = np.linalg.eigh(covariance_matrix

    # sort eigenvalue in increasing order (get the indices fr
    idx_sorted =  np.argsort(eigen_vals)

    # reverse the order so that it's from highest to lowest.
    idx_sorted_decreasing = idx_sorted[::-1]

    # sort the eigen values by idx_sorted_decreasing
    eigen_vals_sorted = eigen_vals[idx_sorted_decreasing]

    # sort eigenvectors using the idx_sorted_decreasing indic
    eigen_vecs_sorted = eigen_vecs[:, idx_sorted_decreasing]

    # select the first n eigenvectors (n is desired dimension
    # of rescaled data array, or n_components)
    eigen_vecs_subset = eigen_vecs_sorted[:, :n_components]

    # transform the data by multiplying the transpose of the 
    # Then take the transpose of that product.
    X_reduced = np.dot(X_demeaned, eigen_vecs_subset)

    return X_reduced
```