MET CS688

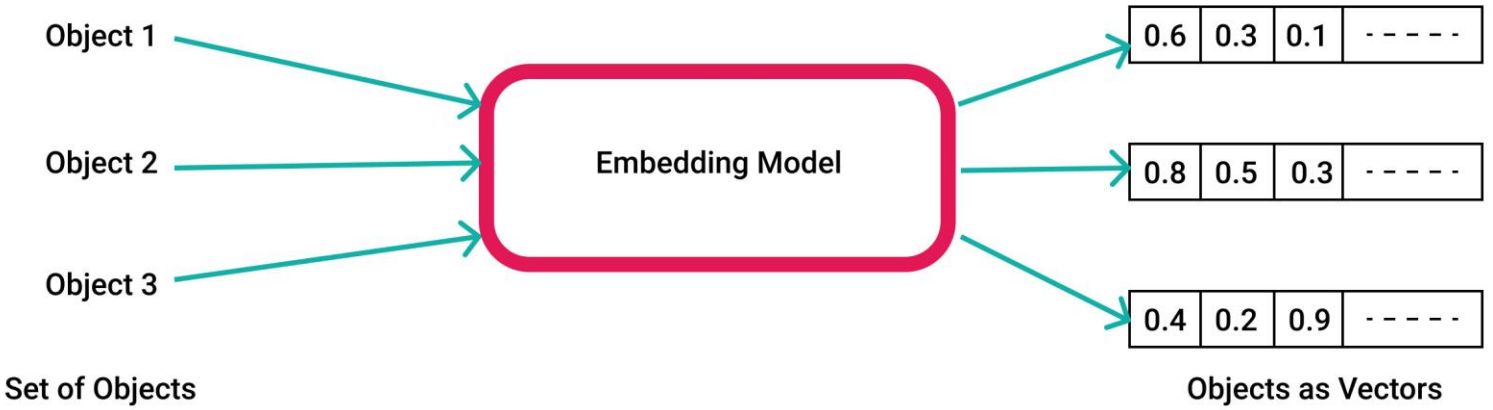
WEB ANALYTICS AND MINING

ZLATKO VASILKOSKI

WORD EMBEDDINGS

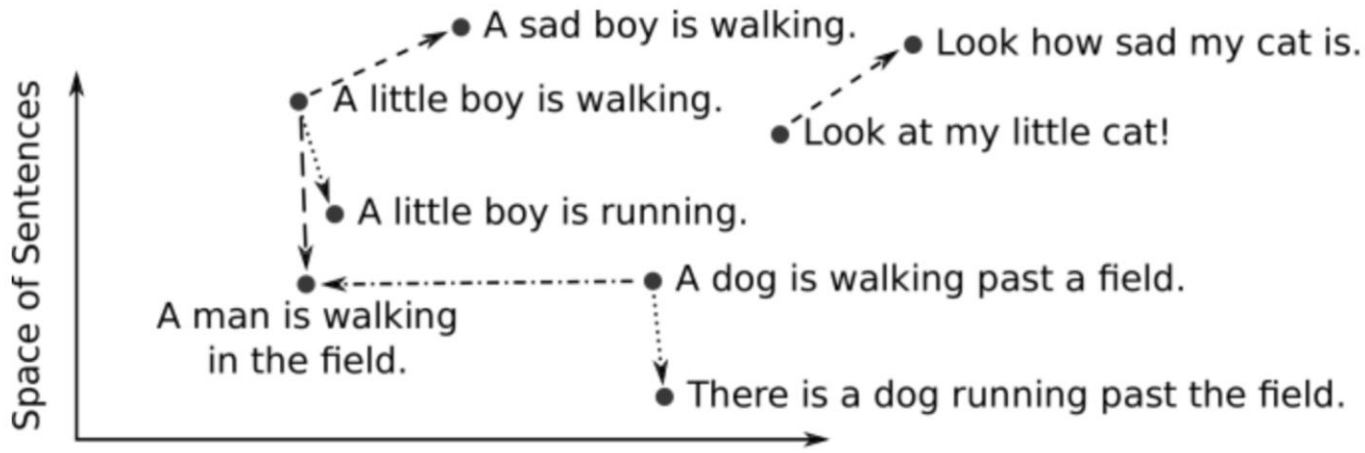
What are Vector Embeddings?

- Vector embeddings are one of the most fascinating and useful concepts in machine learning. They are central to many NLP, recommendation, and search algorithms.
- If you've ever used things like recommendation engines, voice assistants, language translators, you've come across systems that rely on embeddings.
- ML algorithms, like most software algorithms, need numbers to work with. Numeric values can easily be vectorized. How about something more abstract like an entire document of text?
- Objects – letters, n-grams, sentences, paragraphs or entire document of text can be reduced to a vector and then perform various operations with them.



What are Vector Embeddings?

- We need vector embeddings whose semantic similarity and concepts can be quantified by how close they are to each other as points in vector spaces.
- Then we can efficiently perform machine learning tasks such as clustering, recommendation, and classification.
  - In a **clustering** task, clustering algorithms assign similar points to the same cluster while keeping points from different clusters as dissimilar as possible.
  - In a **recommendation** task, when making recommendations for an unseen object, the recommender system would look for objects that are most similar to the object in question, as measured by their similarity as vector embeddings.
  - In a **classification** task, we classify the label of an unseen object by the major vote over labels of the most similar objects.

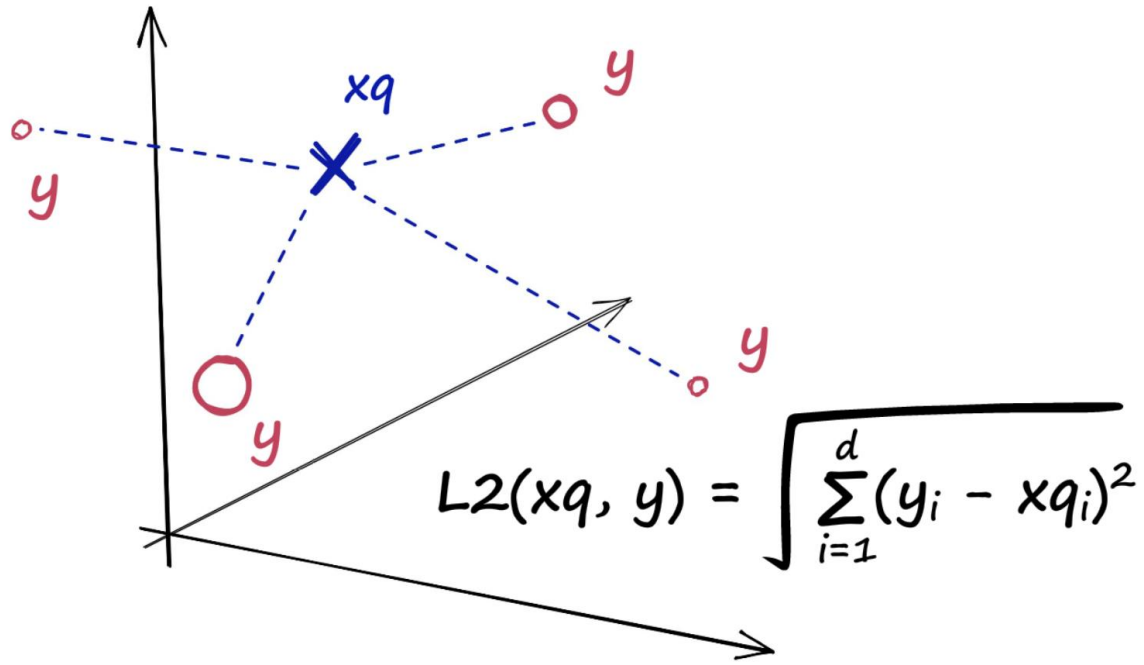


Source: DeepAI

Using Vector Embeddings

- The fact that embeddings can represent an object as a dense vector that contains its semantic information makes them very useful for a wide range of ML applications.

- Similarity search is one of the most popular uses of vector embeddings.
  - Algorithms like KNN and ANN calculate distance between vectors to determine similarity.
  - It can be used for NLP tasks like de-duplication, recommendations, anomaly detection.



L2 distance calculation between a query vector **xq** and our indexed vectors (shown as **y**)

Types of Vector Embeddings

Embedding models are a fundamental technique in natural language processing (NLP) and machine learning that represent high-dimensional categorical data (like words or items) in a lower-dimensional continuous space.

- This transformation allows machines to understand and process text more effectively.
- There are two main types of embedding models: **dense** and **sparse**.

In practice, dense embeddings are often preferred due to their superior performance in many NLP tasks.

However, there are situations where sparse embeddings may be more suitable, especially when

- interpretability or
- computational efficiency is a concern.

Types of Vector Embeddings

Dense Embedding Models

- **Continuous representation:** Each item is represented by a dense, fixed-length vector.
- **No explicit meaning:** The components of the vector don't necessarily have direct semantic meaning.
- **Learned from data:** The embeddings are learned from the data itself, often using techniques like word2vec, GloVe, or FastText.
- **Advantages:**
  - Efficient storage and computation.
  - Can capture semantic relationships between items.
  - Often perform well in various NLP tasks.
- **When to Use:**
  - When you have a large vocabulary or a high-dimensional feature space.
  - When you want to capture semantic relationships between items.
  - For tasks like **text classification**, **machine translation**, and **recommendation systems**.

Sparse Embedding Models

- **One-hot encoding:** Each item is represented by a vector with a single non-zero element.
- **Explicit meaning:** The position of the non-zero element corresponds directly to the item's identity.
- **No learning:** No training is required to create sparse embeddings.
- **Advantages:**
  - Simple and intuitive.
  - Can be used directly with many machine learning algorithms.
  - Suitable for categorical data with a relatively small number of unique values.
- **When to Use:**
  - When you have a small vocabulary or a low-dimensional feature space.
  - When you need to preserve the identity of individual items.
  - For tasks like **collaborative filtering** and **categorical feature encoding**.

What is Collaborative Filtering?

Collaborative filtering is a popular technique used in **recommendation systems** to predict a user's preferences for items based on the preferences of similar users. It assumes that if two users have similar tastes in the past, they are likely to have similar tastes in the future.

How Collaborative Filtering Works:

1. **User-Item Matrix:** A matrix is created where rows represent users and columns represent items. The cells in the matrix indicate whether a user has interacted with (e.g., rated, purchased, clicked on) a particular item.

2. **Similarity Calculation:** The similarity between users or items is calculated. Common similarity metrics include:
    1. **Pearson correlation coefficient:** Measures the linear relationship between two variables.
    2. **Cosine similarity:** Measures the angle between two vectors.
    3. **Jaccard similarity:** Measures the similarity between sets.

3. **Recommendation Generation:** Based on the calculated similarities, recommendations are generated. For example:
    1. **User-based collaborative filtering:** Recommendations are made based on the preferences of similar users. If a user is similar to another user who has rated an item highly, the system recommends that item to the first user.
    2. **Item-based collaborative filtering:** Recommendations are made based on the similarity between items. If an item is similar to another item that a user has rated highly, the system recommends the similar item to the user.

Collaborative Filtering

Advantages of Collaborative Filtering:

- **No domain knowledge required:** The system learns user preferences directly from the data.
- **Scalable:** Can handle large datasets with many users and items.
- **Effective:** Often provides accurate recommendations.

Disadvantages of Collaborative Filtering:

- **Cold-start problem:** Difficulty recommending items to new users or for new items.
- **Sparsity problem:** The user-item matrix is often sparse, making it difficult to find similar users or items.
- **Scalability limitations:** Can become computationally expensive for very large datasets.

Hybrid Approaches:

- To address the limitations of collaborative filtering, hybrid approaches often combine it with other techniques, such as content-based filtering or knowledge-based filtering.

Embedding Models

Some specific embedding models:

- **Word2vec:**
  - A family of models that learn word embeddings by predicting surrounding words in a text corpus.
- **FastText:**
  - An extension of word2vec that learns character-level embeddings and can handle out-of-vocabulary words.
- **GloVe:**
  - A model that learns word embeddings by factoring a co-occurrence matrix.
- **BERT:**
  - A bidirectional encoder representations from transformers model that has achieved state-of-the-art results on various NLP tasks.
- **FSE (Fast Sentence Embeddings):**
  - A technique used to represent entire sentences as numerical vector that captures the semantic meaning. Very fast and efficient method. Popular techniques for creating FSE:
    - **Word Embeddings:** Using pre-trained word embeddings (like Word2Vec or GloVe) and averaging or concatenating them to represent the sentence.
    - **Sentence-Level Embeddings:** Training models specifically designed to learn sentence-level representations, such as: **Doc2Vec:** A model that learns distributed representations of documents and words.
    - **Transformers:** Neural network architectures that have become very popular for NLP tasks, such as BERT and RoBERTa.

Word-level Embeddings

Word-level embeddings represent individual words as vectors in a vector space. These embeddings are based on the idea that words with similar meanings appear in similar contexts.

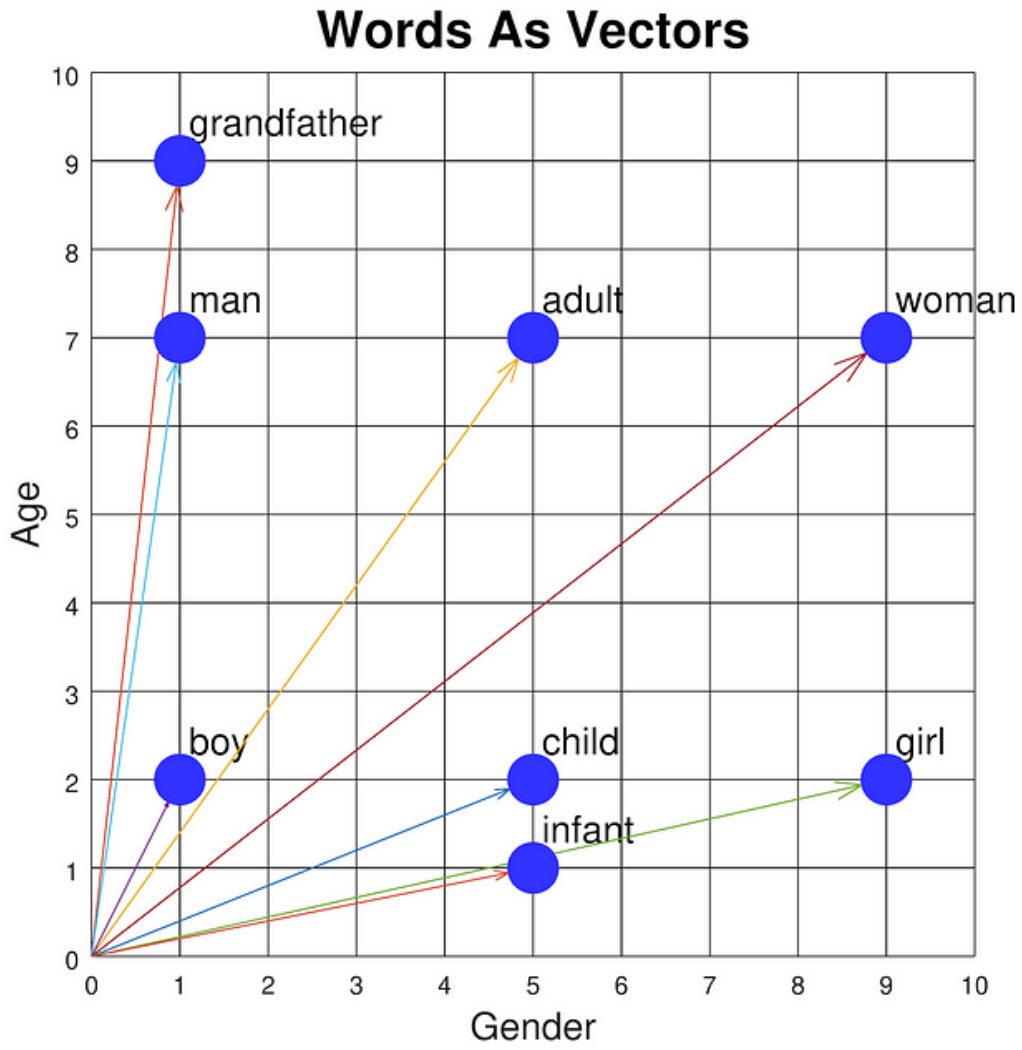
How it works: Each word in the vocabulary gets its own vector.

- Words that appear in similar contexts tend to have similar vectors.
- Relationships like word similarity or analogy can be computed by measuring distances between vectors (e.g., cosine similarity).

Word2Vec Example:

Word.	Embedding (Vector)
king.	[0.21, -0.45, ..., 0.12]
queen.	[0.19, -0.47, ..., 0.14]
apple.	[0.78, 0.24, ..., -0.35]
...	...

similarity(king, queen) ≈ 0.92



Sentence-level Embeddings

Sentence embeddings represent entire sentences (or paragraphs) as vectors. These embeddings aim to capture the meaning of the entire sentence, accounting for the order and relationships between words.

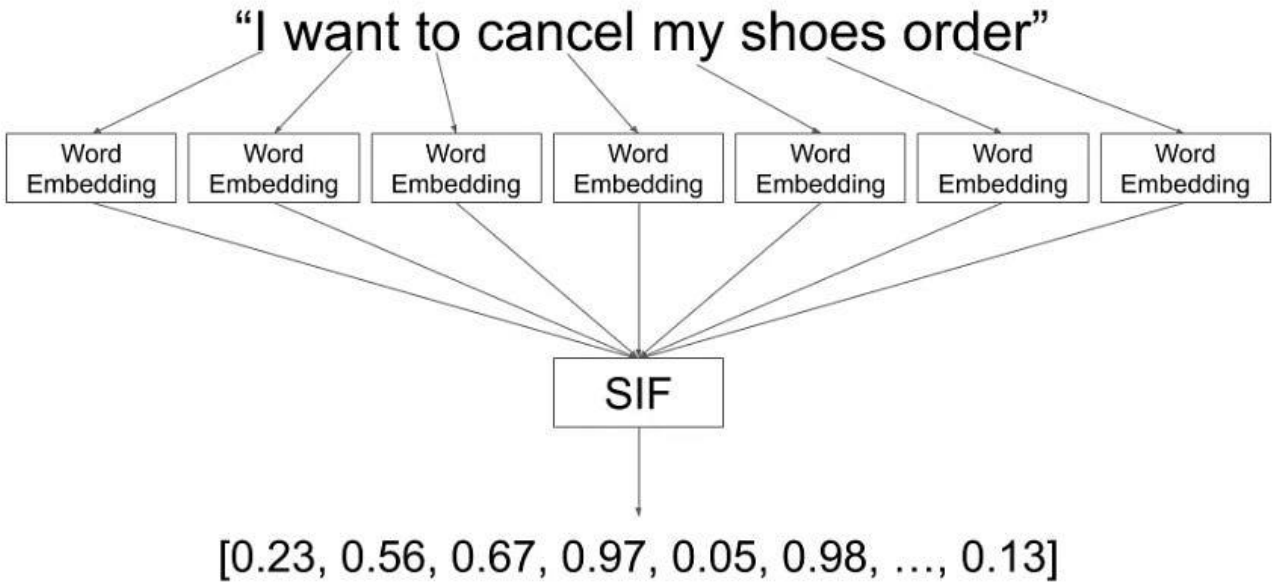
- Each sentence gets its own vector.

How it works: Each word in the vocabulary gets its own vector.

- The sentence embedding captures the context and meaning of the sentence.
- These embeddings are often used in tasks like sentence similarity, paraphrase detection, and document classification.

Weakness

- Requires more computational resources compared to word embeddings. Longer sentences may lose some granularity.

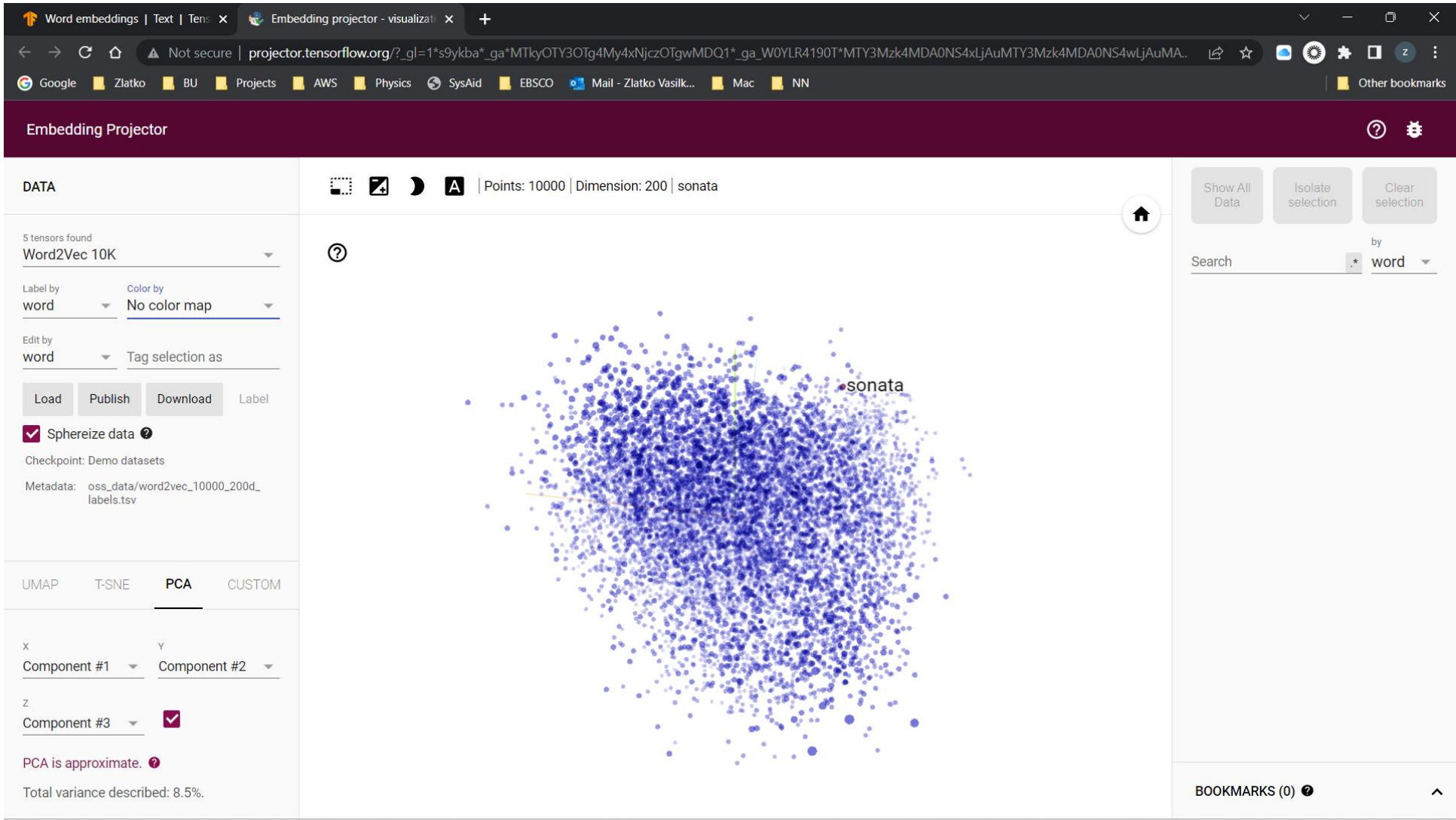


Creating Vector Embeddings

- One way of creating vector embeddings is to engineer the vector values using domain knowledge.
  - In medical imaging, we use medical expertise to quantify a set of features such as shape, color, and regions in an image that capture the semantics.
- However, engineering vector embeddings requires domain knowledge, and it is too expensive to scale.
- Instead of engineering vector embeddings, commonly a deep neural network models is trained to translate objects to vectors.
- The resulting embeddings are usually
  - High dimensional (up to two thousand dimensions) and
  - Dense (all values are non-zero).
- Text embedding - Typically used **models** to obtain vector embeddings for words, sentences, or paragraphs:
  - **Word2Vec, Doc2Vec, GLoVE, and BERT**
- Image embedding - using deep neural network models such as convolutional neural networks (CNNs)

Representing Text as Numbers

- https://www.tensorflow.org/text/guide/word_embeddings
- http://projector.tensorflow.org/?_gl=1*s9ykba*_ga*MTkyOTY3OTg4My4xNjczOTgwMDQ1*_ga_W0YLR4190T*MTY3Mzk4MDA0NS4xLjAuMTY3Mzk4MDA0NS4wLjAuMA



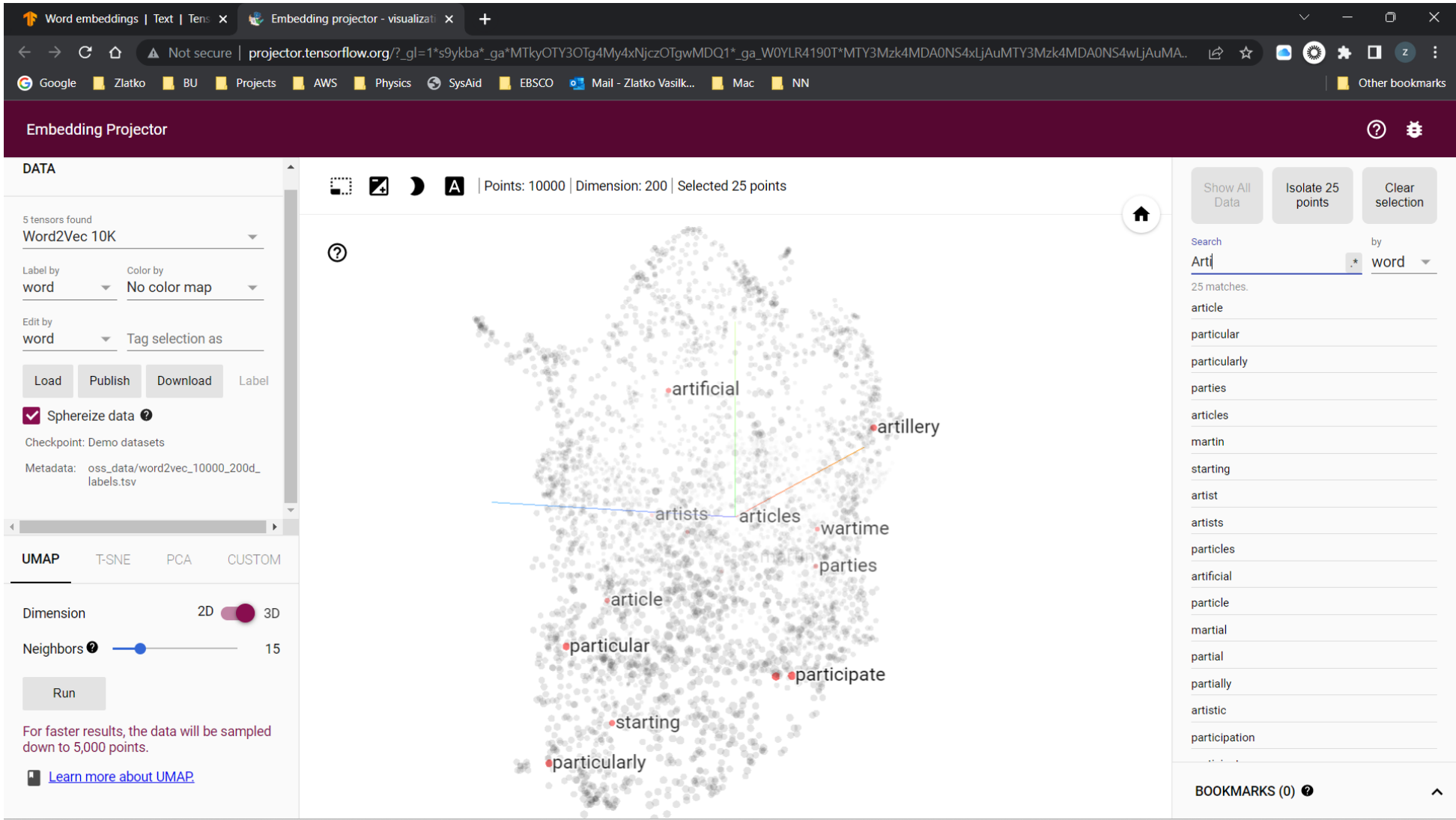
Representing Text as Numbers

Representing Text as Vectors - Embedding Projector using Word2Vec:

http://projector.tensorflow.org/?_gl=1*s9ykba*_ga*MTkyOTY3OTg4My4xNjczOTgwMDQ1*_ga_W0YLR4190T*MTY3Mzk4MDA0NS4xLjAuMTY3Mzk4MDA0NS4wLjAuMA..

The search for "Arti" results in this Representation of Similarity.

You can train your own word embeddings using a simple Keras model (for a sentiment classification Task), and then visualize them in this Embedding Projector.



Word Embeddings

But introducing neural networks to text and language also requires encodings of the words, which is something that people struggled for decades to achieve in a satisfactory manner.

Representing text as numbers

- Machine learning models take vectors (arrays of numbers) as input. When working with text, the first thing that needs to be done is to come up with a strategy to convert strings to numbers (or to "vectorize" the text) before feeding it to the model.
- This vectorization is called **word/sentence/text embedding**.

Consider these three embedding strategies.

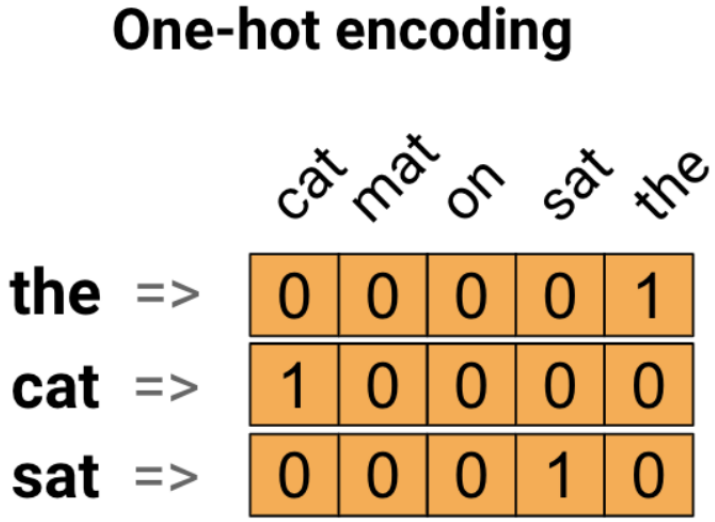
1. One-hot encodings - encode each word with a binary number
2. Integer (Dense) encodings - encode each word with a unique number (integer)
3. Vector encodings (embeddings) - encode each word with a vector

1) One-hot Encodings

The simplest approach is to "one-hot" encode each word in our vocabulary.

- Consider the sentence
  - "The cat sat on the mat".

- The vocabulary (or unique words) in this sentence is
  - cat,
  - mat,
  - on,
  - sat,
  - the



- To represent each word, you will create a zero vector with length equal to the vocabulary, then place a one in the index that corresponds to the word, as shown in the diagram.
- To create a vector that contains the encoding of the sentence, you could then concatenate the one-hot vectors for each word.
- This approach is inefficient. A one-hot encoded vector is sparse (meaning, most indices are zero).
  - To one-hot encode each word in the vocabulary, the created vector would have 99.99% of the elements are zero.

1) Term-Document Incidence Matrix

Terms \ Documents	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	…
Antony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
…							

- The result is a binary term-document incidence matrix, as shown above. The rows of the term-document incidence matrix are made up of terms (words), and the columns of the term-document incidence matrix are made up of documents (plays in this case).

- In a term-document incidence matrix an element (*t*, *d*) is 1 if the play (the document) in column *d* contains the word (term) in the row *t*, and is 0 otherwise.
- We can see that the document "Julius Caesar" does not contain the term "Cleopatra" thus the entry "0" in the term-document incidence matrix.

1) Simple Use of TDM in Text Mining

- **The Boolean Retrieval Model** – This is one of the basic models used to retrieve information from the term-document incidence matrix.
  - Uses term-document incidence matrix so it avoids linear search of all documents.
  - Uses bitwise (binary) logical operations which are the fastest possible.

Illustration:

To answer the query Brutus AND Caesar AND NOT Calpurnia, we take the vectors for Brutus, Caesar and Calpurnia, complement (negate) the binary vector for Calpurnia, and then do a bitwise AND:

(110100) AND (110111) AND (101111) = (100100)

- So, this query is contained in the first and the fourth document (see previous slides).

1) How efficient is this sparse encoding?

Illustration:

- Suppose we have N = 1 million documents (or a collection - units we use to build an information retrieval system from).
- Suppose each document is about W=1000 words long (2–3 book pages). Assuming an average of 6 bytes per word including spaces and punctuation, then this is a document collection of about 6 GB in size. Assume we have M = 500,000 distinct terms.
- This gives a matrix of zeros and ones of size M*N=5 10^{11}, which is too big to keep and analyze.
- What can we do about this?
- Note that the DTM-matrix is extremely sparse (only few nonzero entries). The maximum number of ones is W*N=10^9, which is much smaller than M*N5 10^{11} (total number of 1's and 0's). Or at least 99.8% in M*N will be 0's.

- What is the best way to deal with sparse matrices?
  - Indexing!

2) Integer (Dense) Encodings

Encode each word using a unique number.

- Continuing the example above, you could assign 1 to "cat", 2 to "mat", and so on. You could then encode the sentence "The cat sat on the mat" as a dense vector like [5, 1, 4, 3, 5, 2]. This approach is efficient. Instead of a sparse vector, you now have a dense one (where all elements are full).

There are two downsides to this approach, however:

- The integer-encoding is arbitrary (it does not capture any relationship between words).
- An integer-encoding can be challenging for a model to interpret. A linear classifier, for example, learns a single weight for each feature. Because there is **no relationship** between the similarity of any two words and the similarity of their encodings, this feature-weight combination is not meaningful.

2) Dense Encoding

Another possible approach is to encode each word with a unique number.

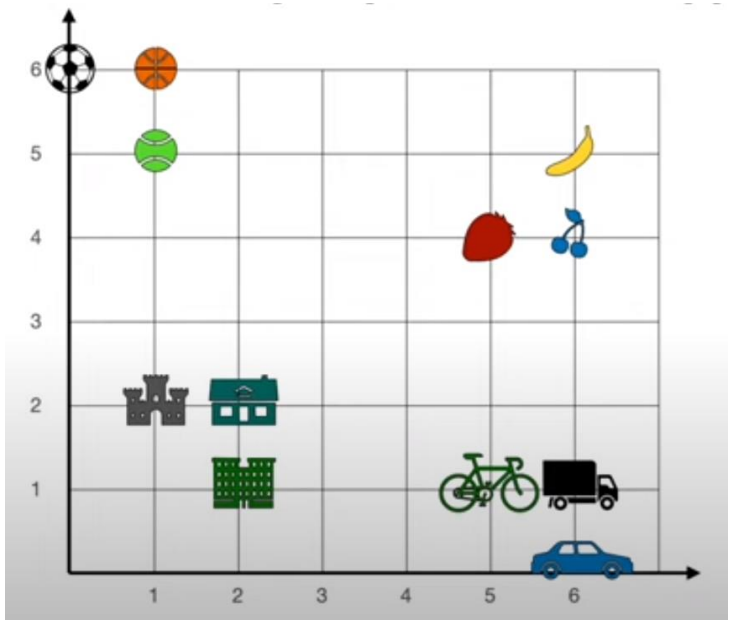
- Using the previous sentence example,
  - "*The cat sat on the mat*".
- We could assign 1 to "cat", 2 to "mat", and so on. We could then encode the sentence as a dense vector (with no zero elements)
  - "*The cat sat on the mat*".
  - [5, 1, 4, 3, 5, 2].
- This approach is efficient and instead of a **sparse** vector (**with** zero elements), we now have a **dense** vector (**with no** zero elements).

- 2D Matrix representation

  Whew would you encode the word “Apple”?

  (5,5) seems like a good spot, since **similar** (fruit) **words** would have **similar encoding**!

Word	Numbers	
Apple	?	?
Banana	6	5
Strawberry	5	4
Cherry	6	4
Soccer	0	6
Basketball	1	6
Tennis	1	5
Castle	1	2
House	2	2
Building	2	1
Bicycle	5	1
Truck	6	1
Car	6	0

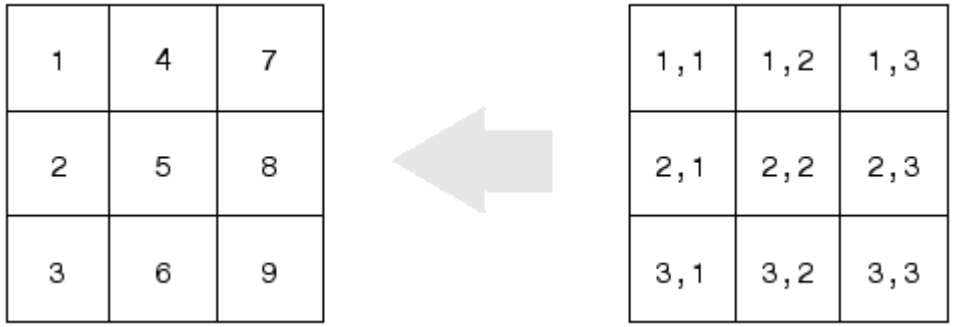


2) Dense Encoding

- **Indexed Notation of a Matrix** – TDM is sparse, more memory economical (thus faster) to keep just the nonzero entries.

Illustration:

- Better representation is to record only the position of 1's. Example of a mapping from subscript (right) to linear indexes notation (left) for a 3-by-3 matrix.



- In the case of having a large sparse matrix the advantage of the index notation is in the fact that only the indices of the nonzero elements need to be kept.
- How many indices do you need to keep track of the nonzero entries in the matrix A?

$$A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

2) Dense Encoding

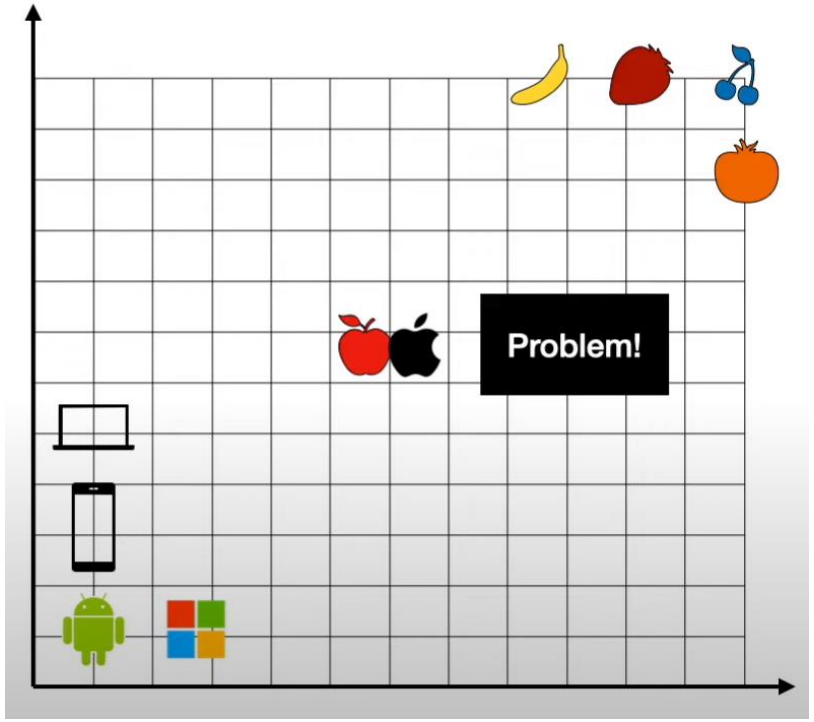
However, there are two downsides to this approach:

- The integer-encoding is arbitrary (it does not capture any relationship (semantics) between words).
- An integer-encoding can be challenging for a model to interpret.
- A linear classifier, for example, learns a single weight for each feature. Because there is no relationship between the similarity of any two words and the similarity of their encodings, this feature-weight combination is not meaningful.

Same words, such as “Apple” have different meaning.

How can we solve the semantic (meaning) ambiguity of the words?

- Knowledge Graphs
- Vector encoding would perform better!
- Adding Attention even better performance!



Word	Numbers	
Apple	5	5
Soccer	0	6
House	2	2
Car	6	0

Word	Numbers			
A	-0.82	-0.32	...	0.23
Aardvark	0.419	1.28	...	-0.06
...			...	
Zygote	-0.74	-1.02	...	1.35

4096

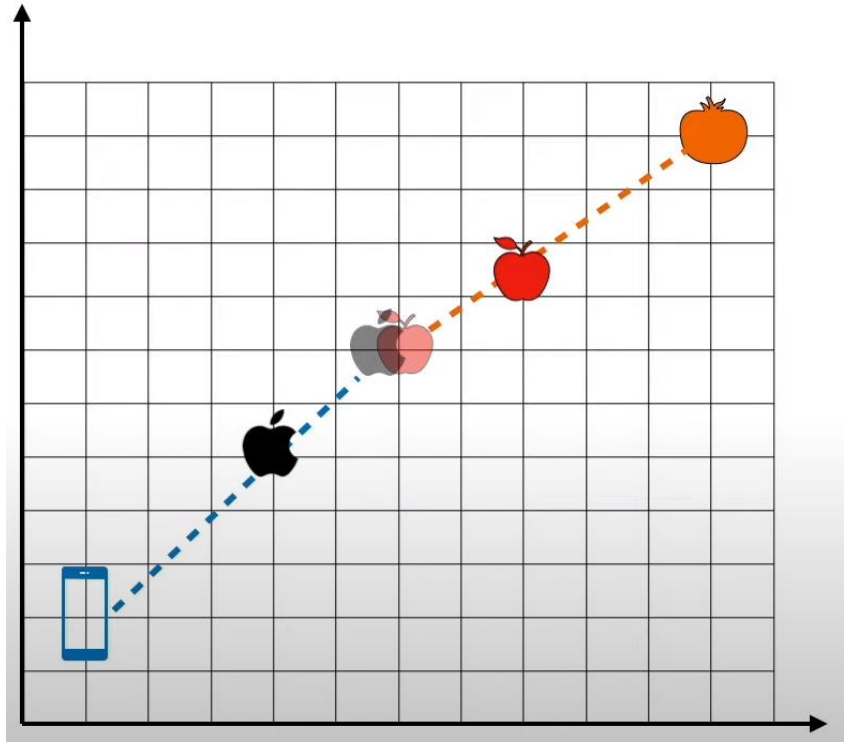
2) Semantic Ambiguity & Attention

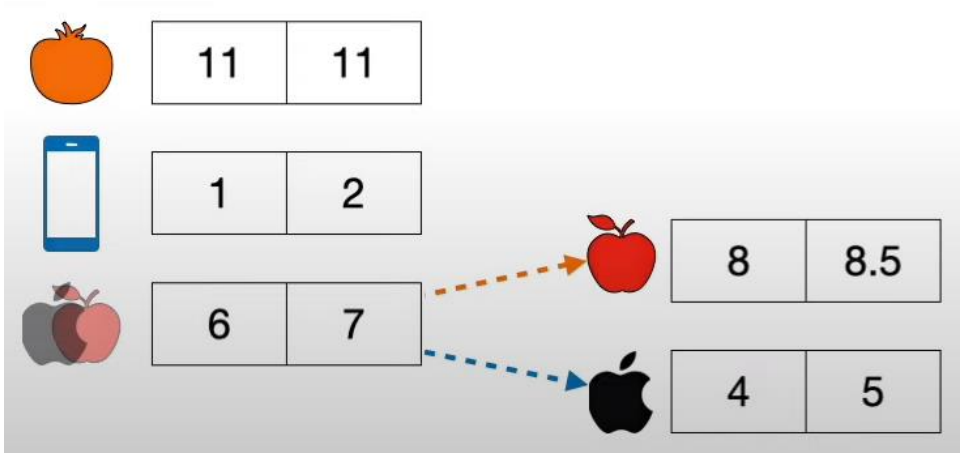
Attention uses the context of the sentence (other words in the sentence) in order to resolve the embedding ambiguity.

"*do you like apples or oranges?*"

"*do you like apple phone*"

What helps us resolve the semantics of the word "apple" in these 2 sentences are the words "orange" and "phone".

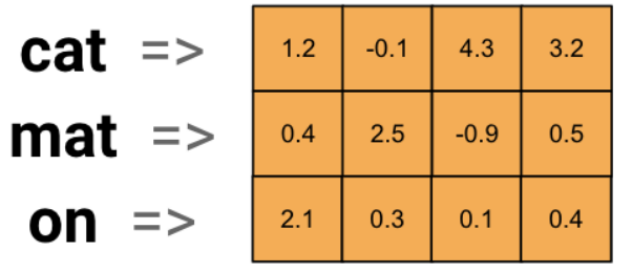




3) Vector Encodings (Embeddings)

Word embeddings give us a way to use an efficient, dense representation in which **similar words have a similar encoding**. Importantly, you do not have to specify this encoding by hand. An embedding is a dense vector of floating-point values (the length of the vector is a parameter you specify – 4 in the example below).

A 4-dimensional embedding



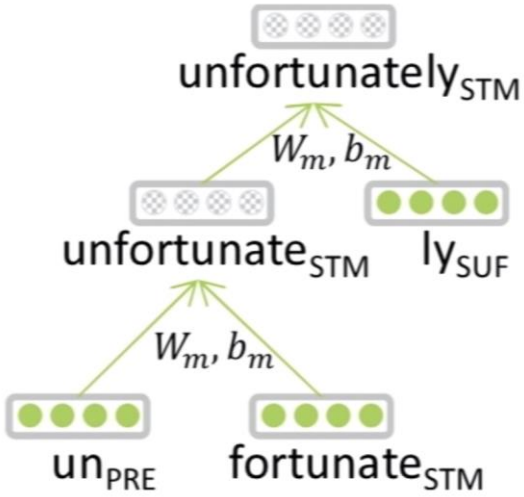
Instead of specifying the values for the embedding manually, they are trainable parameters (weights learned by the model during training, in the same way a model learns weights for a dense layer).

It is common to see word embeddings that are 8-dimensional (for small datasets), up to 1024-dimensions when working with large datasets. A higher dimensional embedding can capture fine-grained relationships between words but takes more data to learn.

Above is a diagram for a word embedding. Each word is represented as a 4-dimensional vector of floating point values. Another way to think of an embedding is as "lookup table". After these weights have been learned, you can encode each word by looking up the dense vector it corresponds to in the table.

Word Embeddings

- Represent the meaning of the word as a higher dimensional vector space. Expectation is that words with similar meaning will cluster together.



- Words made of **morphems** that we represent as vectors.
  - prefix stem suffix
  - un interest ed
- Vocabulary – localistic representation – vector [0, 0, 0, 0, 1,0,0, 0, 0]
  - Does not give relationship between words (i.e. motel vs hotel)
    - motel = [0,0,0,0,0,1,0,0,0]
    - hotel = [0,0,1,0,0,0,0,0,0]
    - These two vectors are orthogonal (their dot product is zero) – not that good!
- We would like a representation that translates the semantic ***similarity*** as perceived by humans to ***proximity*** in a vector space.

Vector Embeddings for Documents

- Vector Embeddings are used as feature generation tools for text documents.
    1. Trained Word-Embeddings, such as Skipgram, CBOW, GloVe, fastText can be downloaded from the web.
    2. By applying packages such as genism, word-embeddings can easily be trained from an arbitrary collection of texts.
    3. Training of a word embedding then can be integrated into an end-to-end neural network for a specific application, such as for document-classification.
- The typically used vector embedding methods are:
    - Based on word frequency per document over the entire corpus
        - **Bag of Words**
        - **TF-IDF** (term frequency–inverse document frequency)
    - **Word2Vec** (sliding window over the entire corpus)
        - **CBOW**
        - **Skip-gram**
    - **SIF** (Smooth Inverse Frequency)

1) Vector Embedding - Bag of Words Model

- Consider the following 2 documents:
  - **Doc_1 = {“The cat meows”, “Cat purrs”}**
  - **Doc_2 = {“The dog barks”, “My dog also likes to bark”}**

- **Bag of Words Model** – text document is represented as a
  - Bag of its word frequency
  - Disregarding grammar and
  - The order of words

- The Bag of Words for these 2 documents:
  - **BoW_1 = {“The”:1, “cat”:2, “meows”:1, “purrs”:1}**
  - **BoW_2 = {“The”:1, “dog”:2, “My”:1, “also”:1, “likes”:1, “to”:1, “bark”:1, “barks”:1}**

2) Vector Embedding - TF-IDF Model

- **TF-IDF** (Term Frequency (relative frequency) – Inverse Document Frequency) – statistics intended to reflect how important a word is to a document in a collection or corpus. Mathematically it is

$$TF(\text{relative frequency}) = \frac{\#\ times\ the\ word\ \boldsymbol{t}\ appears\ in\ a\ Document}{Total\ \#\ of\ terms\ in\ the\ same\ Document}$$

$$IDF = \ln \frac{Tot\ \#\ of\ Docs}{\#\ of\ Docs\ with\ word\ \boldsymbol{t}}$$

$$TFIDF\ =\ TF * IDF$$

- TF-IDF value increases proportionally to the number of times a word appears in the document
  - TF-IDF is offset by the number of documents in the corpus that contain that word
  - This adjust for the fact that some words appear more frequently in general.

Doc_1 = {"The cat meows", "Cat purrs"}
Doc_2 = {"The dog barks", "My dog also likes to bark"}

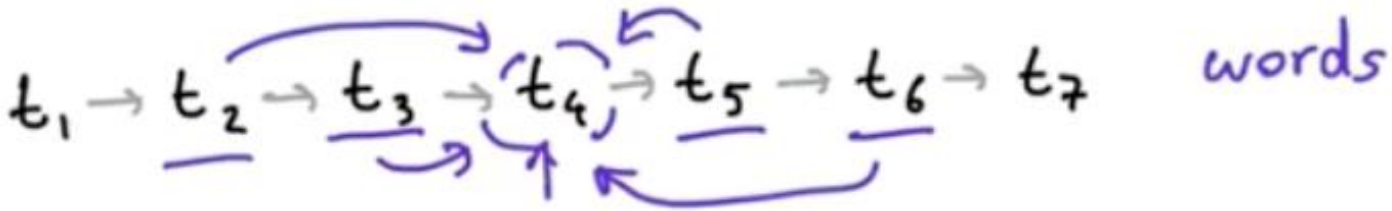
- For example:
  - The word "**The**" appears with frequency 1 in both **Doc_1** and **Doc_2** documents.
  - Because **Doc_2** has more words, the relative frequency is different
    - **TF_1 = 1/5**
    - **TF_2 = 1/9**
  - The IDF is the same (1/1) for both documents, which makes IDF=0 since *ln(1)=0*, thus we can conclude that that this word is not very informative as it appears in all documents.
  - The words **"cat"** and **"dog"** are more interesting, and they have nonzero IDF. Note that semantically we would consider **"cat"** & **"dog"** the same!
    - **"cat"; TF_1 = 2/5; IDF=2/1 => TF-IDF = 2/5*ln(2) = 0.28**
    - **"dog"; TF_1 = 2/9; IDF=2/1 => TF-IDF = 2/9*ln(2) = 0.15**

```
> TFIDF
          Cat  Dog
Doc_1 0.28 0.00
Doc_2 0.00 0.15
```

3) Vector Embedding - Word2vec Model

- **Word2Vec** (sliding window over the entire corpus) – look at the context of the surrounding words.
- First proposed by Bengio et al. in 2003 (no deep learning yet).
- More recently - feed this objective to deep learning to get the word embeddings

- Two methods to do this
  1. **CBOW** (Continuous Bag of Words) – prediction from the window of surrounding context words **without any order** (bag-of-words assumption).
  2. **Skip-gram** – **weighs nearby** context words more heavily than more distant context words.



Word Embedding - Word2vec Model

Vord2Vec algorithm is based on distribution similarity – looks at the context of the surrounding words w_c for each center word w_t.

- First proposed by Bengio et al. in 2003 (no deep learning yet).
- A sliding window is implemented over the entire corpus. For each center word w_t, a probability of a context word w_c is defined.
- More recently - feed this objective to deep learning to get the word embeddings

Distribution similarity – look at the context of the surrounding words (**word embeddings**) – represent words as vectors

The algorithm has these 3 steps:

1. $p(w_c|w_t)$ - Define probability of a context word w_c for a given center word w_t (up to some window size).
2. $J = 1 - p(w_c|w_t)$- Define loss function J.
3. Keep adjusting the vector representations of words to minimize the loss function.

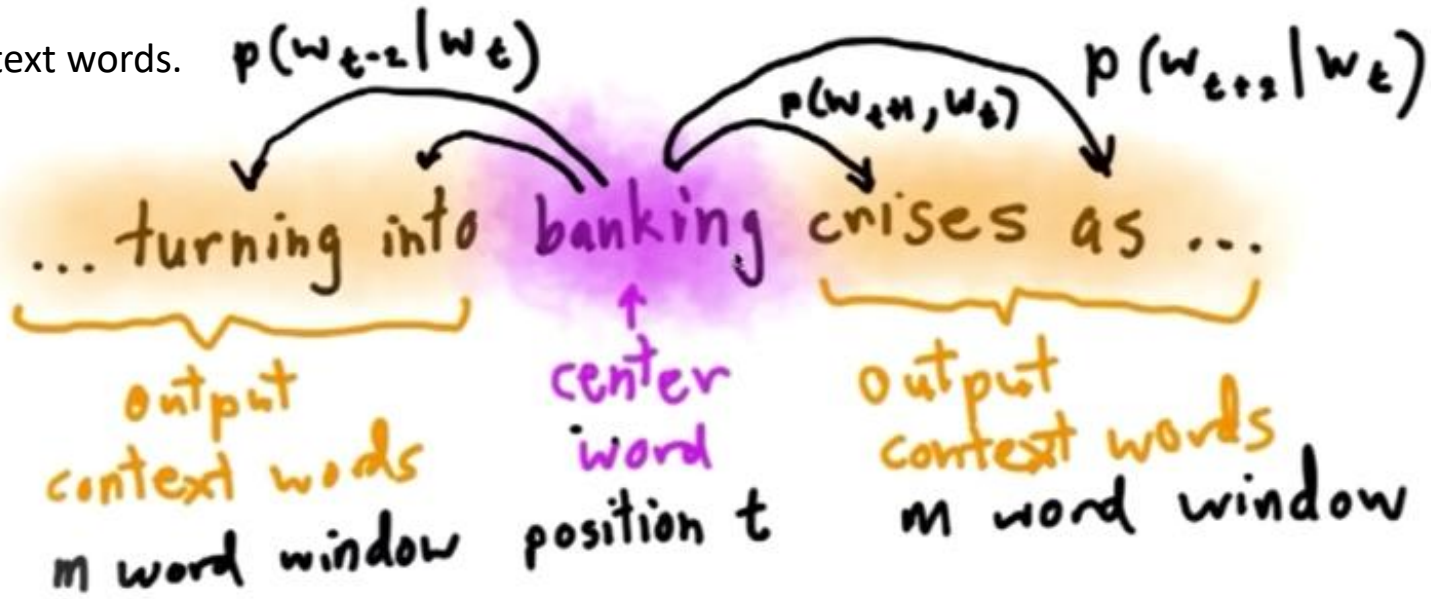
Vord2Vec – A prediction for each word what are its context words.

It has two algorithms:

1. **Skip-gram** – probability of surrounding words
2. **Bag Of Words** – just add vectors in a sentence

It has two training algorithms:

1. Hierarchical Softmax
2. Negative sampling



3) Vector Embedding - Word2vec Model

For each word $t = 1 \cdots N$ predict surrounding words in a window of m context words to the left and to the right of the word w_t.

J' – Objective (loss or cost) function. Maximize the probability of any context word given the center word w_t. Typically, this is done on a very large amount of text (such as entire Wikipedia content).

$$J'(\theta) = \prod_{t=1}^{N} \prod_{-m \le j \le m} p(w_{t+j}|w_t;\theta)$$

Here $p(w_{t+j}|w_t)$ is the **probability of co-occurrence**, and θ is a parameters of the model (a vector representation of the words). These vectors (the parameter θ) are being adjusted to have the probabilities $p(w_{t+j}|w_t;\theta)$ of the context words to be as high as possible.

In practice, instead of maximizing the objective function $J'(\theta)$, it is more practical to minimize (negative sign) the average $(1/N)$:

$$J(\theta) = -\frac{1}{N}\sum_{t=1}^{N} \sum_{-m \le j \le m} \log\left(p(w_{t+j}|w_t;\theta)\right)$$

$J(\theta)$ is called **negative log likelihood** (uses cross-entropy loss). How to minimize **negative log likelihood**?

- Use **Softmax** form, which is a standard way of turning numbers into probability distribution.

3) Vector Embedding - Word2vec Model

Softmax finds the probability estimate of word w_o using word w_c.

Note: Each word has 2 vector representations, one as **center word** (v) and another as **context word** (u), so $\theta = \begin{bmatrix} v_{hotel} \\ \vdots \\ u_{hotel} \\ \vdots \end{bmatrix}$

Matrix representation of the skip-gram model gives the probability p_i of context word u_i for given center word w_t.

$$w_t^T V = [0 \cdots 1 \cdots 0] \begin{bmatrix} \vdots & v_1 & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & v_N & \vdots \end{bmatrix} = \begin{bmatrix} v_1 \\ \vdots \\ v_N \end{bmatrix} = v_c \quad \text{next} \quad U^T v_c = \begin{bmatrix} \cdots & \cdots & \cdots \\ u_1 & \cdots & u_N \\ \cdots & \cdots & \cdots \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_N \end{bmatrix} = \begin{bmatrix} u_1^T \cdot v_c \\ u_i^T \cdot v_c \\ u_N^T \cdot v_c \end{bmatrix} = \begin{bmatrix} z_1 \\ z_i \\ z_N \end{bmatrix} = z$$

w_t - center word (one-hot vector representation i.e. $[0,0,0,0,0,1,0,0,0]^T$).

V – matrix representation of the *center* words. Each column contains word's embedding (the vector representation).

U – matrix representation of the *context* words. Each column contains word's embedding (the vector representation).

When w_t multiplies matrix V it actually selects the column of V that represents the vector representation of the center word w_t.

The elements of vector z are the dot product $z_i = u_i^T \cdot v_c$. A dot product is a loose measure of similarity, how similar u_i is to v_c.

The dot product is larger when the two vectors are more similar to each other.

3) Vector Embedding - Word2vec Model

Softmax form is a standard way of turning numbers into probability distribution and it follows these steps:

- We turn vector z into Softmax (e^{z_i}).
  - Exponentiation turns even negative z values into positive and makes larger values very large (max).
- Then we turn e^{z_i} into a probability distribution by normalizing it with the sum $\sum_j e^{z_j}$ (a partition function).

$$p(u_i|v_c) = \frac{e^{u_i^T v_c}}{\sum_{j=1}^{N} e^{u_j^T v_c}} \quad \text{or} \quad p_i = \frac{e^{z_i}}{\sum_j e^{z_j}} = [0.1,\ 0.3, 0.4, 0.2]^T$$

Note that the word indices i and c belong to the entire vocabulary of N words! They do not need to be next to each other in a same sentence.

If we have ground truth (train data, the vectors for w_{t+j}) then we subtract p_i from $w_{t+j} = [0,\ 0, 1, 0]^T$ and this is the loss.

Optimization by defining all the parameters (the vectors v_i and u_i) as one vector $\theta = \begin{bmatrix} v_i \\ u_i \end{bmatrix}$ where $i = 1 \cdots N$.

- We want to change the vectors in θ in such a way to minimize negative log likelihood $J(\theta)$.
- Having the probability$p(u_i|v_c)$ we find the derivative $\partial J/\partial v_c$ of $J = -\frac{1}{N}\sum_{t=1}^{N}\sum_{-m\le j\le m}\log\left(p\left(w_{t+j}|w_t\right)\right)$. The result is

$$\frac{\partial J}{\partial v_c} = u_i - \sum_{j=1}^{N} p(u_j|v_c)u_j = 0$$

Condition - when u_i balances with the probability of every possible word appearing as a context word for given venter word v_c.

3) Vector Embedding - Word2vec Model

Optimization by Gradient Descent

Once we have the objective function and its derivatives, we can implement the Gradient Descent algorithm to optimize.

$$\theta_j^{new} = \theta_j^{old} - \alpha \frac{\partial}{\partial \theta_j^{old}} J(\theta)$$

For large amount of data very slow. Instead, Stochastic Gradient Descent used. Gradient estimate is done only for one of the parameters. This is a very rough estimate and does not guarantee move toward min, but in practice, when implemented in neural networks works very well and much faster.

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J_t(\theta)$$