Text Similarities : Estimate the degree of similarity between two texts

Adrien Sieg Follow
Jul 4, 2018 · 29 min read ★

Note to the reader: Python code is shared at the end

We always need to **compute the similarity in meaning between texts**.

- **Search engines** need to model the relevance of a document to a query, beyond the overlap in words between the two. For instance, **question-and-answer** sites such as Quora or Stackoverflow need to determine whether a question has already been asked before.

- In **legal matters**, text similarity task allow to mitigate risks on a new contract, based on the assumption that if a new contract is similar to a existent one that has been proved to be resilient, the risk of this new contract being the cause of financial loss is minimised. Here is the principle of **Case Law principle**. **Automatic linking of related documents ensures that identical situations are treated similarly in every case**. Text similarity foster fairness and equality. Precedence retrieval of **legal documents** is an information retrieval task to retrieve prior case documents that are related to a given case document.

- **In customer services**, AI system should be able to understand semantically similar queries from users and provide a uniform response. The emphasis on semantic similarity aims to create a system that recognizes language and word patterns to craft responses that are similar to how a human conversation works. For example, if the user asks **“What has happened to my delivery?”** or **“What is wrong with my shipping?”**, the user will expect the same response.

What is text similarity?

Text similarity has to determine how 'close' two pieces of text are both in surface closeness [**lexical similarity**] and meaning [**semantic similarity**].

For instance, how similar are the phrases ***"the cat ate the mouse"*** with ***"the mouse ate the cat food"*** by just looking at the words?

- **On the surface**, if you consider only **word level similarity**, these two phrases appear very similar as 3 of the 4 unique words are an exact overlap. It typically does not take into account the actual meaning behind words or the entire phrase in context.
- Instead of doing a **word for word comparison**, we also need to pay attention to **context** in order to capture more of the **semantics**. To consider **semantic similarity** we need to focus on **phrase/paragraph levels** (or lexical chain level) where a piece of text is broken into a relevant group of related words prior to computing similarity. We know that while the words significantly overlap, these two phrases actually have **different meaning.**

There is a **dependency structure** in any sentences:

> **mouse** *is the object of* **ate** *in the first case and* ***food*** *is the object of* ***ate*** *in the second case*

Since **differences in word order** often go hand in hand with **differences in meaning** (compare `the dog bites the man` with `the man bites the dog`), we'd like our sentence embeddings to **be sensitive to this variation**.

But lucky we are, word vectors have evolved over the years **to know the difference** between `record the play` vs `play the record`

What is our winning strategy?

The **big idea** is that you **represent documents as vectors of features**, and **compare documents by measuring the distance between these features**. There are **multiple ways** to **compute features** that capture the **semantics of documents** and multiple algorithms to capture **dependency structure of documents** to focus on meanings of documents.

Supervised training can help sentence embeddings learn the meaning of a sentence more directly.

Here we'll compare the most popular ways of **computing sentence similarity** and investigate **how they perform**.

- **Jaccard Similarity** 😟😟😟
- Different embeddings**+ K-means** 😟😟
- Different embeddings**+ Cosine Similarity** 😟
- **Word2Vec + Smooth Inverse Frequency** + **Cosine Similarity** 😋
- Different embeddings**+LSI + Cosine Similarity** 😟
- Different embeddings**+ LDA** + **Jensen-Shannon distance** 😋
- Different embeddings**+ Word Mover Distance** 😋😋
- Different embeddings**+ Variational Auto Encoder (VAE)** 😋 😋
- Different embeddings**+ Universal sentence encoder** 😋😋
- Different embeddings**+ Siamese Manhattan LSTM** 😋😋😋
- BERT embeddings + Cosine Similarity ❤
- **Knowledge-based Measures** ❤

What do we mean by different embeddings?

We tried different word embedding in order to feed back our different ML/DL algorithms. Here is **our list of embeddings we tried —** *to access all code, you can visit my github repo.*

```
- Bag of Words (BoW)
- Term Frequency - Inverse Document Frequency (TF-IDF)
- Continuous BoW (CBOW) model and SkipGram model embedding(SkipGram)
- Pre-trained word embedding models :
     ->  Word2Vec (by Google)
     ->  GloVe (by Stanford)
     ->  fastText (by Facebook)
```

- **Poincarré** embedding
- **Node2Vec** embedding based on Random Walk and Graph

Word embedding is one of the most popular representation of document vocabulary. It is capable of capturing context of a word in a **document, semantic** and **syntactic similarity**, relation with other words, etc.

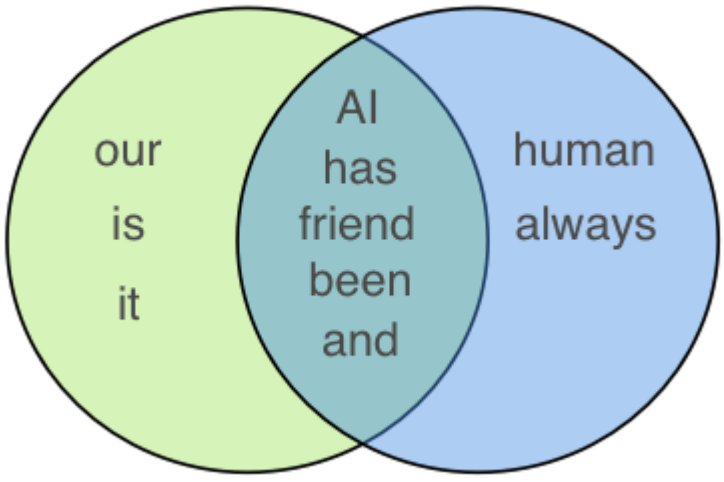
A very sexy approach [Knowledge-based Measures (wordNet)] [Bonus]

It is common to find in many sources (blogs etc) that **the first step to cluster text data is to transform text units to vectors**. This is not 100% true. But this step depends mostly on the **similarity measure** and the **clustering algorithm**. **Some of the best performing text similarity measures don't use vectors at all**. This is the case of the winner system in SemEval2014 sentence similarity task which uses lexical word alignment. However, vectors are more efficient to process and allow to benefit from existing ML/DL algorithms.

0. Jaccard Similarity ☹☹☹:

Jaccard similarity or intersection over union is defined as **size of intersection divided by size of union of two sets.** Let's take example of two sentences:

Sentence 1: AI is our friend and it has been friendly
Sentence 2: AI and humans have always been friendly



Jaccard Similarity Principle

In order to calculate similarity using Jaccard similarity, we will first perform **lemmatization** to reduce words to the same root word. In our case, “friend” and “friendly” will both become “friend”, “has” and “have” will both become “has”.

```
def jaccard_similarity(query, document):
    intersection = set(query).intersection(set(document))
    union = set(query).union(set(document))
    return len(intersection)/len(union)
```

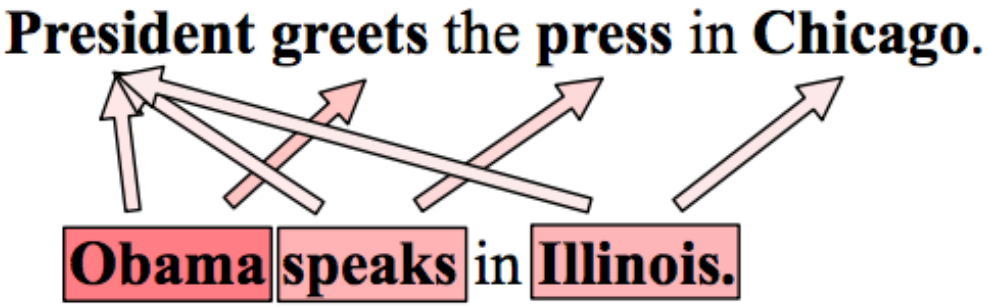
JaccardSimilarity.py hosted with ♡ by GitHub view raw

Jaccard Similarity Function

For the above two sentences, we get Jaccard similarity of $5/(5+3+2) = 0.5$ which is size of intersection of the set divided by total size of set.

Let’s take another example of two sentences having a similar meaning:

Sentence 1: President greets the press in Chicago
Sentence 2: Obama speaks in Illinois



Why Jaccard Similarity is not efficient?

My 2 sentences **have no common words** and will have **a Jaccard score of 0**. This is a **terrible distance** score because **the 2 sentences have very similar meanings**. Here Jaccard similarity is neither able to capture **semantic similarity** nor **lexical semantic** of these two sentences.

Moreover, this approach has an **inherent flaw**. That is, as the **size of the document increases**, the number of common words tend to increase even if the documents talk about different topics.

1. K-means and Hierarchical Clustering Dendrogram☹:

With K-mean related algorithms, we first need to convert sentences into vectors. Several ways to do that is to use :

- **Bag of words with either TF** (term frequency) called Count Vectorizer method
- **TF-IDF** (term frequency- inverse document frequency)
- **Word Embeddings** either coming from **pre-trained methods** such as Fastext, Glove or Word2Vec or **customized method** by using Continuous Bag of Words (CBoW) or Skip Gram models

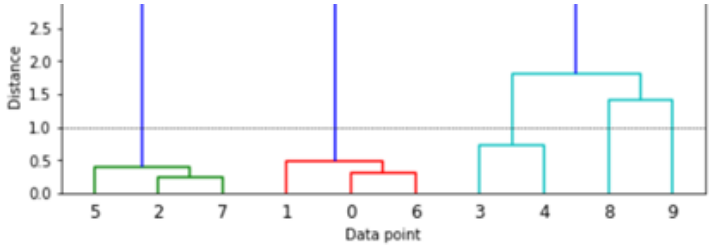
There are **two main difference between BoW** or **TF-IDF** in keeping with **word embeddings**:

- **BoW** or **TF-IDF** create one number per word while **word embeddings** typically creates one vector per word.
- **BoW** or **TF-IDF** is good for classification documents as a whole, but **word embeddings** is good for identifying contextual content

Let's consider several sentences in which we found our initial sentences which are "**President greets the press in Chicago"** and "**Obama speaks in Illinois"** with index 8 and 9 respectively.

```
corpus = ['The sky is blue and beautiful.',
 'Love this blue and beautiful sky!',
 'The quick brown fox jumps over the lazy dog.',
 "A king's breakfast has sausages, ham, bacon, eggs, toast and
beans",
 'I love green eggs, ham, sausages and bacon!',
 'The brown fox is quick and the blue dog is lazy!',
 'The sky is very blue and the sky is very beautiful today',
 'The dog is lazy but the brown fox is quick!',
 'President greets the press in Chicago',
 'Obama speaks in Illinois'
]
```

	Document	Category	ClusterLabel
0	The sky is blue and beautiful.	weather	2
1	Love this blue and beautiful sky!	weather	2



2	The quick brown fox jumps over the lazy dog.	animals	1
3	A king's breakfast has sausages, ham, bacon, eggs, toast and beans	food	3
4	I love green eggs, ham, sausages and bacon!	food	3
5	The brown fox is quick and the blue dog is lazy!	animals	1
6	The sky is very blue and the sky is very beautiful today	weather	2
7	The dog is lazy but the brown fox is quick!	animals	1
8	President greets the press in Chicago	politics	4
9	Obama speaks in Illinois	politics	4

It can be noted that k-means (and minibatch k-means) are very sensitive to feature scaling and that in this case the **IDF weighting helps improve the quality of the clustering.**

This improvement is not visible in the Silhouette Coefficient which is small for both as this measure seem to suffer from the phenomenon called **"Concentration of Measure"** or **"Curse of Dimensionality"** for **high dimensional datasets such as text data**.

Reducing the dimensionality of our document vectors by applying latent semantic analysis will be the solution.

However to overcome this big issue of dimensionality, there are measures such as **V-measure** and **Adjusted Rand Index** wich are information theoretic based evaluation scores: as they are only **based on cluster assignments** rather than **distances**, **hence not affected by the curse of dimensionality**.

Note: as k-means is optimizing a non-convex objective function, it will likely end up in a local optimum. Several runs with independent random init might be necessary to get a good convergence.

2. Cosine Similarity ☹:

Cosine similarity calculates similarity by measuring **the cosine of angle between two vectors**.

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Mathematically speaking, **Cosine similarity** is a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them. The cosine of 0° is 1, and it is less than 1 for any angle in the interval (0,π] radians. It is thus a judgment of **orientation** and **not magnitude**: two vectors with the same orientation have a cosine similarity of 1, two vectors oriented at 90° relative to each other have a similarity of 0, and two vectors diametrically opposed have a similarity of -1, independent of their magnitude.

*The cosine similarity is **advantageous** because even if the two similar documents are **far apart by the Euclidean distance (due to the size of the document)**, chances are they may **still be oriented closer together**. The smaller the angle, higher the cosine similarity.*

CountVectorizer Method + Cosine Similarity 😟

```
cosine_distance_countvectorizer_method(ss1 , ss2)
```

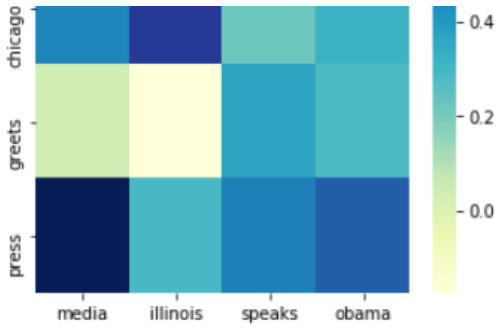
```
   Similarity of two sentences are equal to  37.8 %

0.6220355269907728
```

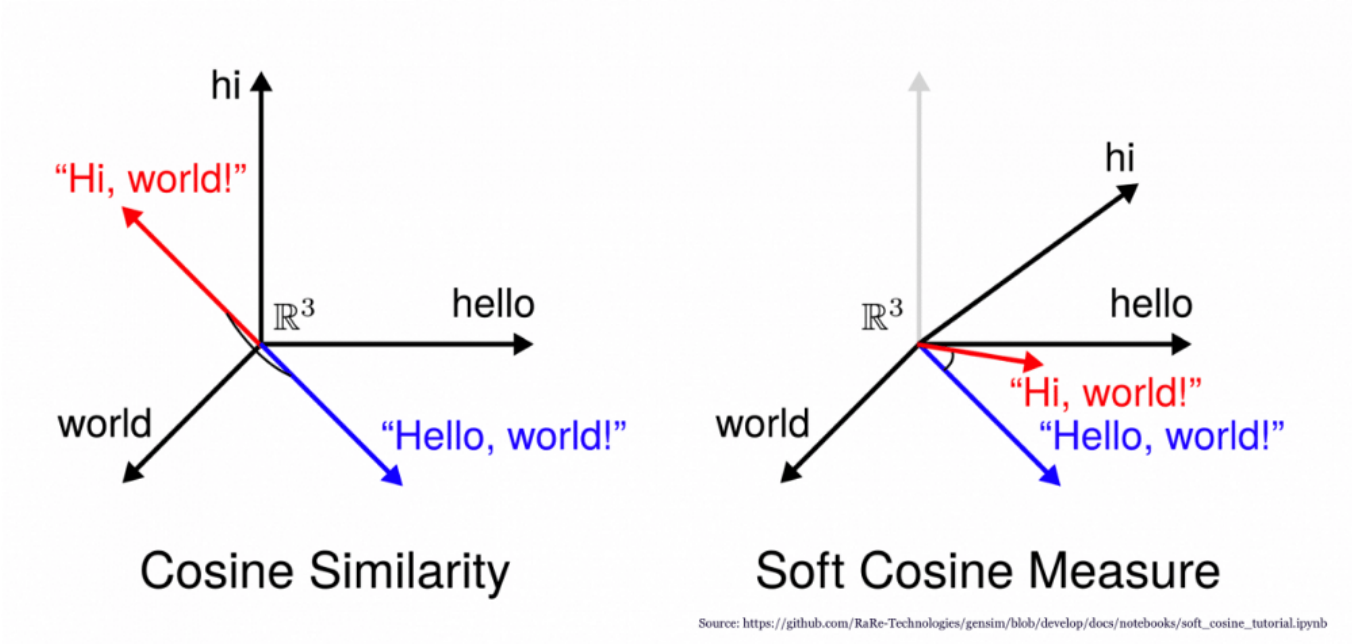
My sparse vectors for the 2 sentences **have no common words** and will have **a cosine distance of 0.622** — too much closer to 1. This is a **terrible distance** score because **the 2 sentences have very similar meanings**.

Pre-trained Method (such as Glove) + Cosine Similarity 😊

```
ss1 = 'The president greets the press in Chicago'
ss2 = 'Obama speaks to the media in Illinois'

model = loadGloveModel(gloveFile)
heat_map_matrix_between_two_sentences(ss1,ss2)
```

```
   Loading Glove Model
   Done. 400000  words loaded!
   Word Embedding method with a cosine distance asses that our two sentences are similar to 81.25 %
   None

<matplotlib.axes._subplots.AxesSubplot at 0x1a66016b5c0>
```





For Example, '*President*' vs '*Prime minister*', '*Food*' vs '*Dish*', '*Hi*' vs '*Hello*' should be considered similar. For this, converting the words into respective word vectors, and then, computing the similarities can address this problem.

```
def cosine_distance_wordembedding_method(s1, s2):
    import scipy
    vector_1 = np.mean([model[word] for word in preprocess(s1)],axis=0)
    vector_2 = np.mean([model[word] for word in preprocess(s2)],axis=0)
    cosine = scipy.spatial.distance.cosine(vector_1, vector_2)
    print('Word Embedding method with a cosine distance asses that our two sentences are similar
```

CosineDistance.py hosted with ♡ by GitHub view raw

Smooth Inverse Frequency

Taking the **average of the word embeddings in a sentence** (as we did just above) tends to give **too much weight to words** that are quite **irrelevant**, semantically

speaking. Smooth Inverse Frequency tries to solve this problem in two ways:

1. **Weighting**: SIF takes the weighted average of the word embeddings in the sentence. Every word embedding is weighted by `a/(a + p(w))`, where `a` is a parameter that is typically set to `0.001` and `p(w)` is the estimated frequency of the word in a reference corpus.

2. **Common component removal**: SIF computes the principal component of the resulting embeddings for a set of sentences. It then subtracts from these sentence embeddings their projections on their first principal component. This should remove variation related to frequency and syntax that is less relevant semantically.

SIF downgrades unimportant words such as `but`, `just`, etc., and keeps the information that contributes most to the semantics of the sentence.

3. Latent Semantic Indexing (LSI)

Not directly comparing the cosine similarity of bag-of-word vectors, but first **reducing the dimensionality of our document vectors** by applying latent semantic analysis.

A

M	D_1	D_2	D_3	D_4	D_5	D_6	$\cdots$	D_n
T_1	0.00060	0.00012	0.00003	0.00003	0.00333	0.00048	$\cdots$	a_{1n}
T_2	0	0	0	0	0	0	$\cdots$	a_{2n}
T_3	0	2.98862	0	0	0	1.49431	$\cdots$	a_{3n}
T_4	0	0	0	13.32555	0	0	$\cdots$	a_{4n}
T_5	0	0	0	0	0	0	$\cdots$	a_{5n}
T_6	1.03442	1.03442	0	0	0	3.10326	$\cdots$	a_{6n}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
T_m	a_{m1}	a_{m2}	a_{m3}	a_{m4}	a_{m5}	a_{m6}	$\cdots$	a_{mn}

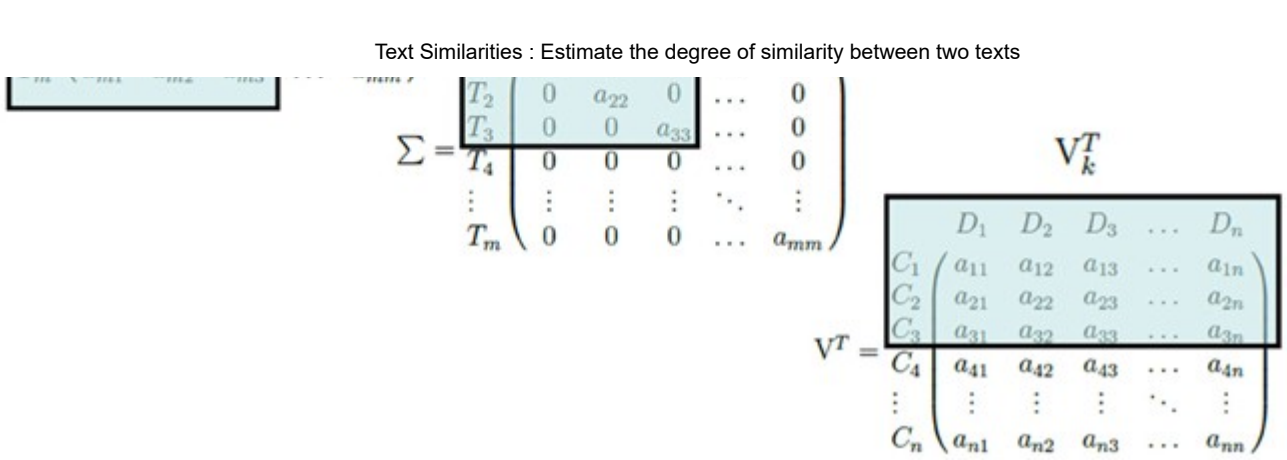
B

U_k

U =	C_1	C_2	C_3	$\cdots$	C_m
T_1	a_{11}	a_{12}	a_{13}	$\cdots$	a_{1m}
T_2	a_{21}	a_{22}	a_{23}	$\cdots$	a_{2m}
T_3	a_{31}	a_{32}	a_{33}	$\cdots$	a_{3m}
T_4	a_{41}	a_{42}	a_{43}	$\cdots$	a_{4m}
T_5	a_{51}	a_{52}	a_{53}	$\cdots$	a_{5m}
T_6	a_{61}	a_{62}	a_{63}	$\cdots$	a_{6m}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
T_m	a_{m1}	a_{m2}	a_{m3}	$\cdots$	a_{mm}

$\sum_k$

	D_1	D_2	D_3	$\cdots$	D_n
T_1	a_{11}	0	0	$\cdots$	0



It is often assumed that the **underlying semantic space of a corpus** is of a **lower dimensionality** than the **number of unique tokens**. Therefore, LSA applies **principal component analysis** on our vector space and only **keeps the directions in our vector space** that contain **the most variance** (i.e. those directions in the space that change most rapidly, and thus are assumed to contain more information). This is influenced by the `num_topics` parameters we pass to the `LsiModel` constructor.

Hypothesis **:** Suppose we have 3 documents :

- ***d1*** = "Shipment of gold damaged in a fire"
- ***d2*** = **"**Delivery of silver arrived in a silver truck"
- ***d3*** = **"**Shipment of gold arrived in a truck"

Goal of the game : We want to find the most similar doc when it comes to the query: **"gold silver truck"**

Method : Use Latent Semantic Indexing (LSI). We use the **term frequency** as **term weights** and **query weights**.

Step 0 : Pre-processing :

- Stop words were not ignored
- Text was tokenized and lowercased
- No stemming was used
- Terms were sorted alphabetically

Step 1 : Set term weights and construct the term-document matrix A and query matrix

Terms	d1	d2	d3	q
a	1	1	1	0
arrived	0	1	1	0
damaged	1	0	0	0
delivery	0	1	0	0
fire	1	0	0	0
gold	1	0	1	1
in	1	1	1	0
of	1	1	1	0
shipment	1	0	1	0
silver	0	2	0	1
truck	0	1	1	1

Step 2: Decompose matrix A matrix and find the U, S and V matrices, where A = U S (V)T

$$U = \begin{bmatrix} -0.4201 & 0.0748 & -0.0460 \\ -0.2995 & -0.2001 & 0.4078 \\ -0.1206 & 0.2749 & -0.4538 \\ -0.1576 & -0.3046 & -0.2006 \\ -0.1206 & 0.2749 & -0.4538 \\ -0.2626 & 0.3794 & 0.1547 \\ -0.4201 & 0.0748 & -0.0460 \\ -0.4201 & 0.0748 & -0.0460 \\ -0.2626 & 0.3794 & 0.1547 \\ -0.3151 & -0.6093 & -0.4013 \\ -0.2995 & -0.2001 & 0.4078 \end{bmatrix} \quad S = \begin{bmatrix} 4.0989 & 0.0000 & 0.0000 \\ 0.0000 & 2.3616 & 0.0000 \\ 0.0000 & 0.0000 & 1.2737 \end{bmatrix}$$

$$V = \begin{bmatrix} -0.4945 & 0.6492 & -0.5780 \\ -0.6458 & -0.7194 & -0.2556 \\ -0.5817 & 0.2469 & 0.7750 \end{bmatrix} \quad V^T = \begin{bmatrix} -0.4945 & -0.6458 & -0.5817 \\ 0.6492 & -0.7194 & 0.2469 \\ -0.5780 & -0.2556 & 0.7750 \end{bmatrix}$$

Step 3: Implement a Rank 2 Approximation by keeping the first two columns of U and V and the first two columns and rows of S.

$$\begin{bmatrix} -0.4201 & 0.0748 \\ -0.2995 & -0.2001 \end{bmatrix}$$

$k = 2$

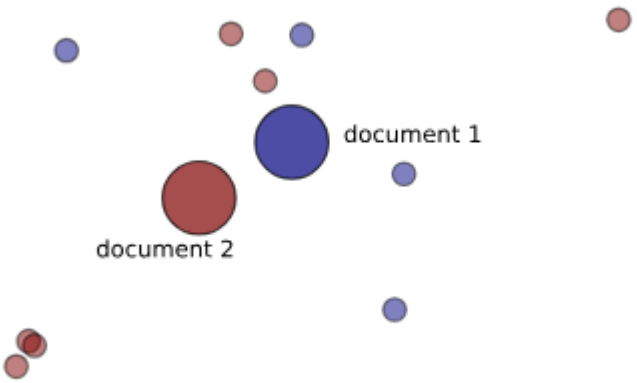
$$U \approx U_k = \begin{bmatrix} -0.1206 & 0.2749 \\ -0.1576 & -0.3046 \\ -0.1206 & 0.2749 \\ -0.2626 & 0.3794 \\ -0.4201 & 0.0748 \\ -0.4201 & 0.0748 \\ -0.2626 & 0.3794 \\ -0.3151 & -0.6093 \\ -0.2995 & -0.2001 \end{bmatrix} \quad S \approx S_k = \begin{bmatrix} 4.0989 & 0.0000 \\ 0.0000 & 2.3616 \end{bmatrix}$$

$$V \approx V_k = \begin{bmatrix} -0.4945 & 0.6492 \\ -0.6458 & -0.7194 \\ -0.5817 & 0.2469 \end{bmatrix} \quad V^T \approx V_k^T = \begin{bmatrix} -0.4945 & -0.6458 & -0.5817 \\ 0.6492 & -0.7194 & 0.2469 \end{bmatrix}$$

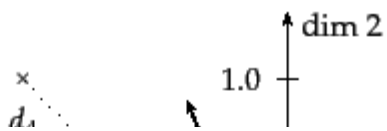
Step 4: Find the new document vector coordinates in this reduced 2-dimensional space.

Rows of V holds ***eigenvector values***. These are the coordinates of individual document vectors, hence

- ***d1*** = (-0.4945, 0.6492)
- ***d2*** = (-0.6458, -0.7194)
- ***d3*** = (-0.5817, 0.2469)



Step 5: Find the **new query vector coordinates** in the **reduced 2-dimensional space**.



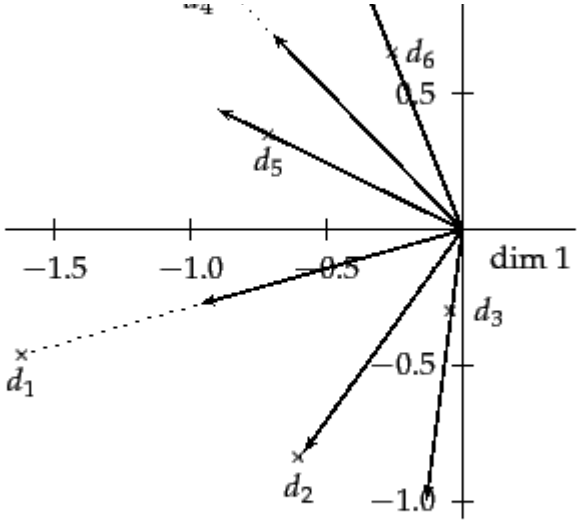


Figure 18.3: The documents of Example 18.4 reduced to two dimensions in $(V')^T$.

These are the **new coordinate of the query vector in two dimensions**. Note how this matrix is now different from the original query matrix q given in Step 1.

$$\mathbf{q} = \mathbf{q}^T \mathbf{U}_k \mathbf{S}_k^{-1} \qquad k = 2$$

$$\mathbf{q} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} -0.4201 & 0.0748 \\ -0.2995 & -0.2001 \\ -0.1206 & 0.2749 \\ -0.1576 & -0.3046 \\ -0.1206 & 0.2749 \\ -0.2626 & 0.3794 \\ -0.4201 & 0.0748 \\ -0.4201 & 0.0748 \\ -0.2626 & 0.3794 \\ -0.3151 & -0.6093 \\ -0.2995 & -0.2001 \end{bmatrix} \begin{bmatrix} \frac{1}{4.0989} & 0.0000 \\ 0.0000 & \frac{1}{2.3616} \end{bmatrix}$$

$$\mathbf{q} = \begin{bmatrix} -0.2140 & -0.1821 \end{bmatrix}$$

Step 6: Rank documents in decreasing order of query-document cosine similarities.

$$\mathbf{sim}(\mathbf{q}, \mathbf{d}) = \frac{\mathbf{q} \bullet \mathbf{d}}{|\mathbf{q}|\,|\mathbf{d}|}$$

$$\mathbf{sim}(\mathbf{q}, \mathbf{d}_1) = \frac{(-0.2140)(-0.4945) + (-0.1821)(0.6492)}{\sqrt{(-0.2140)^2 + (-0.1821)^2}\,\sqrt{(-0.4945)^2 + (0.6492)^2}} = -0.0541$$

$$\text{sim}(q, d_2) = \frac{(-0.2140)(-0.6458) + (-0.1821)(-0.7194)}{\sqrt{(-0.2140)^2 + (-0.1821)^2}\ \sqrt{(-0.6458)^2 + (-0.7194)^2}} = 0.9910$$

$$\text{sim}(q, d_3) = \frac{(-0.2140)(-0.5817) + (-0.1821)(0.2469)}{\sqrt{(-0.2140)^2 + (-0.1821)^2}\ \sqrt{(-0.5817)^2 + (0.2469)^2}} = 0.4478$$

Ranking documents in descending order

$$d_2 > d_3 > d_1$$

Conclusion: We can see that document ***d2*** scores higher than ***d3*** and ***d1***. Its vector is closer to the query vector than the other vectors.

- More information :

Interpreting LSI Document Similarity · Chris McCormick

First, a little refresher on how LSI works. The first step in comparing the two pieces of text is to produce tf-idf...

mccormickml.com

4. Word Mover's Distance

I have 2 sentences:

- Obama speaks to the media in Illinois
- The president greets the press in Chicago

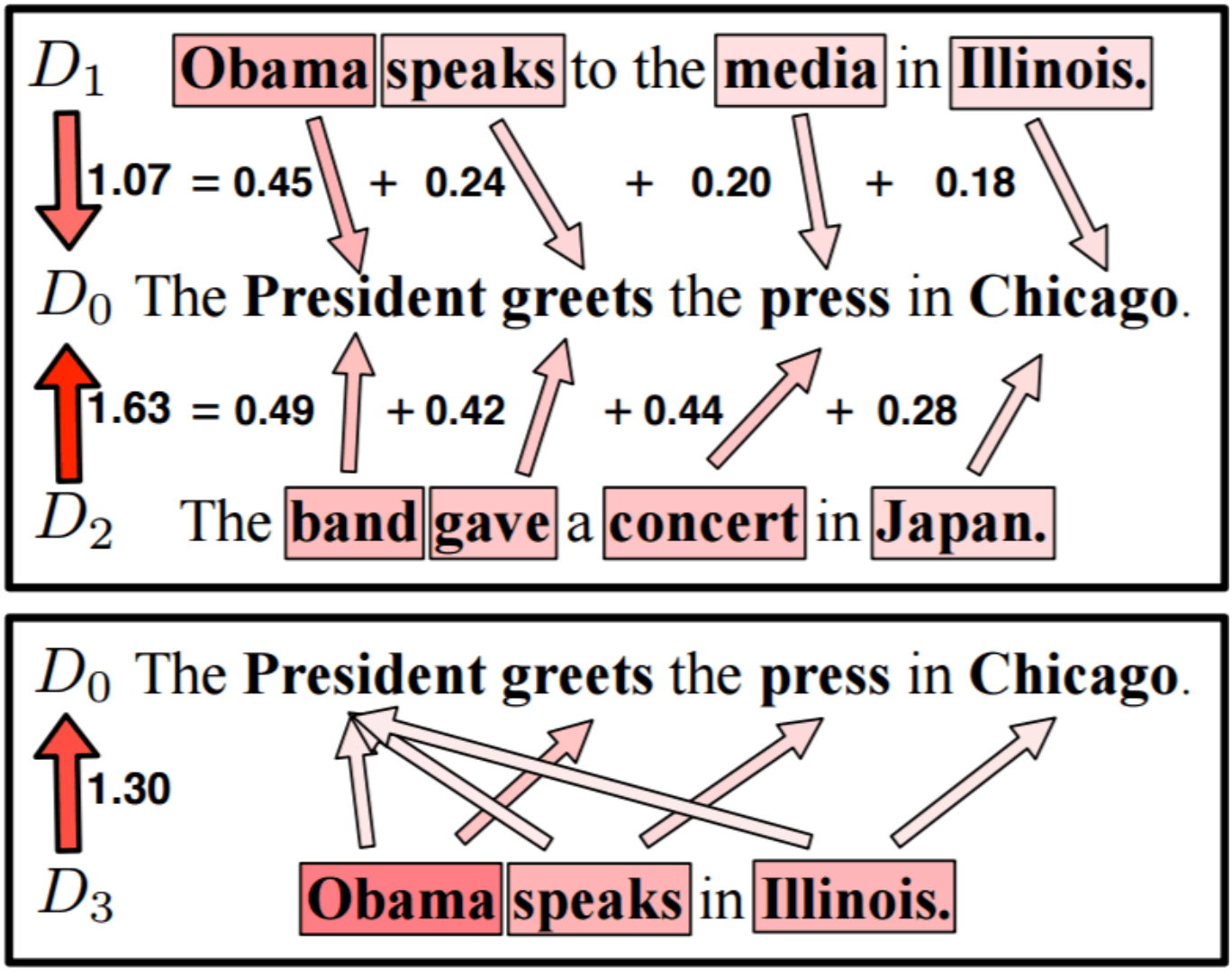
Removing stop words:

- Obama speaks media Illinois
- president greets press Chicago

My sparse vectors for the 2 sentences **have no common words** and will have **a cosine distance of 0**. This is a **terrible distance** score because **the 2 sentences have very similar meanings**.

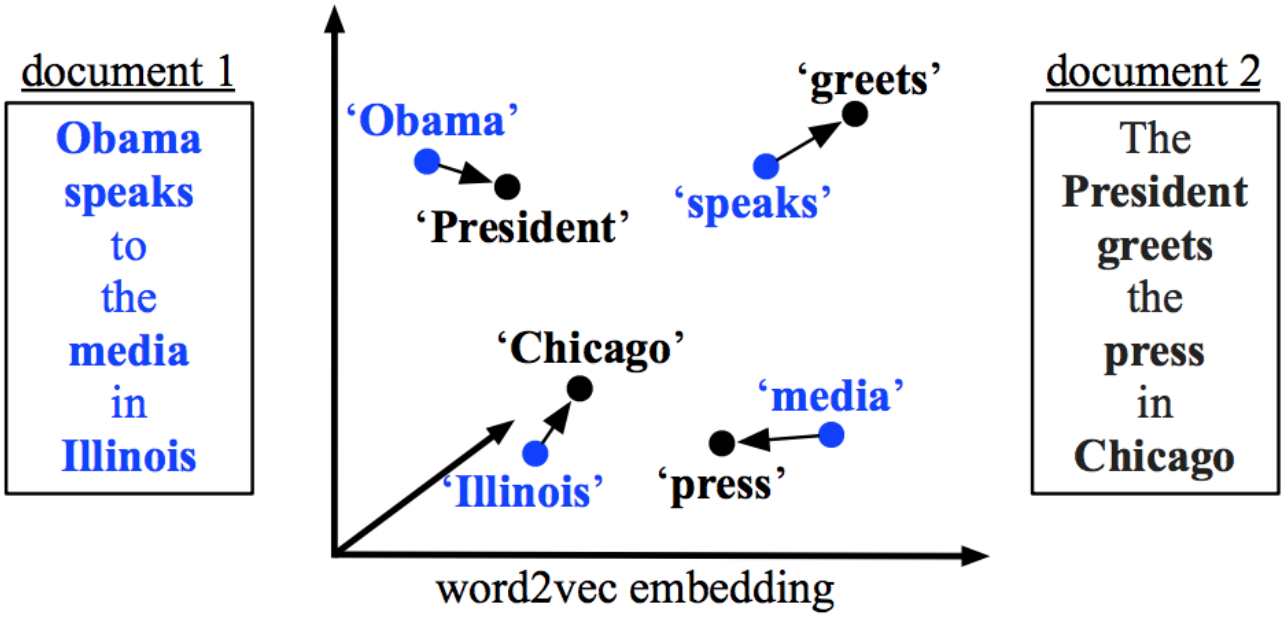
Cosine value of 0 means that the **two vectors are at 90 degrees to each other (orthogonal) and have no match**. The closer the cosine value to 1, the smaller the angle and the greater the match between vectors.

Word Mover's Distance solves this problem by taking account of the **words' similarities in word embedding space**.

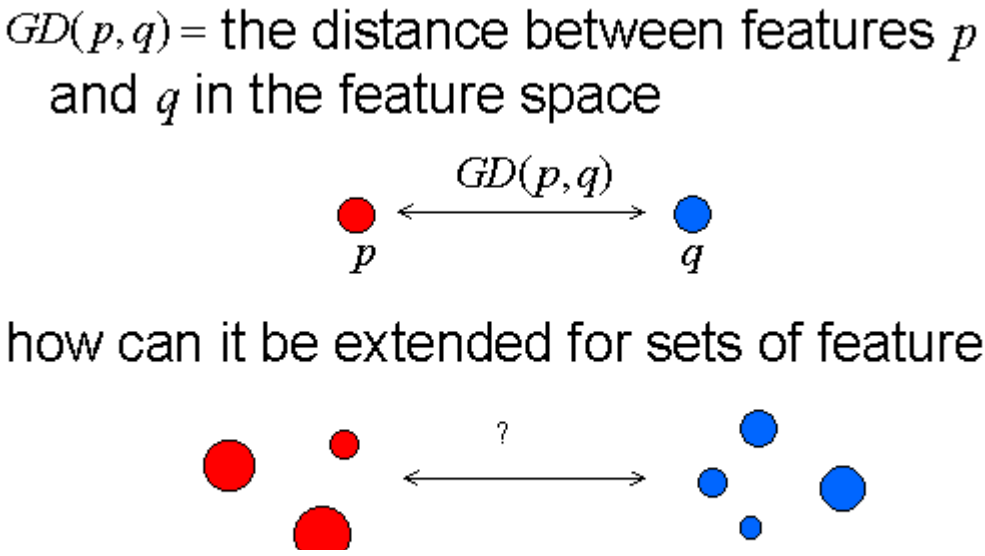


WMD uses the **word embeddings of the words in two texts to measure the minimum distance** that the words in one text need to "travel" in semantic space to reach the words in the other text.

WMD uses the word embeddings of the words in two texts to **measure the minimum distance** that the words in one text need to **"travel" in semantic space** to reach the words in the other text.

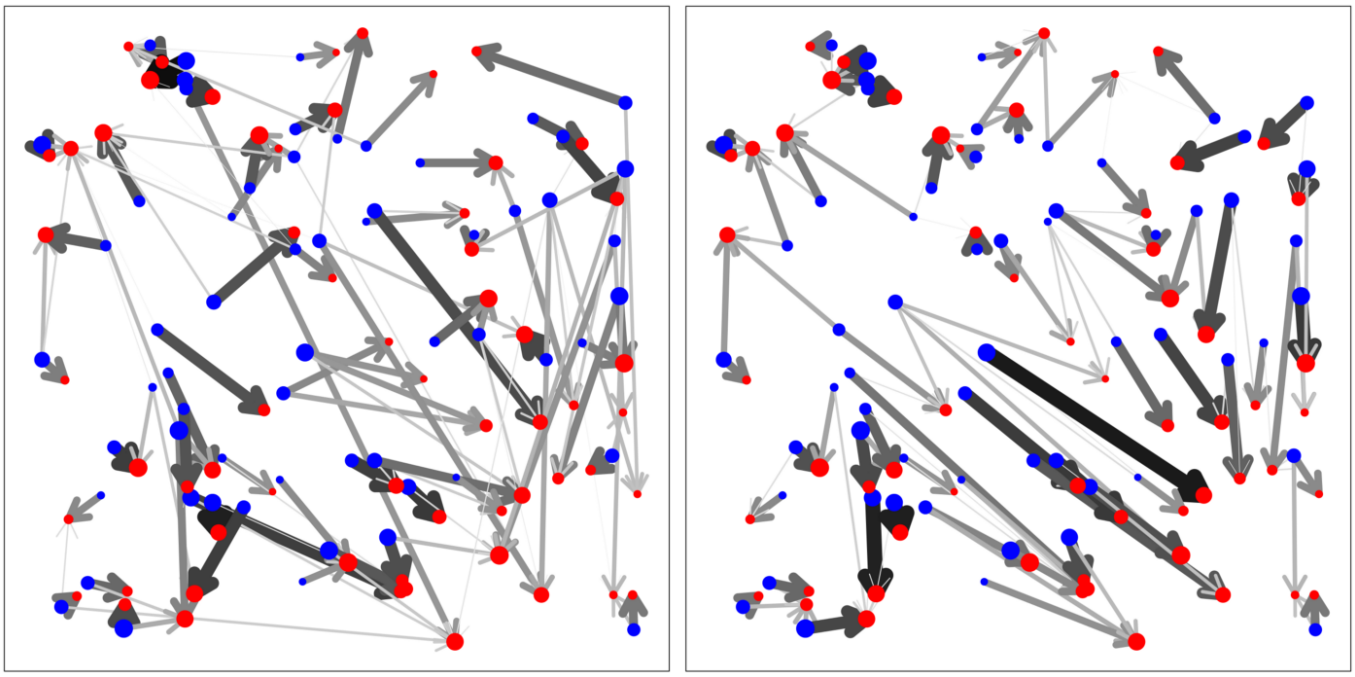


The **earth mover's distance** (**EMD**) is a measure of the distance between two probability distributions over a region *D* (known as the Wasserstein metric). Informally, if the distributions are interpreted as two different ways of piling up a certain amount of dirt over the region *D*, the **EMD is the minimum cost of turning one pile into the other**; where the cost is assumed to be amount of dirt moved times the distance by which it is moved.



$\{p_i\}$ $\{q_j\}$

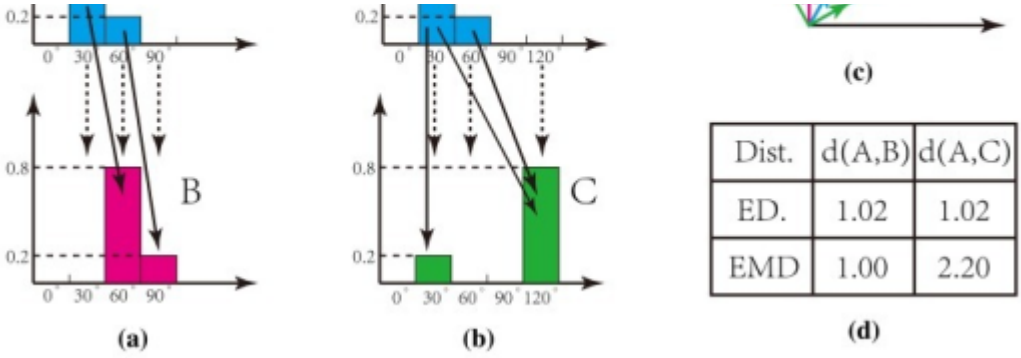
This is a Transportation problem — meaning we want to minimize the cost to transport a large volume to another volume of equal size.



To give some example, it looks like such as:

- Gini's measure of discrepancy
- Transportation problem [Hitchcock]
- Monge Kantorovitch problem
- Omstein Distance
- Lipschitz norm …

First hint : Measurement of the histogram distance by Euclidean and Earth mover distance (EMD)



Dist.	d(A,B)	d(A,C)
ED.	1.02	1.02
EMD	1.00	2.20

(d)

Euclidean distance fails to reflect the true distance. It is clear that histograms A and B are similar, while A and C are much more different. However, ***these two groups are evaluated with the same distance based on the Euclidean distance, which are indicated by the dashed lines***. To tackle this problem, we adopt the Earth mover distance (EMD), in which both ***the orientation*** and ***energy value*** can be taken into account, and an optimal solution will be found by ***minimizing the movement cost***, as indicated by the solid lines.

EMD: the minimum amount of work needed to transform signature **P** to signature **Q**

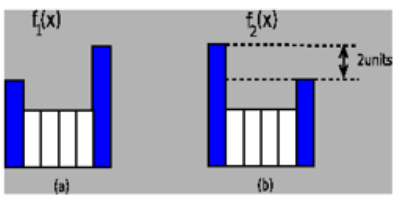
$$\mathrm{EMD}(\mathbf{P},\mathbf{Q}) = \min_{F} \frac{\sum_{i=1}^{m}\sum_{j=1}^{n} f_{ij} GD(p_i, q_j)}{\sum_{i=1}^{m}\sum_{j=1}^{n} f_{ij}}$$

where $F = \{f_{ij}\}$ is an admissible flow from $\{p_i\}$ to $\{q_j\}$

- **P** and **Q** can have different weights!

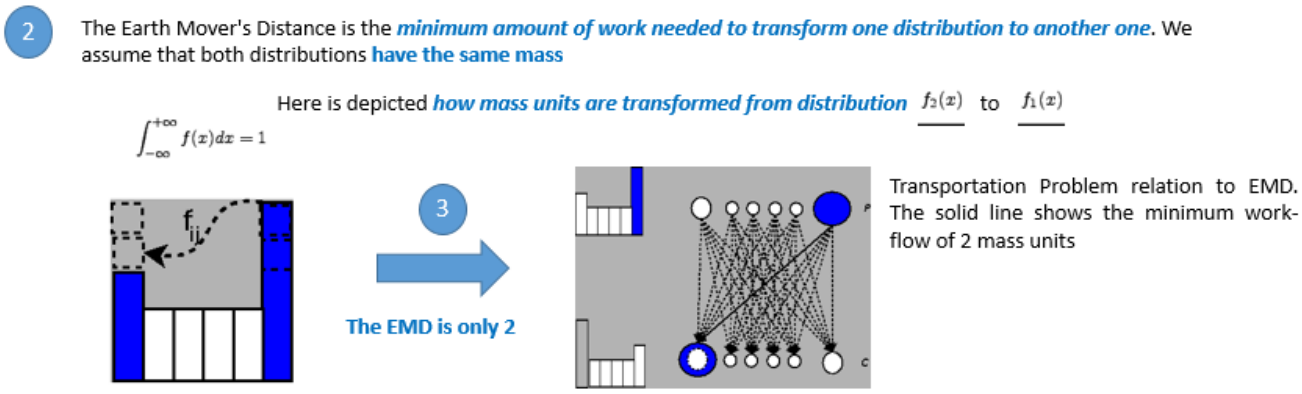
Thus, **the EMD distance can evaluate the true distance of our histograms.**

1 The two direction distributions are ***almost the same*** or at least ***very similar to each other***



These two distributions are ***very close to each other***, but their ***Euclidean distance is quite large***

The Euclidean distance would be $2^2+0+0+0+0+2^2=4$

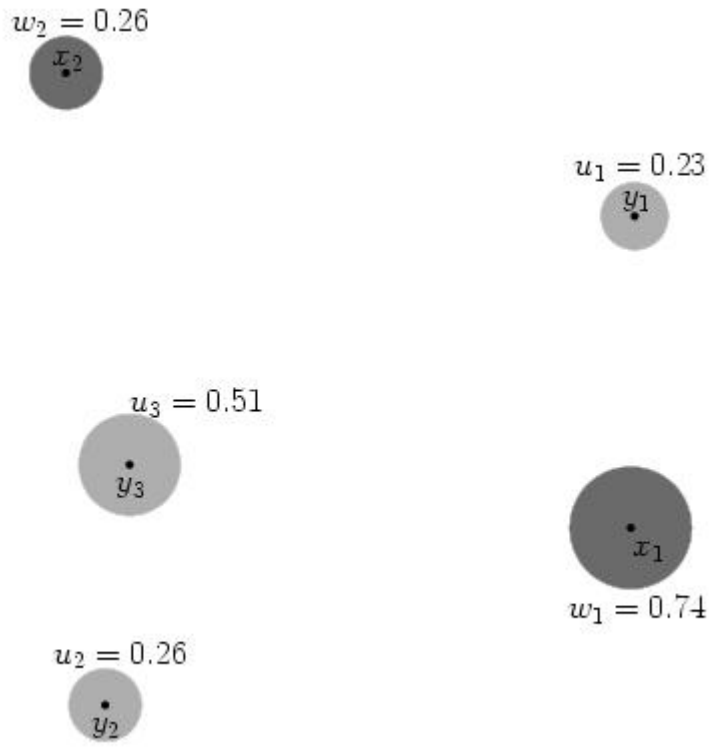


The Earth Mover's Distance (EMD) is a distance measure between discrete, finite distributions :

x = { *(x1,w1), (x2,w2), …, (xm,wm)* }

y = { *(y1,u1), (y2,u2), …, (yn,un)* }.

The ***x distribution*** has an amount of mass or weight **w_i** at position x_i in **R**^K with *i=1,…,m,* while the ***y distribution*** has weight **u_j** at position y_j with *j=1,…,n*. An example pair of distributions in **R**2 is shown below



- **x** has m=2 masses and
- **y** has n=3 masses.

The circle centers are the points (mass locations) of the distributions. The area of a circle is proportional to the weight at its center point.

The total weight of

- **x** is ***w_S*** = Sum of all w_i *[0.74 + 0.26]*
- **y** is ***u_S*** = Sum of all u_j *[0.51 + 0.23 + 0.26]*

In the example above, the distributions have equal total weight **w_S**=**u_S**=1.

Linear Optimization

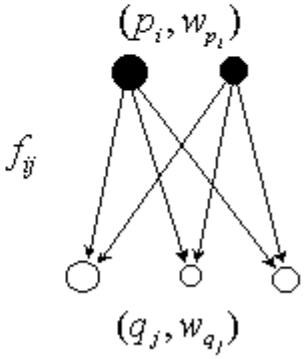
find a flow F that minimizes $\sum_{i=1}^{m}\sum_{j=1}^{n} f_{ij} GD(p_i, q_j)$

subject to: $f_{ij} \geq 0$

$$\sum_{j=1}^{n} f_{ij} \leq w_{p_i}$$

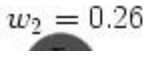
$$\sum_{i=1}^{m} f_{ij} \leq w_{q_j}$$

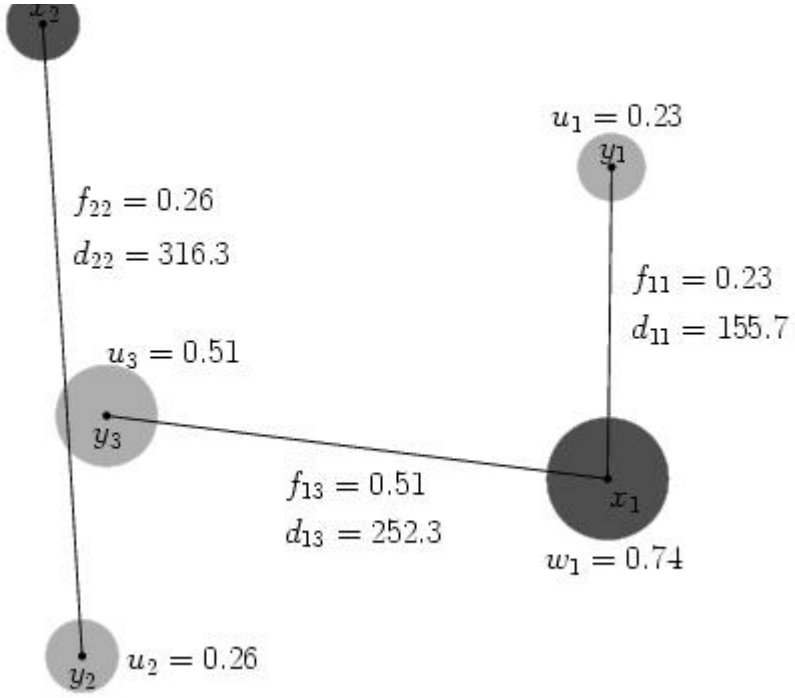
$$F = \sum_{i=1}^{m}\sum_{j=1}^{n} f_{ij} = \min\left(\sum_{i=1}^{m} w_{p_i}, \sum_{j=1}^{n} w_{q_j}\right)$$



Equal-Weight Distributions

The EMD between **two equal-weight distributions** is proportional to the ***amount of work needed to morph one distribution into the other***. We morph **x** into **y** by **transporting mass** from the **x mass locations** to the **y mass locations** until **x** has been r**earranged to look exactly like y**. An example morph is shown below.





The **amount of mass** transported from x_i to y_j is denoted **f_ij**, and is called ***a flow between xi and yj***.

The work done to transport an amount of mass f_ij from xi to yj is the product of the f_ij and the distance dij=d(xi,yj) between xi and yj.

The total amount of work to morph **x** into **y** by the flow F=(f_ij) is the sum of the individual works:

WORK(F,**x**,**y**) = [sum_i = (1..m) & j = (1..n)] f_ij d(x_i,y_j).

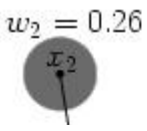
In the above flow example, the total amount of work done is

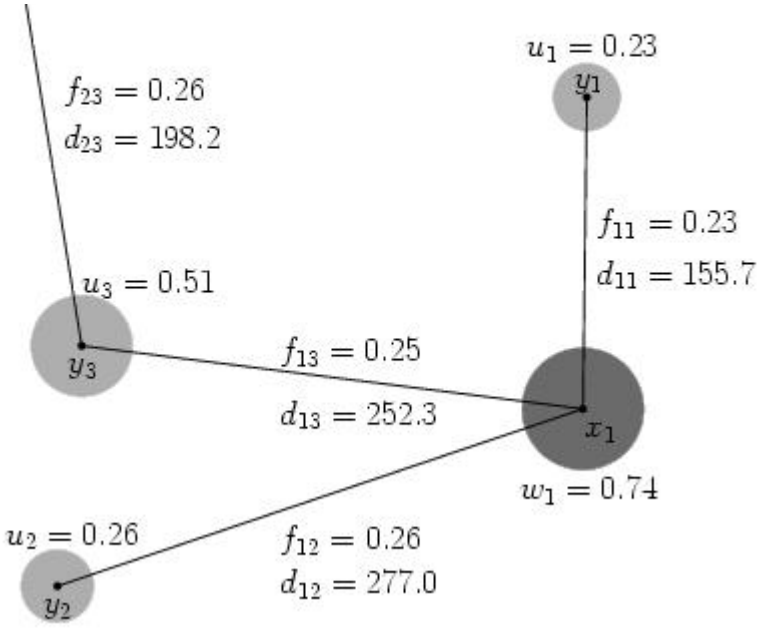
0.23*155.7 + 0.51*252.3 + 0.26*316.3 = **246.7 … not the best one.**

A minimum work flow

The total amount of work done by this flow is

0.23*155.7 + 0.26*277.0 + 0.25*252.3 + 0.26*198.2 = 222.4.





The EMD between equal-weight distributions is the minimum work to morph one into the other, divided by the total weight of the distributions.

The normalization by the total weight makes the EMD equal to the average distance travelled by mass during an optimal (i.e. work minimizing) flow.

The EMD does not change if all the weights in both distributions are scaled by the same factor.

In the previous example, the total weight is 1, so the EMD is equal to the minimum amount of work: EMD(**x**,**y**)=222.4.

Flow

It's very intuitive when all the words line up with each other, but what happens when the number of words are different?

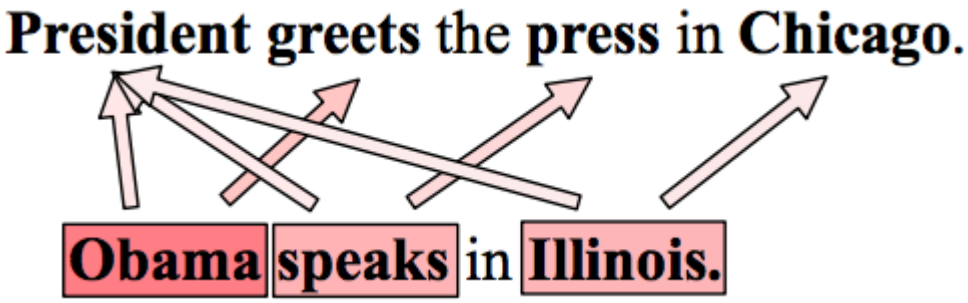
Example:

1. Obama speaks to the media in Illinois –> Obama speaks media Illinois –> 4 words
2. The president greets the press –> president greets press –> 3 words

WMD stems from an optimization problem called the Earth Mover's Distance, which has been applied to tasks like image search. EMD is an optimization problem that tries to

solve for **flow**.

Every unique word (out of N total) is given a flow of `1 / N`. Each word in sentence 1 has a flow of `0.25`, while each in sentence 2 has a flow of `0.33`. Like with liquid, what goes out must sum to what went in.



Since `1/3 > 1/4`, excess flow from words in the bottom also flows towards the other words. At the end of the day, this is an optimization problem to minimize the distance between the words.

$$\min_{\mathbf{T} \geq 0} \sum_{i,j=1}^{n} \mathbf{T}_{ij} c(i,j)$$

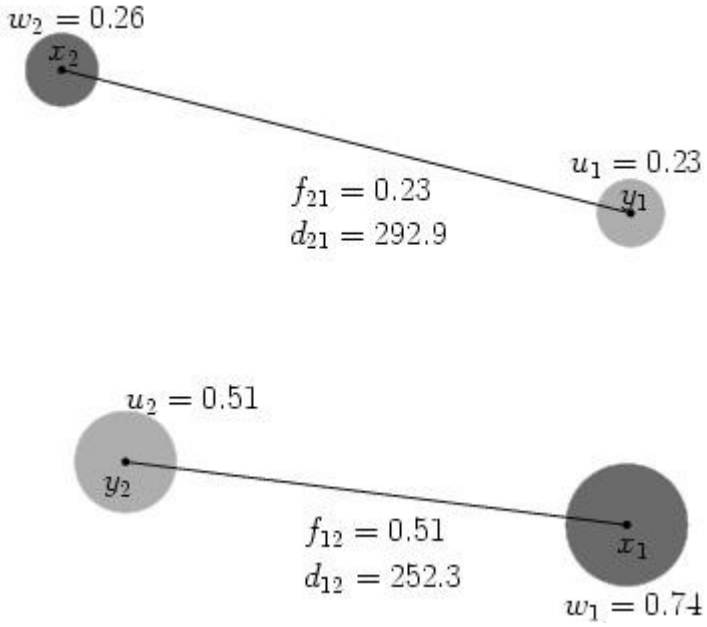
$$\text{subject to:} \sum_{j=1}^{n} \mathbf{T}_{ij} = d_i \quad \forall i \in \{1, \dots, n\}$$

$$\sum_{i=1}^{n} \mathbf{T}_{ij} = d'_j \quad \forall j \in \{1, \dots, n\}.$$

`T` is the flow and `c(i,j)` is the Euclidean distance between words i and j.

When the distributions **do not have equal total weights**, it is not possible to rearrange the mass in one so that it exactly matches the other. The EMD between unequal-weight

distributions is proportional to the minimum amount of work needed to cover the mass in the lighter distribution by mass from the heavier distribution. An example of a flow between unequal-weight distributions is given below.



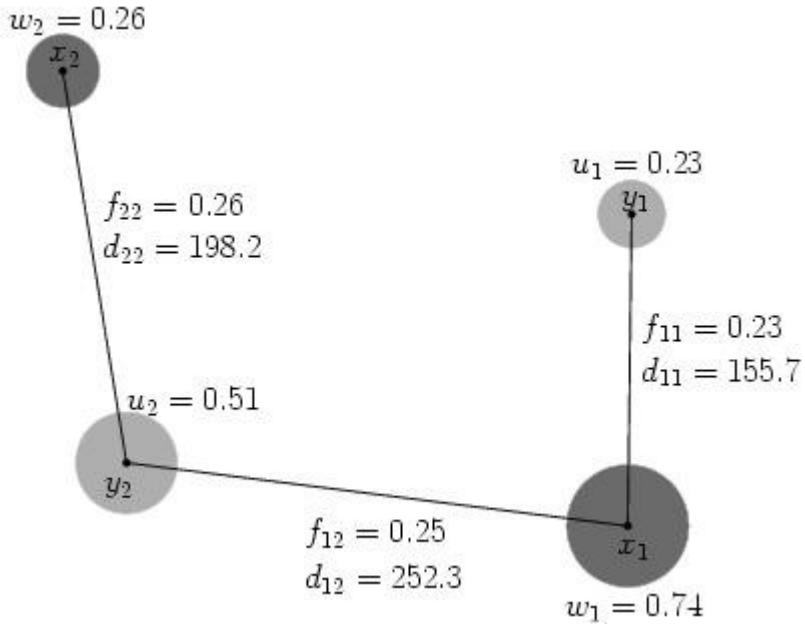
Here wS=1 and uS=0.74, so **x** is heavier than **y**. The flow covers or matches all the mass in **y** with mass from **x**. The total amount of work to cover **y** by this flow is

0.51*252.3 + .23*292.9 = 196.0.

In this example, 0.23 of the mass at x1 and 0.03 of the mass at x2 is not used in matching all the **y** mass.

In general, some of the mass (wS-uS if **x** is heavier than **y**) in the heavier distribution is not needed to match all the mass in the lighter distribution. For this reason, the case of matching unequal-weight distributions is called the *partial matching* case. The case when the distributions are equal-weight is called the *complete matching* case because all the mass in one distribution is matched to all the mass in the other distribution. Another way to visualize matching distributions is to imagine piles of dirt and holes in the ground. The dirt piles are located at the points in the heavier distribution, and the the holes are located at the points of the lighter distribution. The volume of a dirt pile or the volume of dirt missing from a hole is equal to the weight of its point. Matching

distributions means filling all the holes with dirt. In the partial matching case, there will be some dirt leftover after all the holes are filled.



The total amount of work for this flow to cover **y** is

0.23*155.7 + 0.25*252.3 + 0.26*198.2 = 150.4.

The EMD is the minimum amount of work to cover the mass in the lighter distribution by mass from the heavier distribution, divided by the weight of the lighter distribution (which is the total amount of mass moved). As in the complete matching case, the normalization of the minimum work makes the EMD equal to the average distance mass travels during an optimal flow. Again, scaling the weights in both distributions by a constant factor does not change the EMD. In the above example, the weight of the lighter distribution is uS=0.74, so EMD(**x**,**y**)= 150.4/0.74 = 203.3.

5. LDA with Jensen-Shannon distance

LDA has many uses:

- Understanding the different varieties topics in a corpus (obviously),
- Getting a better insight into the type of documents in a corpus (whether they are about news, wikipedia articles, business documents)

- Quantifying the most used / most important words in a corpus
- … and even **document similarity and recommendation** (here is we focus all our attention)

Let's kick off by reading this amazing article from Kaggle called ***LDA and Document Similarity***

Latent Dirichlet Allocation (LDA), is an unsupervised **generative model** that assigns **topic distributions** to **documents**.

At a high level, the model assumes that **each document will contain several topics**, so that there is **topic overlap within a document**. The words in each document contribute to these topics. The topics may not be known a priori, and needn't even be specified, but the **number** of topics must be specified a priori. Finally, there can be words overlap between topics, so several topics may share the same words.

The model generates to **latent** (hidden) variables :

- A distribution over topics for each document **(1)**
- A distribution over words for each topics **(2)**

After training, **each document will have a discrete distribution over all topics**, and **each topic will have a discrete distribution over all words**.

The mathematics of collapsed gibbs sampling (cut back version)

Recall that when we iterate through each word in each document, we unassign its current topic assignment and reassign the word to a new topic. The topic we reassign the word to is based on the probabilities below.

$$P\left(\text{document "likes" the topic}\right) \times P\left(\text{topic "likes" the word } w'\right)$$

$$\Rightarrow \frac{n_{i,k} + \alpha}{N_i - 1 + K\alpha} \times \frac{m_{w',k} + \gamma}{\sum_{w \in V} m_{w,k} + V\gamma}$$

where

$n_{i,k}$ - number of word assignments to topic k in document i

$n_{i,k}$ - number of assignments to topic k in document i

α - smoothing parameter (hyper parameter - make sure probability is never 0)

N_i - number of words in document i

-1 - don't count the current word you're on

K - total number of topics

$m_{w',k}$ - number of assignments, corpus wide, of word w' to topic k

$m_{w',k}$ - number of assignments, corpus wide, of word w' to topic k

γ - smoothing parameter (hyper parameter - make sure probability is never 0)

$\sum_{w \in V} m_{w,k}$ - sum over all words in vocabulary currently assigned to topic k

V size of vocabulary i.e. number of distinct words corpus wide

Amazing ❤ https://www.kaggle.com/ktattan/lda-and-document-similarity

Now we have **a topic distribution for a new unseen document.** The goal is to **find the most similar documents in the corpus**.

> *To do that we compare the topic distribution of the new document to all the topic distributions of the documents in the corpus. We use the Jensen-Shannon distance metric to find the most similar documents.*

What the Jensen-Shannon distance tells us, is **which documents are statisically "closer" (and therefore more similar), by comparing the divergence of their distributions**.

Jensen-Shannon is a method of **measuring the similarity between two probability distributions**. It is also known as **information radius** (**IRad**) or **total divergence to the average**

Jensen-Shannon is symmetric, unlike Kullback-Leibler on which the formula is based. This is good, because we want the **similarity between documents A and B to be the same as the similarity between B and A**.

Similarity query

Ok, now that we have a topic distribution for a new unseen document, let's say we wanted to find the most similar documents in the corpus. We can do this by comparing the topic distribution of the new document to all the topic distributions of the documents in the corpus. We use the Jensen-Shannon distance metric to find the most similar documents.

What the Jensen-Shannon distance tells us, is which documents are statisically "closer" (and therefore more similar), by comparing the divergence of their distributions. Jensen-Shannon is symmetric, unlike Kullback-Leibler on which the formula is based. This is good, because we want the similarity between documents A and B to be the same as the similarity between B and A.

The formula is described below.

For discrete distirbutions P and Q, the Jensen-Shannon divergence, JSD is defined as

$$JSD\left(P||Q\right) = \frac{1}{2}D\left(P||M\right) + \frac{1}{2}D\left(Q||M\right)$$

where $M = \frac{1}{2}(P+Q)$

and D is the Kullback-Leibler divergence

$$D\left(P||Q\right) = \sum_i P(i)\log\left(\frac{P(i)}{Q(i)}\right)$$

$$\Rightarrow JSD\left(P||Q\right) = \frac{1}{2}\sum_i\left[P(i)\log\left(\frac{P(i)}{\frac{1}{2}(P(i)+Q(i))}\right) + Q(i)\log\left(\frac{Q(i)}{\frac{1}{2}(P(i)+Q(i))}\right)\right]$$

The square root of the Jensen-Shannon divergence is the Jensen-Shannon Distance: $\sqrt{JSD\left(P||Q\right)}$

The smaller the Jensen-Shannon Distance, the more similar two distributions are (and in our case, the more similar any 2 documents are)

```python
def jensen_shannon(query, matrix):
    """
    This function implements a Jensen-Shannon similarity
    between the input query (an LDA topic distribution for a document)
    and the entire corpus of topic distributions.
    It returns an array of length M where M is the number of documents in the corpus
    """
    # lets keep with the p,q notation above
    p = query[None,:].T # take transpose
    q = matrix.T # transpose matrix
    m = 0.5*(p + q)
    return np.sqrt(0.5*(entropy(p,m) + entropy(q,m)))

def get_most_similar_documents(query,matrix,k=10):
    """
    This function implements the Jensen-Shannon distance above
    and retruns the top k indices of the smallest jensen shannon distances
    """
    sims = jensen_shannon(query,matrix) # list of jensen shannon distances
    return sims.argsort()[:k] # the top k positional index of the smallest Jensen Shannon distan
```

Jensen_Shannon_distance.py hosted with ♡ by GitHub view raw

https://www.kaggle.com/ktattan/lda-and-document-similarity

Create the **average document M between the two documents 1 and 2** by **averaging their probability distributions** or **merging the contents of both documents**. We measure **how much each of the documents 1 and 2 is different from the average document M through KL(P||M) and KL(Q||M)** Finally we average the differences from point 2. *Potential issue*: we might want to use weighted average to account for the documents lengths of both documents.

Beyond Cosine: A Statistical Test.

Instead of talking about whether two documents are similar, it is better to check whether two documents come from the same distribution.

This is a much more precise statement since it requires us to **define the distribution which could give origin to those two documents** by implementing **a test of homogeneity**.

Often similarity is used where more precise and powerful methods will do a better job.

- Using a symmetric formula, when the problem does not require symmetry. If we denote *sim(A,B)* as the similarity formula used, the formula is symmetric when $sim(A,B) = sim(B,A)$. For example, ranking documents based on a query, does not work well with a symmetric formula. The best working methods like BM25 and DFR are not similarities in spite of the term used in the Lucene documentation since they are not symmetric with respect to the document and query. As a consequence of the preceding point similarity is not optimal for ranking documents based on queries
- Using a similarity formula without understanding its origin and statistical properties. For example, the cosine similarity is closely related to the normal distribution, but the data on which it is applied is not from a normal distribution. In particular, the squared length normalization is suspicious.

6. Variational Auto Encoder

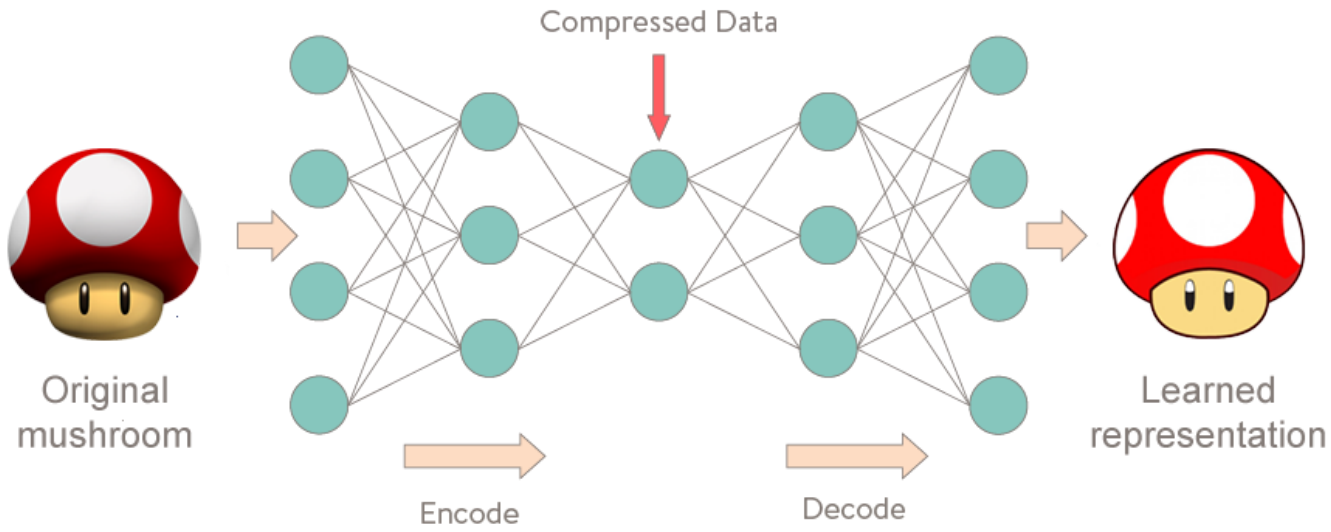
The job of those models is to **predict the input, given that same input**.

More specifically, let's take a look at Autoencoder Neural Networks. This **Autoencoder tries to learn to approximate the following identity function**:

$$f_{W,b}(x) \approx x$$

While trying to do just that might sound trivial at first, it is important to note that we want to learn a compressed representation of the data, thus **find structure**. This can be done by limiting the number of hidden units in the model. Those kind of autoencoders are called *undercomplete*.

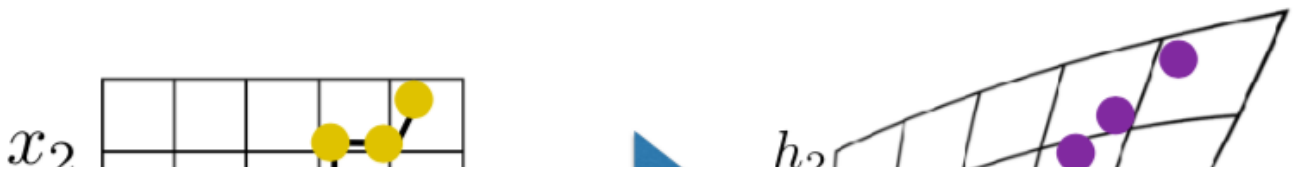
Here's a visual representation of what an Autoencoder might learn:

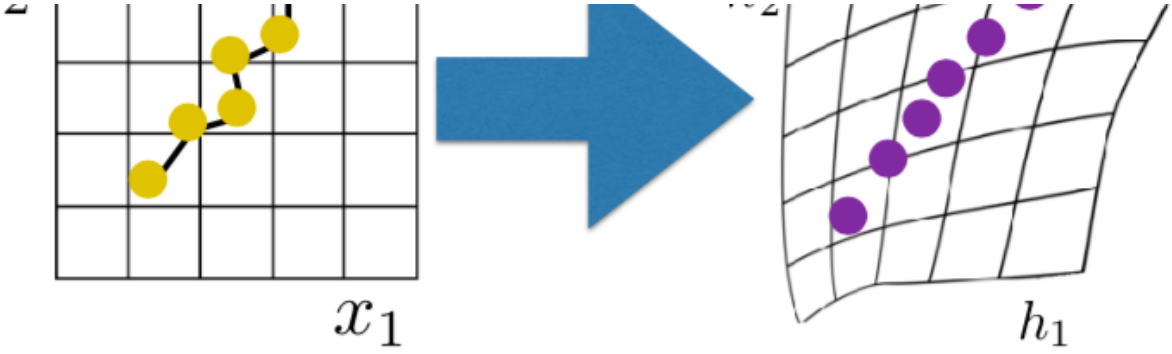


The key problem will be to **obtain the projection of data in single dimention without loosing information**. When this type of data is projected in latent space, **a lot of information is lost** and it is **almost impossible to deform and project it to the original shape**. No matter how much shifts and rotation are applied, original data cannot be recovered.

So how does neural networks solves this problem ? The intution is, **in the manifold space, deep neural networks has the property to bend the space in order to obtain a linear data fold view**. Autoencoder architectures applies this property in their hidden layers which allows them **to learn low level representations in the latent view space**.

The following image describes this property:





https://www.kaggle.com/shivamb/how-autoencoders-work-intro-and-usecases

Autoencoders are trained in an unsupervised manner in order to learn the exteremely low level repersentations of the input data.

A typical autoencoder architecture comprises of three main components:

- **Encoding Architecture :** The encoder architecture comprises of series of layers with decreasing number of nodes and ultimately reduces to a latent view repersentation.
- **Latent View Repersentation :** Latent view repersents the lowest level space in which the inputs are reduced and information is preserved.
- **Decoding Architecture :** The decoding architecture is the mirro image of the encoding architecture but in which number of nodes in every layer increases and ultimately outputs the similar (almost) input.

It **encodes data to latent (random) variables**, and then **decodes the latent variables to reconstruct the data**.

We may be interested in inferring the hidden states h from the input X (that is, we want to know what hidden states contributed to us seeing X). We can represent this idea using the `posterior distribution` $P(h|X)$

By conditional probability,

$$P(h|X) = P(h, X)/P(X)$$

But it is also true that,

$$P(X|h) = P(h, X)/P(h)$$

(because $P(Z, X) = P(X, Z)$, the joint distribution of X and h)

So we can rearrange a bit:

$$P(h|X) = P(X|h)P(h)/P(X)$$

In our model we have a graphical relationship between X and h. That is, we can infer which states caused X, $P(h|X)$ and we can generate more data from these hidden variables $P(X|h)$. We also have a prior distribution over our h's, $P(Z)$. This is often a Normal distribution. The problem is that $P(X)$ term.

To get that `marginal likelihood` $P(X)$, we need to marginalize out the h's. That is:

$$P(X) = \int_h P(X, h)dh$$

and the real issue is that the integral over h could be computationally `intractable`. That is, the space of h is so large that we cannot integrate over it in a reasonable amount of time. That means we cannot calculate our posterior distribution $P(h|X)$, since we can't calculate that denominator term, $P(X)$.

> *Note: The word `intractable` has a lot of meaning from the computational point of view and theoritical standpoint. A good discussion can be found over here.*

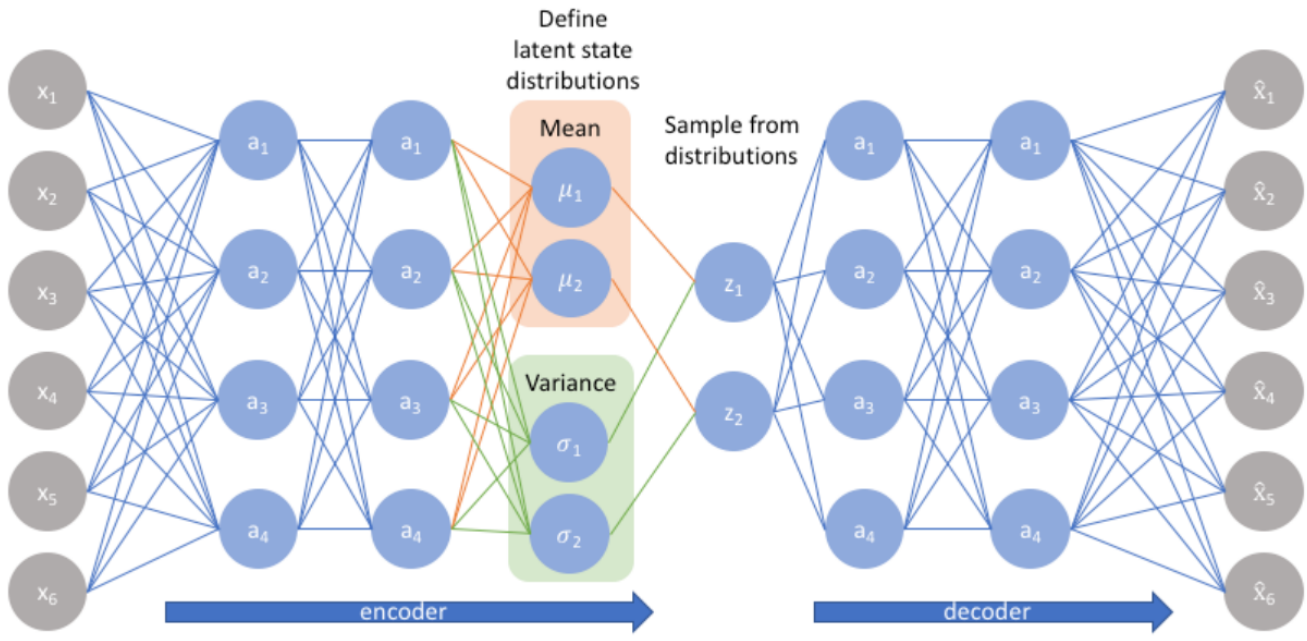
One trick that's used to combat this problem is `Markov chain Monte Carlo`, where the posterior distribution is set up as the equilibrium distribution of a Markov chain. However, this type of sampling method takes an extremely long time for a high-dimensional integral. The more popular method right now is variationalinference.

In variational inference, we introduce an approximate posterior distribution to stand in for our true, intractable posterior. The notation for the approximate posterior is $Q(Z|X)$. Then, to make sure that our approximate posterior is actually a good stand-in for our true posterior, we optimize an objective to ensure that they're close to one another in terms of a metric called ```Kullback-Leibler Divergence (KL Divergence)'''. Think of this as a distance function for probability distributions: it measures how close two probability distributions, in this case $Q(Z|X)$ and $P(Z|X)$, are.

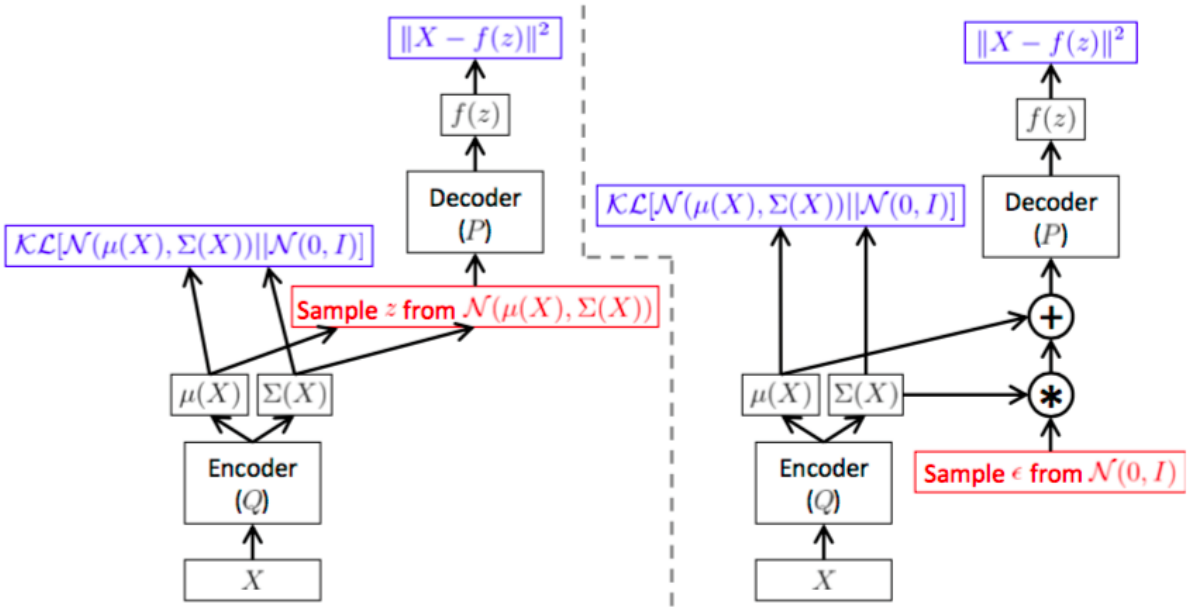
> *Rather than directly outputting values for the latent state as we would in a standard autoencoder, the encoder model of a VAE will* ***output parameters describing a distribution for each dimension in the latent space****. Since we're assuming that* ***our prior follows a normal distribution****, we'll output* two *vectors describing the* ***mean*** *and* ***variance of the latent state distributions****. If we were to build a true multivariate Gaussian model, we'd need to define a* ***covariance matrix describing how each of the dimensions are correlated****. However, we'll make a simplifying assumption that our covariance matrix only has nonzero values on the diagonal, allowing us to describe this information in a simple vector.*
>
> *Our* ***decoder model will then generate a latent vector by sampling from these defined distributions and proceed to develop a reconstruction of the original input****.*

In normal deterministic autoencoders the latent code does not learn the probability distribution of the data and therefore, it's not suitable to generate new data.

The VAE solves this problem since it explicitly defines a probability distribution on the latent code. In fact, **it learns the latent representations of the inputs not as single points but as soft ellipsoidal regions in the latent space, forcing the latent representations to fill the latent space rather than memorizing the inputs as punctual, isolated latent representations.**



A nice explanation of how low level features are deformed back to project the actual datahttps://www.kaggle.com/shivamb/how-autoencoders-work-intro-and-usecases



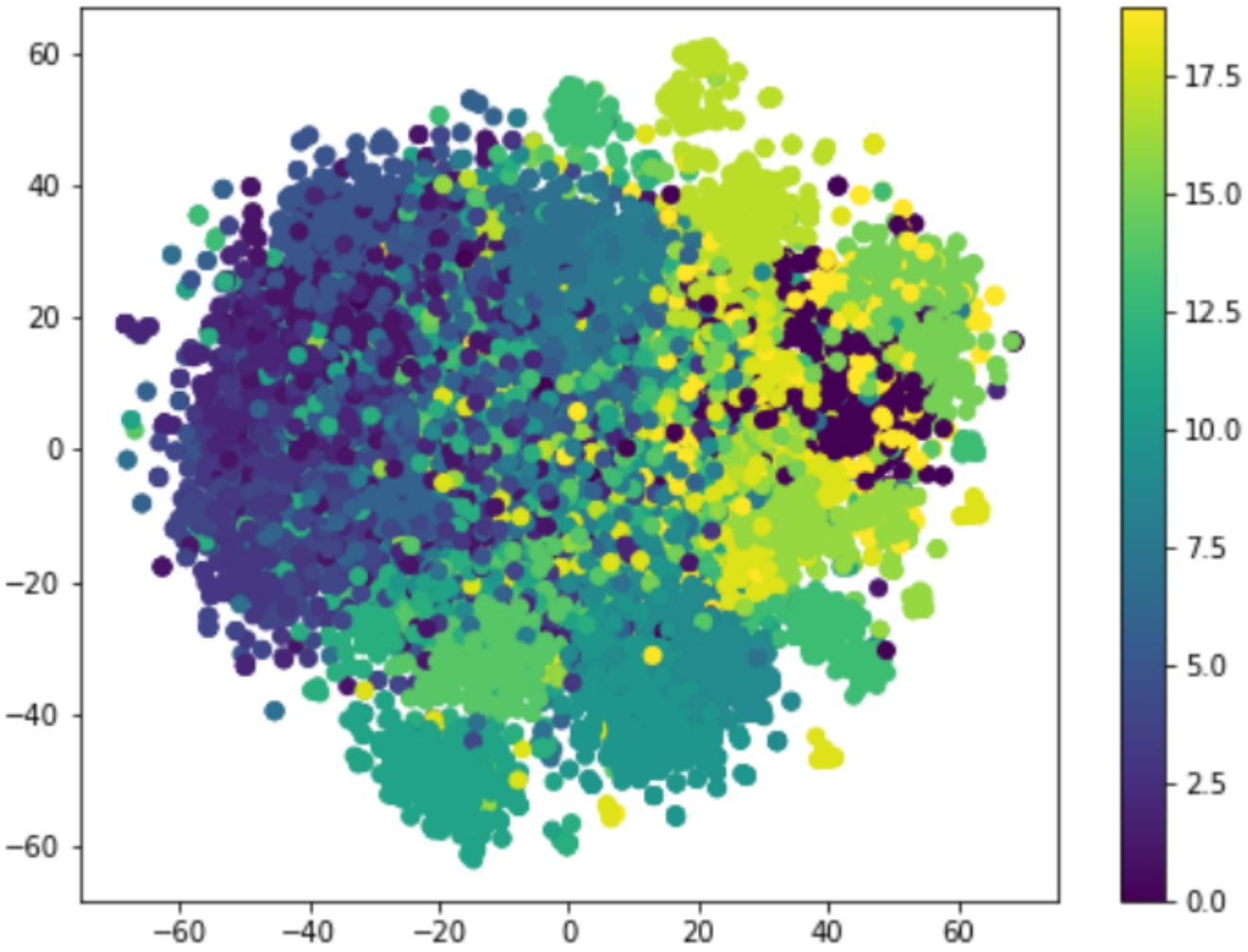
http://blog.qure.ai/notes/using-variational-autoencoders

The image illustrated above shows the architecture of a VAE. We see that the encoder part of the model i.e Q models the Q(z|X) (z is the latent representation and X the data). Q(z|X) is the part of the network that maps the data to the latent variables. The decoder part of the network is P which learns to regenerate the data using the latent variables as P(X|z). So far there is no difference between an autoencoder and a VAE. The difference is the constraint applied on z i.e the distribution of z is forced to be as close to Normal distribution as possible (KL divergence term).

Using a VAE we are able to fit a parametric distribution (in this case gaussian). This is what differentiates a VAE from a conventional autoencoder which relies only on the reconstruction cost. This means that during run time, when we want to draw samples from the network all we have to do is generate random samples from the Normal Distribution and feed it to the encoder P(X|z) which will generate the samples.

Our goal here is to use the **VAE to learn the hidden or latent representations of our textual data** — which is a matrix of Word embeddings. We will be using the VAE to **map the data to the hidden or latent variables.** We will then visualize these features to see

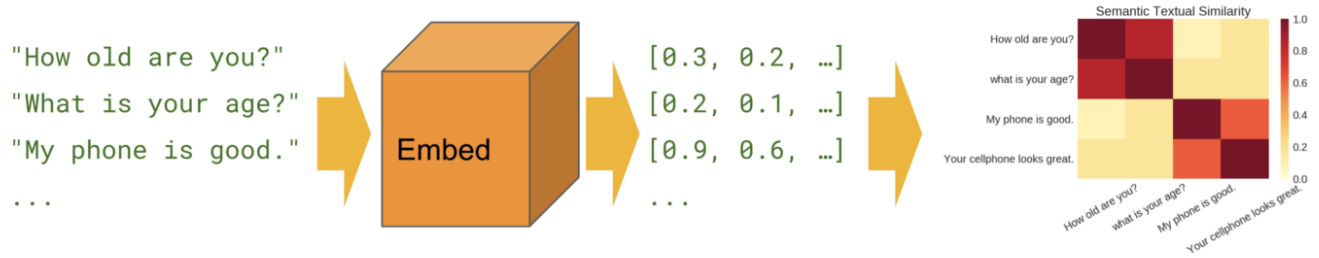
if the model has learnt to **differentiate between documents from different topics**. We hope that similar documents are closer in the Euclidean space in keeping with their topics. **Similar documents are next to each other.**



7. Pre-trained sentence encoders

The Universal Sentence Encoder encodes text into high dimensional vectors that can be used for text classification, semantic similarity, clustering and other natural language tasks.

Pre-trained sentence encoders aim to play the **same role as word2vec and GloVe, but for sentence embeddings**: the embeddings they produce can be used in a variety of applications, such as text classification, paraphrase detection, etc. Typically they have been trained on a range of supervised and unsupervised tasks, in order to capture as much universal semantic information as possible.



> … ***encoding sentences*** *into* ***embedding vectors*** *that specifically target* ***transfer learning*** *to other NLP tasks. (…)* ***transfer learning using sentence embeddings tends to outperform word level transfer****. With transfer learning via sentence embeddings, we observe surprisingly good performance with minimal amounts of supervised training data for a transfer task. We obtain encouraging results on Word Embedding Association Tests (WEAT) targeted at detecting model bias.*

The model is trained and optimized for greater-than-word length text, such as sentences, phrases or short paragraphs. It is trained on a variety of data sources and a variety of tasks with the aim of dynamically accommodating a wide variety of natural language understanding tasks. The input is variable length English text and the output is a 512 dimensional vector. We apply this model to the STS benchmark for semantic similarity, and the results can be seen in the example notebook made available. The universal-sentence-encoder model is trained with a deep averaging network (DAN) encoder.

Several such encoders are available :

- **InferSent (Facebook Research)** : BiLSTM with max pooling, trained on the SNLI dataset, 570k English sentence pairs labelled with one of three categories: entailment, contradiction or neutral.
- **Google Sentence Encoder** : a simpler Deep Averaging Network (DAN) where input embeddings for words and bigrams are averaged together and passed through a feed-forward deep neural network.

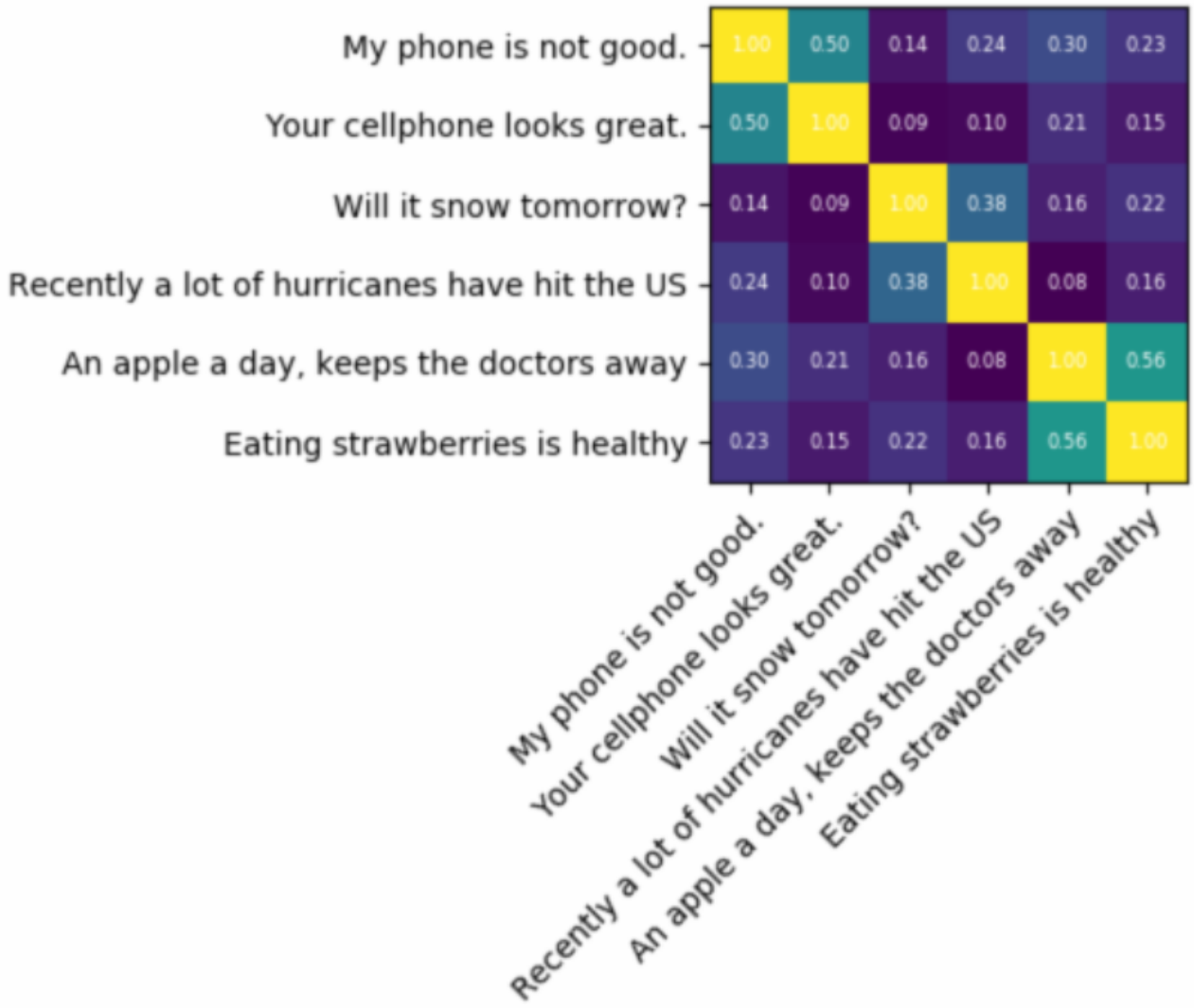
```
module_url = "https://tfhub.dev/google/universal-sentence-encoder/1?tf-hub-format=compressed"

# Import the Universal Sentence Encoder's TF Hub module
embed = hub.Module(module_url)

# sample text
```

```
messages = [
# Smartphones
"My phone is not good.",
"Your cellphone looks great.",

# Weather
"Will it snow tomorrow?",
"Recently a lot of hurricanes have hit the US",

# Food and health
"An apple a day, keeps the doctors away",
"Eating strawberries is healthy",
]

similarity_input_placeholder = tf.placeholder(tf.string, shape=(None))
similarity_message_encodings = embed(similarity_input_placeholder)
with tf.Session() as session:
    session.run(tf.global_variables_initializer())
    session.run(tf.tables_initializer())
    message_embeddings_ = session.run(similarity_message_encodings, feed_dict={similarity_input_

    corr = np.inner(message_embeddings_, message_embeddings_)
    print(corr)
    heatmap(messages, messages, corr)

def heatmap(x_labels, y_labels, values):
    fig, ax = plt.subplots()
    im = ax.imshow(values)

    # We want to show all ticks...
    ax.set_xticks(np.arange(len(x_labels)))
    ax.set_yticks(np.arange(len(y_labels)))
    # ... and label them with the respective list entries
    ax.set_xticklabels(x_labels)
    ax.set_yticklabels(y_labels)

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right", fontsize=10,
         rotation_mode="anchor")

    # Loop over data dimensions and create text annotations.
    for i in range(len(y_labels)):
        for j in range(len(x_labels)):
            text = ax.text(j, i, "%.2f"%values[i, j],
                           ha="center", va="center", color="w",
```

```
52  fontsize=6)
53
54      fig.tight_layout()
55      plt.show()
```

Universal_sentence_encoder.py hosted with ♡ by GitHub view raw

Finally what we got:



8. Siamese Manhattan LSTM (MaLSTM)

Siamese networks are networks that have **two or more identical sub-networks in them**.

Siamese networks seem to **perform well on similarity tasks** and have been used for

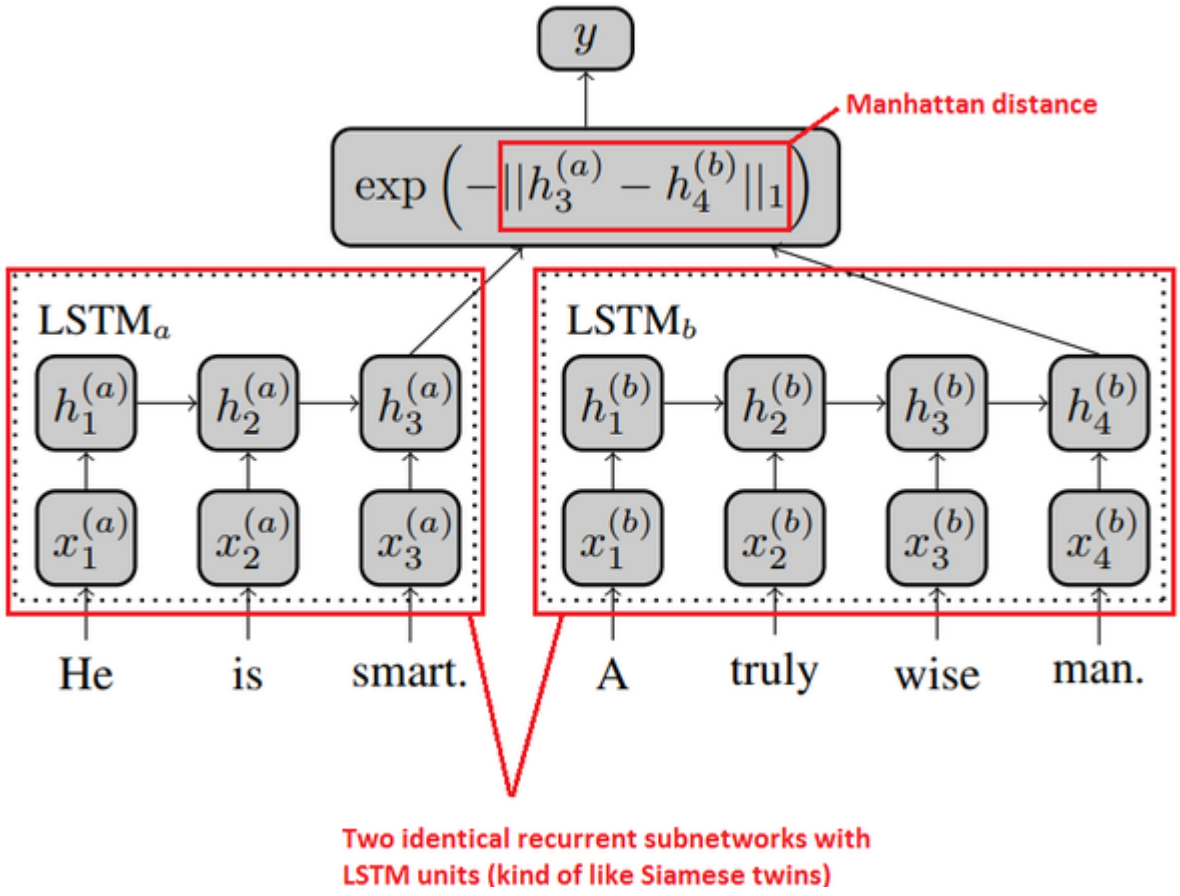
tasks like **sentence semantic similarity**, **recognizing forged signatures** and many more.

The names MaLSTM and SiameseLSTM might leave an impression that there are some kind of new LSTM units proposed, **but that is not the case**.

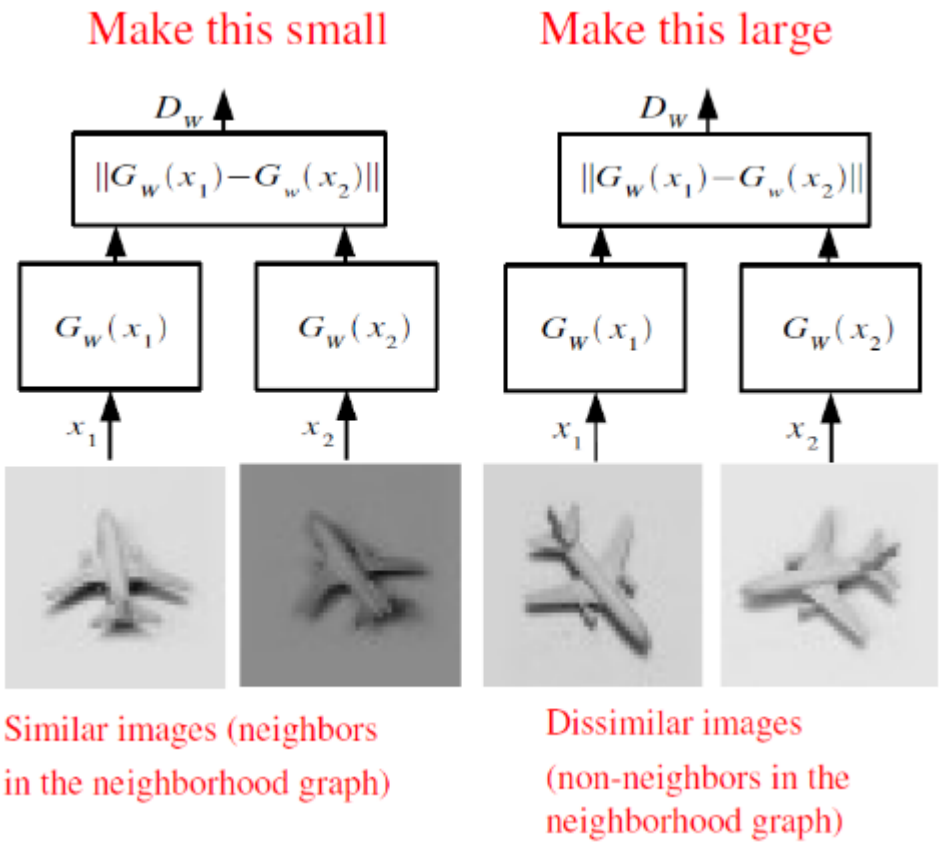
Siamese is the name of the general model architecture where the model consists of two identical subnetworks that compute some kind of representation vectors for two inputs and a distance measure is used to compute a score to estimate the similarity or difference of the inputs. In the attached figure, the LSTMa and LSTMb share parameters (weights) and have identical structure. The idea itself is not new and goes back to 1994.

MaLSTM (Manhattan LSTM) just refers to the fact that they chose to use Manhattan distance to compare the final hidden states of two standard LSTM layers. Alternatives like cosine or Euclidean distance can also be used, but the authors state that: "**Manhattan distance slightly outperforms other reasonable alternatives such as cosine similarity**".

Also, you can trivially swap LSTM with GRU or some other alternative if you want.



It **projects data into a space in which similar items are contracted and dissimilar ones are dispersed over the learned space.** It is computationally efficient since **networks are sharing parameters.**



http://www.erogol.com/duplicate-question-detection-deep-learning/

Siamese network tries to ***contract instances belonging to the same classes and disperse instances from different classes in the feature space.***

9. Bidirectional Encoder Representations from Transformers (BERT) with cosine distance

Please this is just a summary of this excellent article

The language modeling tools such as ELMO, GPT-2 and BERT allow for obtaining **word vectors** that **morph knowing their place and surroundings.**

The pre-trained BERT models can be downloaded and they have scripts to run BERT and get the word vectors from any and all layers. The base case BERT model that we use here

employs **12 layers** (transformer blocks) and yields word vectors with $p = 768$. The script getBertWordVectors.sh below reads in some sentences and generates word embeddings for each word in each sentence, and from every one of 12 layers.

We pull the embeddings from the 11th layer (Bert As a Service uses the 11th layer)

Our goal here is to show that the BERT word vectors morph themselves based on context. Take the following three sentences for example.

- record the ***play***
- ***play*** the record
- ***play*** the game

The word *play* in the second sentence should be more similar to *play* in the third sentence and less similar to *play* in the first. We can come up with any number of triplets like the above to test how well BERT embeddings do. Here are a bunch of such triplets and the results show that BERT is able to figure out context the word is being used in.

BERT embeddings are contextual. Each row show three sentences. The sentence in the middle expresses the same context as the sentence on its right, but different from the one on its left. All three sentences in the row have a word in common. The numbers show the computed cosine-similarity between the indicated word pairs. BERT

embedding for the word in the middle is more similar to the same word on the right than the one on the left.

When classification is the larger objective, there is no need to build a BoW sentence/document vector from the BERT embeddings. The [CLS] token at the start of the document contains a representation fine tuned for the specific classification objective. But for a clustering task we do need to work with the individual BERT word embeddings and perhaps with a BoW on top to yield a document vector we can use.

10. A word about Knowledge-based Measures

This part is a summary from this amazing article

Knowledge-based measures quantify semantic relatedness of words using a semantic network. Many measures have shown to work well on the WordNet large lexical database for English. The figure below shows a subgraph of WordNet.

- **WordNet**, which is a large lexical database of English. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. WordNet's structure makes it a useful tool for computational linguistics and natural language processing.

Here we consider the problem of **embedding entities** and **relationships** of **multi-relational data in low-dimensional vector spaces**, its objective is to propose **a**

canonical model which is easy to train, contains a reduced number of parameters and can scale up to very large databases.

TransE, [TransE — Translating Embeddings for Modeling Multi-relational Data] is an energy-based model for learning low-dimensional embeddings of entities. In TransE, relationships are represented as translations in the embedding space:

For example,

- **Wu and Palmer metric** measures the **semantic similarity of two concepts as their depth of the least common subsumer in the wordnet graph**

- **Resnik metric** estimates **the similarity as the probability of encountering the least common subsumer in a large corpus**. This probability is known as the Information Content (IC). Note that a concept, here, is a synset which is a word sense (synonym) and each word has several synsets. These examples are implemented in the Python NLTK module.

If you liked this article, consider giving it at least 50 👏 :)

SOURCES

Highly inspired from all these amazing notebooks, papers, articles, …

- http://nlp.town/blog/sentence-similarity/
- https://medium.com/mlreview/implementing-malstm-on-kaggles-quora-question-pairs-competition-8b31b0b16a07
- http://www1.se.cuhk.edu.hk/~seem5680/lecture/LSI-Eg.pdf
- https://markroxor.github.io/gensim/static/notebooks/WMD_tutorial.html
- https://www.machinelearningplus.com/nlp/cosine-similarity/
- http://poloclub.gatech.edu/cse6242/2018spring/slides/CSE6242-820-TextAlgorithms.pdf

- https://github.com/makcedward/nlp/blob/master/sample/nlp-word_embedding.ipynb
- http://robotics.stanford.edu/~scohen/research/emdg/emdg.html#flow_eqw_notopt
- http://robotics.stanford.edu/~rubner/slides/sld014.htm
- http://jxieeducation.com/2016-06-13/Document-Similarity-With-Word-Movers-Distance/
- http://stefansavev.com/blog/beyond-cosine-similarity/
- https://www.renom.jp/index.html?c=tutorial
- https://weave.eu/le-transport-optimal-un-couteau-suisse-pour-la-data-science/
- https://hsaghir.github.io/data_science/denoising-vs-variational-autoencoder/
- https://www.jeremyjordan.me/variational-autoencoders/
- https://www.kaggle.com/shivamb/how-autoencoders-work-intro-and-usecases
- http://www.erogol.com/duplicate-question-detection-deep-learning/

Machine Learning