CHAPTER 4

Natural Language Processing
Word Embeddings

Summary

1. Introduction
2. Discrete text representations
3. Distributed/Continuous text representations
4. Word2vec
5. GloVe
6. fastText
7. Examples in Python

Definition of NLP

- NLP is a branch of artificial intelligence that deals with the interaction between computers and humans using the natural language.
- The ultimate objective of NLP is to read, decipher, understand, and make sense of the human languages in a manner that is valuable.
- Most NLP techniques rely on machine learning to derive meaning from human languages.
- It sits at the intersection of computer science, artificial intelligence, and computational linguistics

Applications

- Summarization
- Document similarity
- Sentiment classification
- Spam classification
- Translation
- Chat Bots
- Generative language
- …

spaCy

Gensim

NLTK

TensorFlow

Hugging Face

The problem of Text Representation

- Computers are brilliant when dealing with numbers.
- They are faster than humans in calculations & decoding patterns by many orders of magnitude.
- But therefore …
  - But what if the data is not numerical?
  - What if it's language?
  - What happens when the data is in characters, words & sentences? How do we make computers process our language?
  - How does Alexa, Google Home & many other smart assistants understand & reply to our speech?

Text Representation

- The most basic step for the majority of NLP tasks is to convert words into numbers for machines to understand & decode patterns within a language.
- We call this step text representation.
- It plays a significant role in deciding features for your machine learning model/algorithm.
- Text representations can be broadly classified into two sections:
  - Discrete text representations
  - Distributed/Continuous text representations

Discrete text representation

- These are representations where words are represented by their corresponding indexes to their position in a dictionary from a larger corpus or corpora.
- Representations that fall within this category are:
  - One-Hot encoding
  - Bag-of-words representation (BOW)
  - Basic BOW — CountVectorizer
  - Advanced BOW — TF-IDF

One-Hot encoding

- It is a type of representation that assigns 0 to all elements in a vector except for one, which has a value of 1. This value represents a category of an element.
- If i had a sentence, “I love my dog”, each word in the sentence would be represented as below:

```
I → [1 0 0 0], love → [0 1 0 0], my → [0 0 1 0], dog → [0 0 0 1]
```

- The entire sentence is then represented as:

```
sentence = [ [1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1] ]
```

One-Hot encoding

- The intuition behind one-hot encoding is that each bit represents a possible category & if a particular variable cannot fall into multiple categories, then a single bit is enough to represent it

- As you may have grasped, the length of an array of word depends on the vocabulary size. This is not scalable for a very large corpus which could contain up to 100,000 unique words or even more.

Implementation

```
from sklearn.preprocessing import OneHotEncoder
import itertools
# two example documents
docs = ["cat","dog","bat","ate"]
# split documents to tokens
tokens_docs = [doc.split(" ") for doc in docs]
# convert list of of token-lists to one flat list of tokens
# and then create a dictionary that maps word to id of word,
all_tokens = itertools.chain.from_iterable(tokens_docs)
word_to_id = {token: idx for idx, token in enumerate(set(all_tokens))}
# convert token lists to token-id lists
token_ids = [[word_to_id[token] for token in tokens_doc] for tokens_doc in tokens_docs]
# convert list of token-id lists to one-hot representation
vec = OneHotEncoder(categories="auto")
X = vec.fit_transform(token_ids)
print(X.toarray())
```

OUTPUT

```
[[0. 0. 1. 0.]
[0. 1. 0. 0.]
[0. 0. 0. 1.]
[1. 0. 0. 0.]]
```

One-Hot encoding

Advantages of one-hot encoding:

- Easy to understand & implement

Disadvantages of one-hot encoding:

- Explosion in feature space if number of categories are very high
- The vector representation of words is orthogonal and cannot determine or measure relationship between different words
- Cannot measure importance of a word in a sentence but understand mere presence/absence of a word in a sentence
- High dimensional sparse matrix representation can be memory & computationally expensive

Bag-of-words representation

- Bag-of-words (BOW) representation as the name suggests intutively, puts words in a “bag” & computes frequency of occurrence of each word.
- It does not take into account the word order or lexical information for text representation

- The intuition behind BOW representation is that document having similar words are similar irrespective of the word positioning

Basic BOW — CountVectorizer

- The CountVectorizer computes the frequency of occurrence of a word in a document. It converts the corpus of multiple sentences (say product reviews) into a matrix of reviews & words & fills it with frequency of each word in a sentence

```
from sklearn.feature_extraction.text import
CountVectorizer
text = ["i love nlp. nlp is so cool"]
vectorizer = CountVectorizer()
# tokenize and build vocab
vectorizer.fit(text)
print(vectorizer.vocabulary_)
# Output: {'love': 2, 'nlp': 3, 'is': 1, 'so':
4, 'cool': 0}
# encode document
vector = vectorizer.transform(text)
# summarize encoded vector
print(vector.shape) # Output: (1, 5)
print(vector.toarray())
```

OUTPUT

```
[[1 1 1 2 1]]
```

Basic BOW — CountVectorizer

Advantage of CountVectorizer:

- CountVectorizer also gives us frequency of words in a text document/sentence which One-hot encoding fails to provide
- Length of the encoded vector is the length of the dictionary

Disadvantages of CountVectorizer:

- This method ignores the location information of the word. It is not possible to grasp the meaning of a word from this representation
- The intuition that high-frequency words are more important or give more information about the sentence fails when it comes to stop words like “is, the, an, I” & when the corpus is context-specific. For example, in a corpus about covid-19, the word coronavirus may not add a lot of value

Advanced BOW (TF-IDF)

- To suppress the very high-frequency words & ignore the low-frequency words, there is a need to normalize the “weights” of the words accordingly
- **TF-IDF representation:** Term frequency-inverse document frequency
- It is a product of 2 factors

$$TFIDF = TF(w, d) * IDF(w)$$

- Where, TF(w, d) is frequency of word ‘w’ in document ‘d’
- IDF(w) is the inverse of document frequency is the log between total number of documents N divided by the number of documents containing the word *‘w’*

$$IDF(w) = log(\frac{N}{|d \in D : w \in d|})$$

-

BOW vs TF-IDF

BOW

	ball	big	cat	moon	small	table	tree	window	zoo
D1	0	1	2	0	**1**	1	0	1	0
D2	0	0	0	0	**1**	1	0	1	0
D3	0	1	0	1	**1**	0	1	0	0

TF-IDF

	ball	big	cat	moon	small	table	tree	window	zoo
D1	0	0.17	0.95	0	**0**	0.17	0	0.17	0
D2	0	0	0	0	**0**	0.17	0	0.17	0
D3	0	0.17	0	0.47	**0**	0	0.47	0	0

Advanced BOW (TF-IDF)

- Intuition behind TF-IDF is that the weight assigned to each word not only depends on a words frequency, but also how frequent that particular word is in the entire corpus/corpora

- It takes the CountVectorizer presented before & multiplies it by the IDF score.

- The resultant output weights for the words from the process is low for very highly frequent words (like stop-words) & very low frequency words (noise terms)

Advanced BOW (TF-IDF)

```
from sklearn.feature_extraction.text import TfidfVectorizer
text1 = ['i love nlp', 'nlp is so cool',
'nlp is all about helping machines process language',
'this tutorial is on baisc nlp technique']
tf = TfidfVectorizer()
txt_fitted = tf.fit(text1)
txt_transformed = txt_fitted.transform(text1)
print ("The text: ", text1)
# Output: The text:  ['i love nlp', 'nlp is so cool',
# 'nlp is all about helping machines process language',
# 'this tutorial is on basic nlp technique']
idf = tf.idf_
print(dict(zip(txt_fitted.get_feature_names(), idf)))
```

OUTPUT

{'about': 1.916290731874155, 'all': 1.916290731874155,
'basic': 1.916290731874155, 'cool': 1.916290731874155,
'helping': 1.916290731874155, 'is': 1.2231435513142097,
'language': 1.916290731874155, 'love': 1.916290731874155,
'machines': 1.916290731874155, 'nlp': 1.0, 'on':
1.916290731874155,
'process': 1.916290731874155, 'so': 1.916290731874155,
'technique': 1.916290731874155, 'this': 1.916290731874155,
'tutorial': 1.916290731874155}

Advanced BOW

- Note the weightage of word ‘nlp’. Since it is present in all the sentences, it is given a low weightage of 1.0.
- Similarly, the stop-word ‘is’ is also given a comparatively low weightage of 1.22 since it is present in 3 out of 4 sentences given.

- Similar to CountVectorizer, there are various parameters that can be tweaked to achieve desired results.
- Some of the important parameters (apart from the text preprocessing parameters like lowercase, strip_accent, stop_words etc.) are max_df, min_df, norm, ngram_range & sublinear_tf.
- The impact of these parameters on outputs weights is beyond scope of this article & will be covered separately.

Advanced BOW

Advantages of TF-IDF representation:

- Simple, easy to understand & interpret implementation
- Builds over CountVectorizer to penalise highly frequent words & low frequency terms in a corpus. So in a way, IDF achieves in reducing noise in our matrix.

Disadvantages of TF-IDF representation:

- Positional information of the word is still not captured in this representation
- TF-IDF is highly corpus dependent. A matrix representation generated out of cricket data cannot be used for football or volleyball. Therefore, there is a need to have high quality training data

Conclussion of Discrete Representations

Advantages of discrete representations:

- Simple representations that are easy to understand, implement & interpret
- Algorithms like TF-IDF can be used to filter out uncommon & irrelevant words easily helping model train & converge faster

Disadvantages of discrete representations:

- The representation is directly proportional to vocabulary size. High vocabulary size can lead to memory constraints
- It does not leverage co-occurrence statistics between words. It assumes all words are independent of each other
- This leads to highly sparse vectors with few non zero values
- They do not capture the context or semantics of the word. It does not consider spooky & scary as similar but as two independent terms with no commonality between them

Applications of Discrete Representations

- Discrete representations are widely used across both classical machine learning & deep learning applications for solving complicated use cases like:
  - document similarity
  - sentiment classification
  - spam classification
  - topic modeling
- However, for more complex applications are not enought
- Next, we will discuss distributed or a continuous text representation of text & how it is better (or worse) than discrete representations.

Distributed Text Representation

- Distributed text representation is when the representation of a word is not independent or mutually exclusive of another word and their configurations often represent various metrics & concepts in data.
- So the information about a word is distributed along the vector it is represented as.
- This is different from discrete representation where each word is considered unique & independent of each others.
- Some of the commonly used distributed text representations are:
  - Co-Occurrence matrix
  - Word2Vec
  - GloVe
  - BERT

Co-Occurrence Matrix

- The co-Occurrence matrix, as the name suggests, considers the co-occurrence of entities nearby each other.
- The entity used could be a single word, could be a bi-gram or even a phrase.
- Predominantly, a single word is used for computing the matrix.
- It helps us understand the association between different words in a corpus.
- Let's look at an example using CountVectorizer used beforehand convert it into continuous representation.

Co-Occurrence Matrix

```
ssfrom sklearn.feature_extraction.text import CountVectorizer
docs = ['product_x is awesome',
        'product_x is better than product_y',
        'product_x is dissapointing','product_y beats product_x by miles',
'ill definitely recommend product_x over others']
# Using in built english stop words to remove noise
count_vectorizer = CountVectorizer(stop_words = 'english')
vectorized_matrix = count_vectorizer.fit_transform(docs)
# We can now simply do a matrix multiplication with the transposed
# image of the same matrix
co_occurrence_matrix = (vectorized_matrix.T * vectorized_matrix)
print(pandas.DataFrame(co_occurrence_matrix.A,
                       columns=count_vectorizer.get_feature_names(),
                       index=count_vectorizer.get_feature_names()))
```

OUTPUT

```
awesome  beats  better  definitely  dissapointing  ill  miles  \\
awesome              1       0       0              0                0     0
0
beats                0       1       0              0                0     0
1
better               0       0       1              0                0     0
0
definitely           0       0       0              1                0     1
0
dissapointing        0       0       0              0                1     0
0
ill                  0       0       0              1                0     1
0
miles                0       1       0              0                0     0
1
product_x            1       1       1              1                1     1
1
product_y            0       1       1              0                0     0
1
recommend            0       0       0              1                0     1
0
                product_x  product_y  recommend
awesome                 1          0          0
beats                   1          1          0
better                  1          1          0
definitely              1          0          1
dissapointing           1          0          0
ill                     1          0          1
miles                   1          1          0
product_x               5          2          1
product_y               2          2          0
recommend               1          0          1
```

Co-Occurrence Matrix

- The representation of each word is its corresponding row (or column) in the co-occurrence matrix

- If we want to understand the word associations to product_x, we can filter for the column & analyse that product_x is being compared with product_y & there are more positive adjectives associated to it than negative ones.

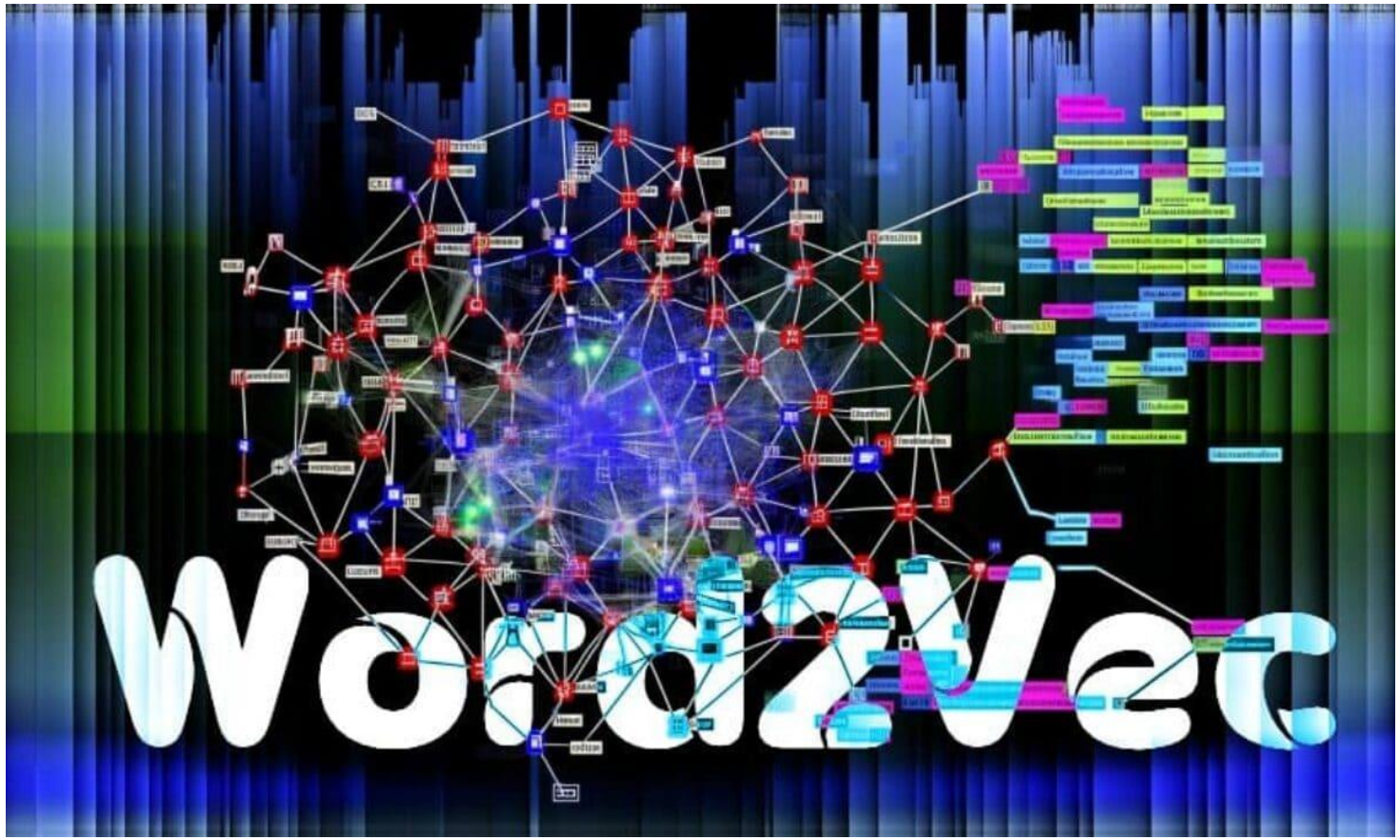
Co-Occurrence Matrix

Advantages:

- Simple representation for finding word associations
- It considers the order of words in the sentence unlike discrete techniques
- The representation coming out of this method is a global representation. i:e it uses the entire corpus for generating the representation

Disadvantages:

- Similar to CountVectorizer & TF-IDF matrices, this too is a sparse matrix. This means its not storage efficient & calculations are inefficient to run on top
- Larger the vocabulary size, larger the matrix size (not scalable to large vocabulary)
- Not all word associations can be understood using this technique. In the above example, if you look at the product_x column, there is a row with name beats. It is uncertain what is the context on beats in this scenario simply by looking at the matrix


Word2Vec

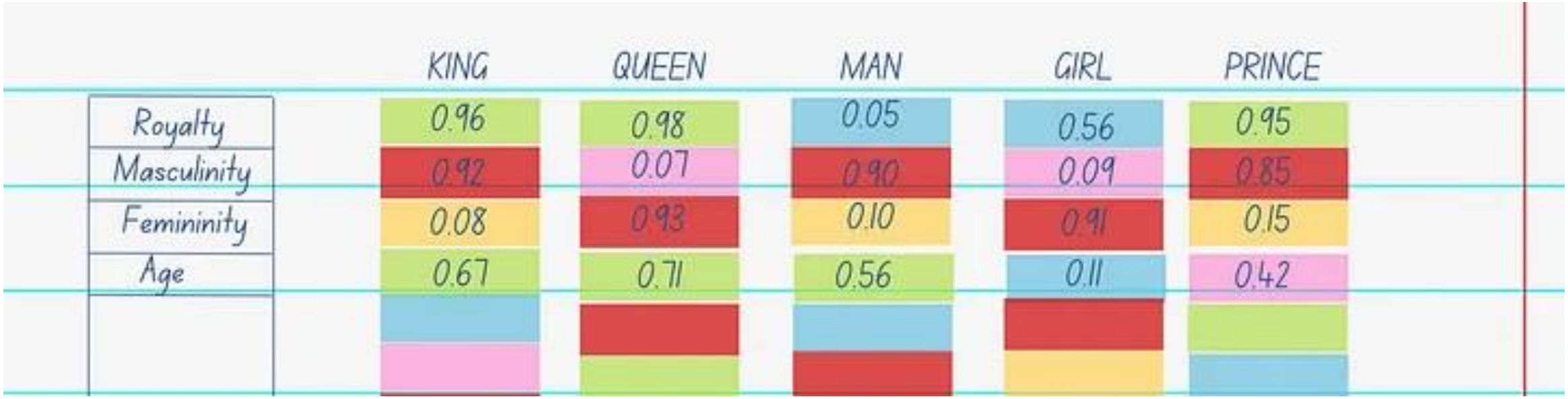
Word2Vec

- Word2Vec is a famous algorithm for representing word embeddings.
- It was developed by Tomas Mikalov in 2013 at Google.
- It's a prediction-based method for representing words rather than count based technique like co-occurrence matrix.
- Word embeddings are vector representation of a word.
- Each word in the vocabulary is represented by a fixed vector size while capturing its semantic & syntactic relation with other words

Word2Vec

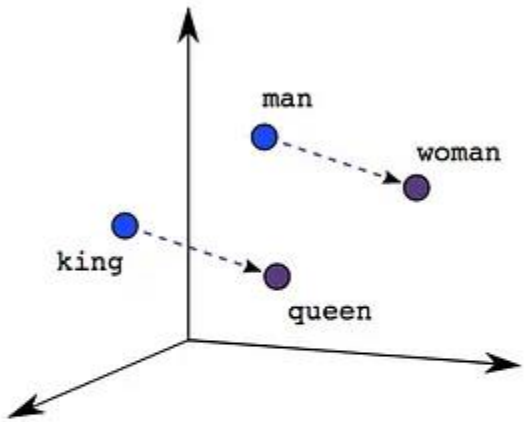
- Word2vec is a neural network approach to learn distributed word vectors in a way that:
  - words used in similar syntactic or semantic context
  - lie closer to each other in the distributed vector space.
- Distributed, in a way such that:
  - the result of a vector calculation vec(“Madrid”) — vec(“Spain”) + vec(“France”)
  - is closer to vec(“Paris”) than to any other word vector
- The quality of the representations was measured by how well it represents word similarity in the vector space using various distance metrics.

Word2Vec (example vector dim.= 4)

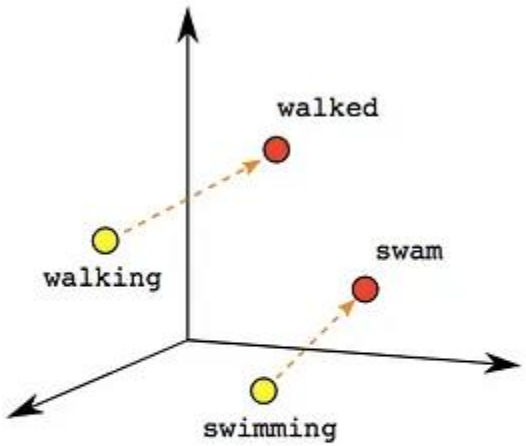


- As expected, “king”, “queen”, “prince” have similar scores for “royalty” and “girl”, “queen” have similar scores for “femininity”.
- An operation that removes “man” from “king”, would yield in a vector very close to “queen” (“king”- “man” = “queen”)
- Vectors “king” and “prince” have the same characteristics, except for age, telling us how they might possibly be semantically related to each other.

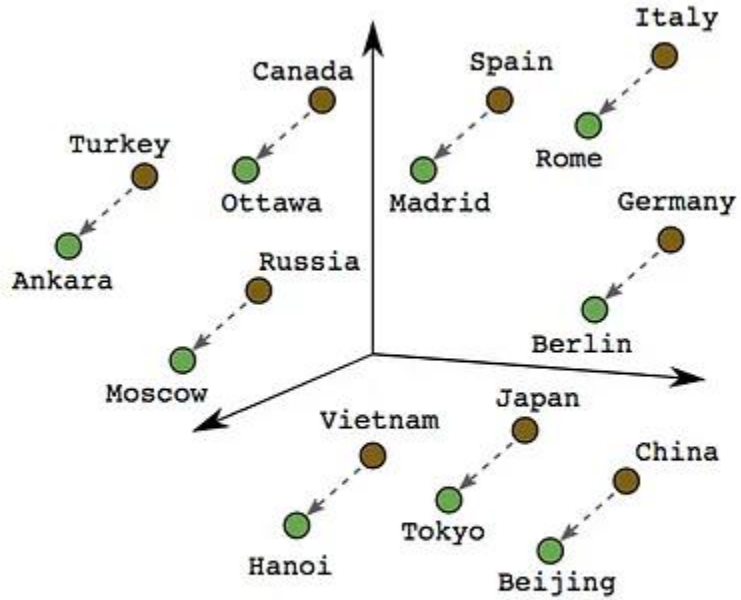
Word2vec



Male-Female



Verb Tense



Country-Capital

Word2Vec


mujer
hombre
reina
agua
rey
jirafa
tortuga
elefante
gato
perro
pájaro
banana
fruta
asia
oceanía

Word2Vec - Word similarity

- There are 2 ways to find the similarity between the vectors depending if they are normalised or not:
  - If normalised: We can compute the simple dot product between the vectors to find how similar they are
  - If not normalised: We can compute the cosine similarity between the vectors using the below formula

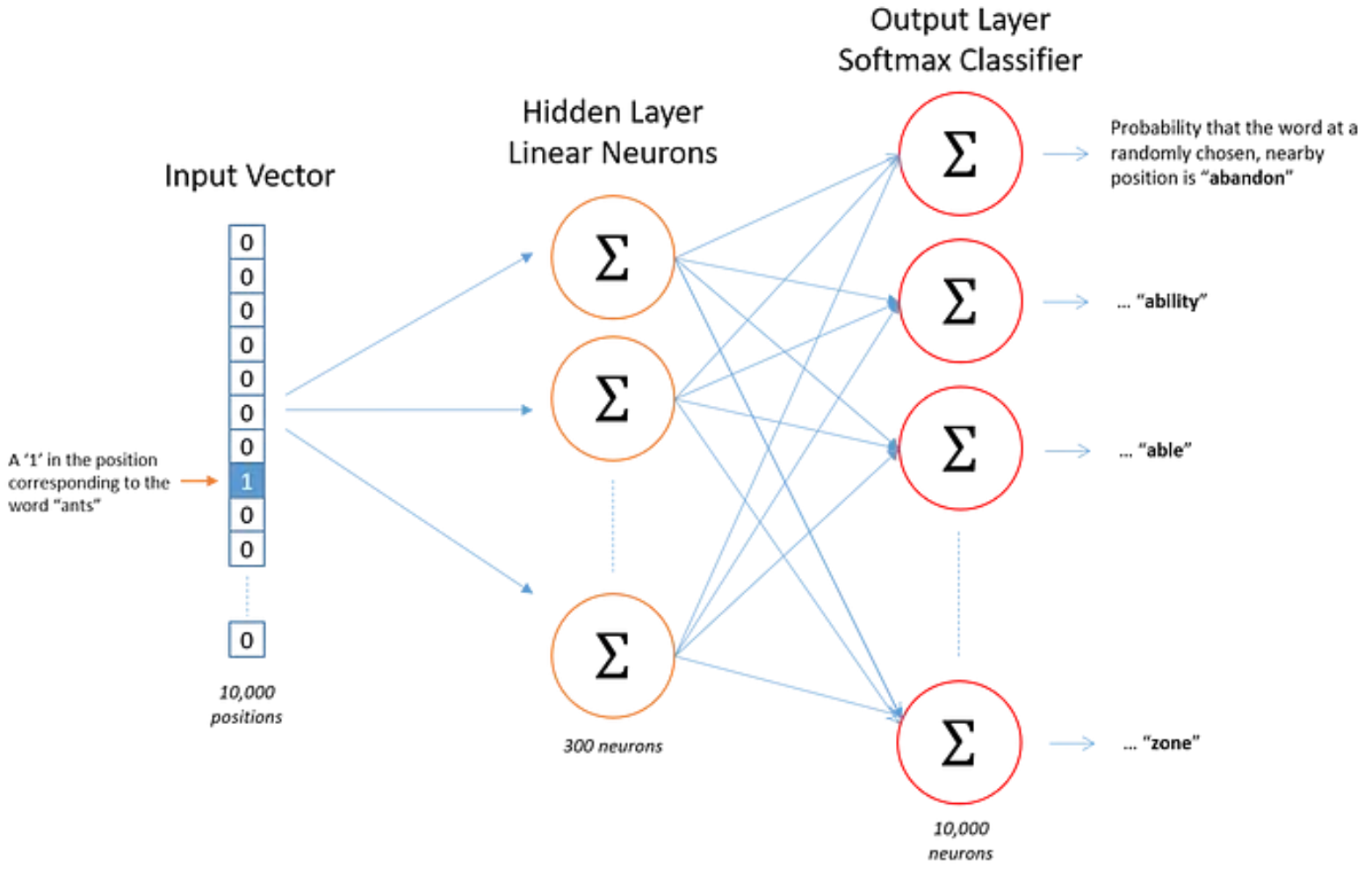
$$cosine\ similarity = \frac{u.v}{||u||_2\ |v||_2}$$

  - cosine distance = 1 – cosine similarity

Word2Vec - Model Architecture (i)

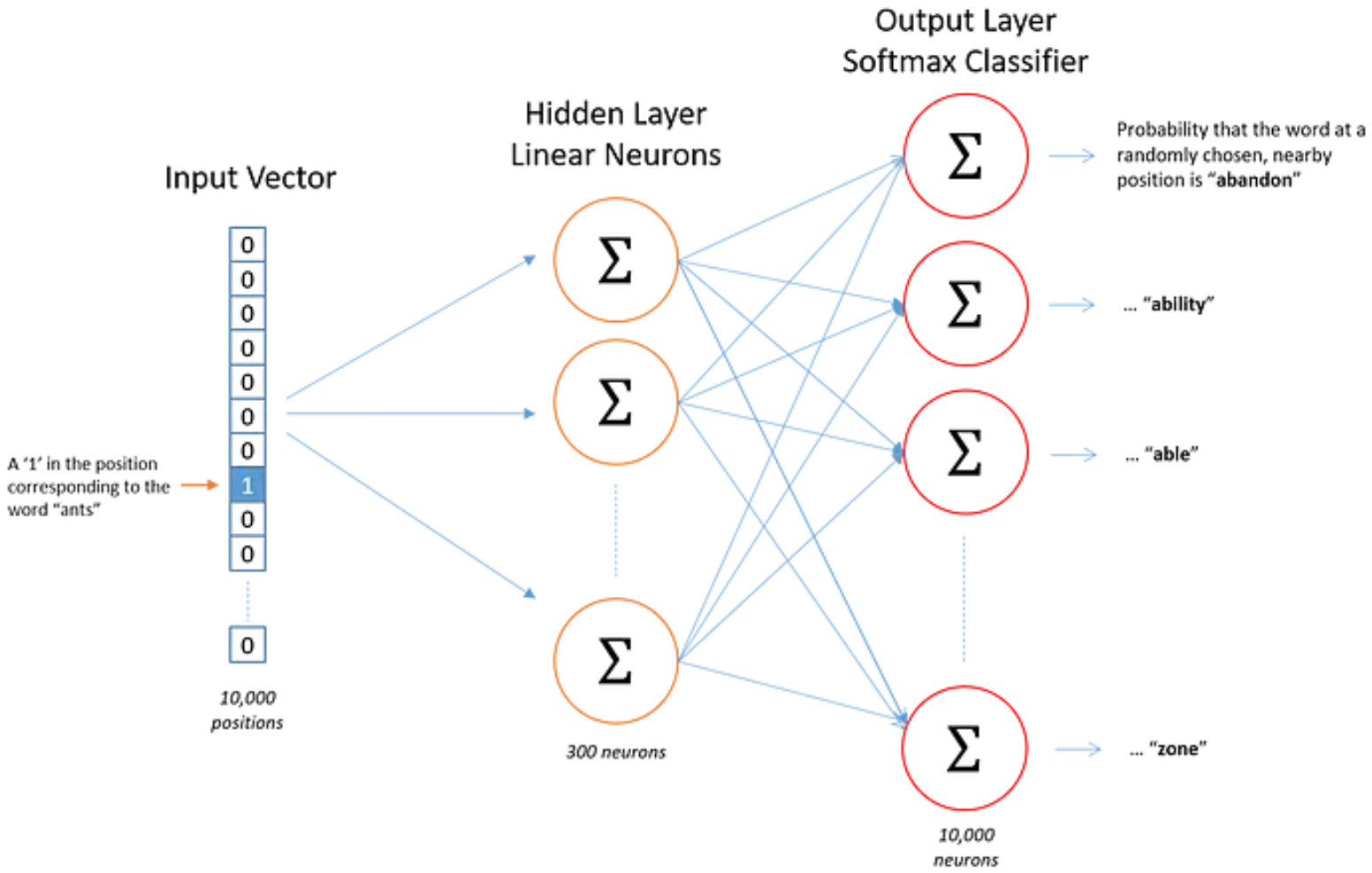
Word2Vec essentially is a shallow 2-layer neural network trained.

- The input contains all the documents/texts in our training set. For the network to process these texts, they are represented in a 1-hot encoding of the words.
- The number of neurons present in the hidden layer is equal to the length of the embedding we want. That is, if we want all our words to be vectors of length 300, then the hidden layer will contain 300 neurons.



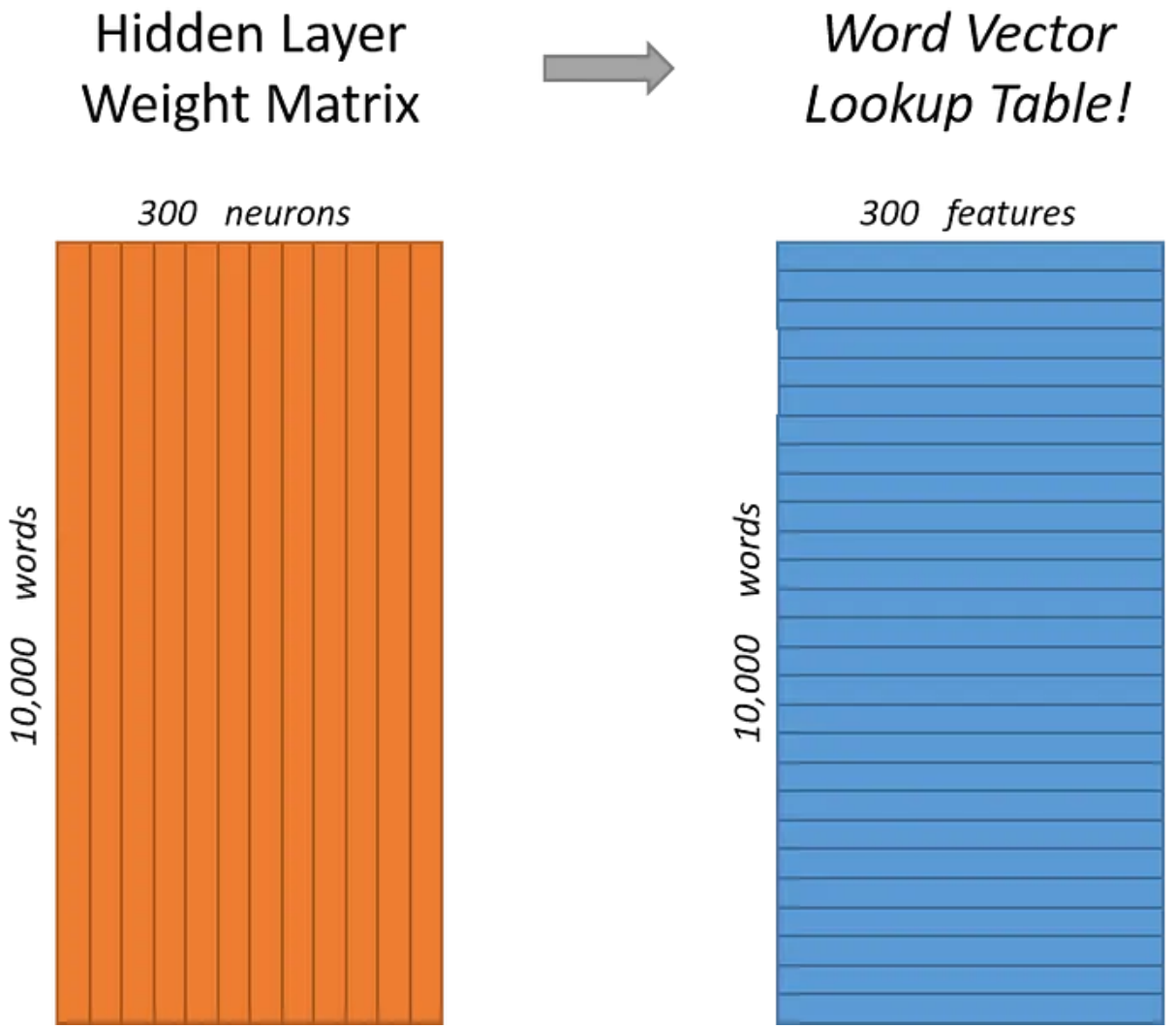
Word2Vec - Model Architecture (ii)

- The output layer contains probabilities for a target word (given an input to the model, what word is expected) given a particular input.
- At the end of the training process, the hidden weights are treated as the word embedding. Intuitively, this can be thought of as each word having a set of n weights (300 considering the example above) “weighing” their different characteristics (an analogy we used earlier).



Word2Vec - Model Architecture (iii)

- At the end of the training process, the hidden weights are treated as the word embedding.
- Intuitively, this can be thought of as each word having a set of n weights (300 considering the example above) “weighing” their different characteristics (an analogy we used earlier).



Word2vec – Two Techniques

- There are two main techniques of produce the vector respresentation of word (embedings).

1. Continuous Bag of Words (CBOW)
2. Skipgram

Word2vec - CBOW

- CBOW predicts the target-word based on its surrounding words.
- For example, consider the sentence, “The cake was chocolate flavoured”.
- The model will then iterate over this sentence for different target words, such as:
- “The ____ was chocolate flavoured” being inputs and “cake” being the target word.
- CBOW thus smoothes over the distribution of the information as it treats the entire context as one observation. CBOW is a faster algorithm than skipgrams and works well with frequent words.

Word2vec - CBOW

- CBOW aims to predict the target word based on its surrounding context words within a fixed window size.
- It works by summing or averaging the word vectors of the context words to predict the target word.
- For example, given the sentence "The cat sat on the ___", CBOW tries to predict the target word "mat" based on the context words "The", "cat", "sat", and "on".
- CBOW is computationally efficient and tends to work well with frequent words.

Word2vec - Skipgram

- Skipgram works in the exact opposite way to CBOW. Here, we take an input word and expect the model to tell us what words it is expected to be surrounded by.

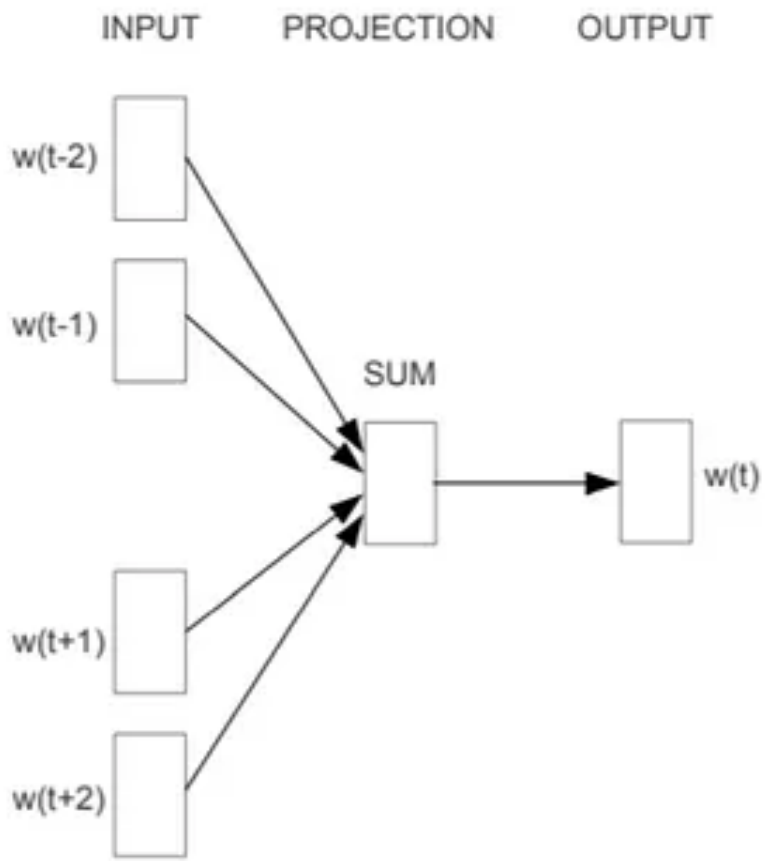
- Taking the same example, with “cake” we would expect the model to give us “The”, “was”, “chocolate”, “flavoured” for the given instance.

- The statistical interpretation of this is that we treat each context-target pair as a new observation. Skipgrams work well with small datasets and can better represent less frequent words.

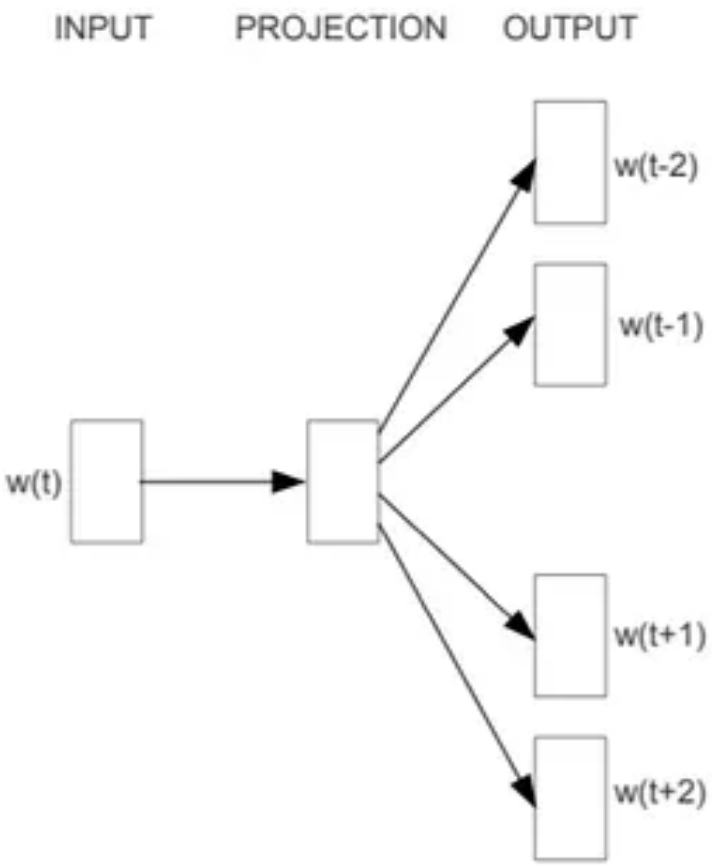
Word2vec - Skipgram

- Skipgram, on the other hand, works in the opposite way compared to CBOW. It predicts the context words given a target word.
- For each word in a sentence, Skip-gram tries to predict the context words within a fixed window size.
- It's particularly useful for capturing semantic relationships between words and tends to perform better with smaller datasets or rare words.
- Using the same example as before, Skip-gram would predict "The", "cat", "sat", and "on" based on the target word "mat".

Word2vec – Comparison CBOW vs Skipgram



CBOW



Skip-gram

CBOW vs Skipgram

- https://kavita-ganesan.com/comparison-between-cbow-skipgram-subword/


GENSIM
topic modelling for humans

What is Gensim?

- Gensim is a free open-source Python library for representing documents as semantic vectors, as efficiently (computer-wise) and painlessly (human-wise) as possible.

- Gensim is designed to process raw, unstructured digital texts ("plain text") using unsupervised machine learning algorithms.

- https://radimrehurek.com/gensim/intro.html

Word2vec in GenSim

- GenSim has (among other things) a Python implementation of Word2Vec which allow to load models and manipulate.
- Intro
  - http://radimrehurek.com/gensim/models/word2vec.html
- Examples of use
  - https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html
- Tutorial
  - http://radimrehurek.com/2014/02/word2vec-tutorial/
- Tutorial
  - https://www.machinelearningplus.com/nlp/gensim-tutorial/
- Tutorial
  - https://www.geeksforgeeks.org/nlp-gensim-tutorial-complete-guide-for-beginners/

GenSim Tutorials in general

- https://radimrehurek.com/gensim/auto_examples/index.html

Core Tutorials: New Users Start Here!

If you're new to gensim, we recommend going through all core tutorials in order. Understanding this functionality is vital for using gensim effectively.

Core Concepts

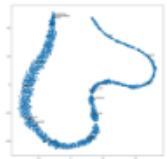
Corpora and Vector Spaces

Topics and Transformations

Similarity Queries

Tutorials: Learning Oriented Lessons

Learning-oriented lessons that introduce a particular gensim feature, e.g. a model (Word2Vec, FastText) or technique (similarity queries or text summarization).

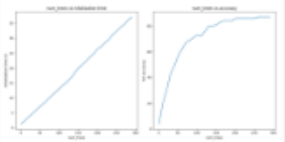


Word2Vec Model

Doc2Vec Model

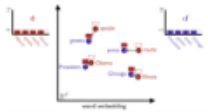
FastText Model

Ensemble LDA

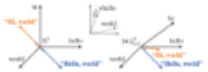


Fast Similarity Queries with Annoy and Word2Vec

LDA Model



Word Mover's Distance



Soft Cosine Measure

Word2vec – Loading pre-trained models

- Gensim comes with several already pre-trained models, in the Gensim-data repository.
- We can import the downloader from the gensim library.
- We can print the list of pre-trained models trained on large datasets available to us.
- This also includes models like GloVe and fastext other than word2vec.
- We can also download a model from our HD

Word2vec - GenSim

Load a model from your disk

```
from gensim.models import Word2Vec
model = Word2Vec.load(path/to/your/model)
```

Load a model from the library provided

```
import gensim.downloader as api

print(api.info()['models'].keys())          #list avail. models
model_twi = api.load("glove-twitter-25")  #load model
model_twi = api.load("word2vec-google-news-300")  #load larger model
```

Word2vec - GenSim

Compute similarity

```
model.similarity('france', 'spain’)
```

Instead of handmade

```
cosine_similarity = numpy.dot(model['spain'],
model['france'])/(numpy.linalg.norm(model['spain'])*
numpy.linalg.norm(model['france']))
```

Word2vec - GenSim

Most similar words

```
model.most_similar("cat")
```

Most similar of two

```
print(model.most_similar(positive=['car', 'minivan'], topn=5))
```

Which of the below does not belong in the sequence?

```
print(wv.doesnt_match(['fire', 'water', 'land', 'sea', 'air', 'car']))
```

Retrieve vocabulary

```
for index, word in enumerate(model.index_to_key):
    if index == 10:
        break
    print(f"word #{index}/{len(model.index_to_key)} is {word}")
```

Word2vec – Saving models

- We can save our existing models and load them again.

```
[ ]  w2v.save("word2vec.model")
```

```
[ ]  model = Word2Vec.load("word2vec.model")
     model.train([["hello", "world"]], total_examples=1, epochs=1)
```

Word2vec - Visualization

- Word2Vec word embedding can usually be of sizes 100 or 300, and it is practically not possible to visualise a 300 or 100 dimensional space with meaningful outputs.
- Two or three dimensional representations are more practical
- However, not two ramdom dimensions are useful
- It is better to reduce dimensionality to 2 or 3 that are relevant
- Principal component analysis (PCA) is a linear dimensionality reduction technique with applications in exploratory data analysis, visualization and data preprocessing.
- The data is linearly transformed onto a new coordinate system such that the directions (principal components) capturing the largest variation in the data can be easily identified.
- The idea is to use PCA to reduce the dimensionality and represent the word through their vectors on a 2-dimensional o 3-dim plane.

Word2vec - Visualization

- Example

- https://web.stanford.edu/class/cs224n/materials/Gensim%20word%20vector%20visualization.html

- Sentiment Analisys
- https://github.com/malinowakrew/text_classifier_sentiment/blob/master/text_classifier.ipynb

Word2vec – Example code

- https://colab.research.google.com/drive/1YkSrvfWR_EBFFrhV5E15Z6k5es4Kluom?usp=sharing

Word2Vec Materials

- TUTORIAL about how Word2Vec works.
- https://medium.com/@zafaralibagh6/a-simple-word2vec-tutorial-61e64e38a6a1

- Gensim Word2Vec Tutorial: An End-to-End Example
- https://kavita-ganesan.com/gensim-word2vec-tutorial-starter-code/

Word2Vec

Advantages

1. Capable of capturing relationships between different words including their syntactic & semantic relationships
2. The size of the embedding vector is small & flexible, unlike all the previous algorithms discussed where the size of embedding is proportional to vocabulary size
3. Since its unsupervised, human effort in tagging the data is less

Disadvantages

1. Word2Vec cannot handle out-of-vocabulary words well. It assigns a random vector representation for OOV words which can be suboptimal
2. It relies on local information of language words. The semantic representation of a word relies only on its neighbours & can prove suboptimal
3. Parameters for training on new languages cannot be shared. If you want to train word2vec in a new language, you have to start from scratch
4. Requires a comparatively larger corpus for the network to converge (especially if using skip-gram)

Word2Vec – Cons & Solutions

1. Global information is not preserved
   - ➢solved by GloVe
2. Doesn’t work well for morphologically rich languages
   - ➢solved by FastText
3. Lacks broad context awareness
   - ➢solved by LSTM, BERT, GPT

Word2Vec - Laboratory Class

- Basic natural language processing techniques
- How to train a model using Word2Vec and how to use the resulting word vectors for sentiment analysis
  - https://www.kaggle.com/c/word2vec-nlp-tutorial/overview

- Classifying questions on Stack Overflow into 3 categories depending on their quality
  - https://github.com/shraddha-an/nlp/blob/main/word_embedding_classification.ipynb

- Classifying Movie Plots by Genre
  - https://github.com/RaRe-Technologies/movie-plots-by-genre/tree/master/ipynb_with_output

Word2Vec - Laboratory Class using Gensim

- Fake News Detection (using Word2Vec)
- https://www.youtube.com/watch?v=ZrgVlfNduj8

- Code
- https://github.com/codebasics/nlp-tutorials/blob/main/16_word_vectors_gensim_text_classification/gensim_w2v_google.ipynb


GloVe
An Introduction
NLP
Hey Siri

GloVe

- Global Vectors for word representation is another famous embedding technique used quite often in NLP.
- Proposed in 2014 by Jeffery Pennington, Richard Socher & Christopher D Manning from Stanford.
- It tries to overcome the 2nd disadvantage of word2vec mentioned before by trying to learn both local & global statistics of a word to represent it.
- It tries to encompass the best of:
  - count-based technique (co-occurrence matrix) &
  - prediction-based technique (Word2Vec)
- … and hence is also referred to as a hybrid technique for continuous word representation

GloVe - Mathematical Foundation

- It is designed to generate word embeddings by capturing global statistical information from a corpus. The key idea is to factorize the co-occurrence matrix to learn word vectors.

Key Concepts

1.Co-occurrence Matrix:

1. This matrix X is constructed from a corpus where each entry X_{ij} represents the number of times word ***j*** appears in the context of word ***i***
2. For example, if "cat" and "pet" often appear together, $X_{cat,pet}$ pet will $_{be}$ high.

2.Objective:

1. Capture the ratio of co-occurrence probabilities between words to reflect semantic relationships.

GloVe - Co-occurrence Matrix

- The co-occurrence matrix is a core component of the GloVe model. It records how often pairs of words appear together within a specified context window.

	Word1	Word2	Word3	Word4
Word1	0	2	3	0
Word2	2	0	1	4
Word3	3	1	0	5
Word4	0	4	5	0

- Each cell (i, j) in the matrix represents the number of times **Word i** appears in the context of **Word j** within a given window size.

The GloVe Loss Function

- **Weighted Least Squares Objective:**

$$J = \sum_{i,j=1}^{V} f(X_{ij}) \cdot (w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log(X_{ij}))^2$$

- **Components:**
  - $w_i, \tilde{w}_j$: Word vectors being learned.
  - $b_i, \tilde{b}_j$: Bias terms associated with each word.
  - $\log(X_{ij})$: Logarithm of the co-occurrence count between words i and j.
- **Weighting Function $f(X_{ij})$:**
  - Controls the influence of each word pair.
  - Gives more weight to frequent co-occurrences while preventing extremely high counts from dominating the optimization.
- **Intuition:**
  - The goal is to minimize the difference between the dot product of word vectors and the logarithm of their co-occurrence.
  - Captures both semantic relationships and the relative importance of words in the corpus.
- **Advantages:**
  - Effectively handles both frequent and infrequent co-occurrences.
  - Enables learning rich and meaningful vector representations.

Training GloVe

- **Co-occurrence Matrix:** Construct a matrix capturing how often words appear together in a corpus.
- **Objective:** Learn word vectors by minimizing the difference between their dot products and the logarithm of their co-occurrence counts.
- **Weighting Function:** Balances the influence of word pairs, giving appropriate importance to frequent and infrequent pairs.
- **Optimization:** Use an algorithm like stochastic gradient descent to adjust word vectors until the model converges.
- **Result:** Word embeddings that effectively capture word meanings and relationships

Hyperparameters in GloVe

- **Vector Dimension (d):**
  - Determines the size of the word embeddings.
  - Higher dimensions capture more semantic nuances but require more computation.
- **Context Window Size:**
  - Defines the range of words considered for co-occurrence.
  - Larger windows capture broader context but may introduce noise.
- **Learning Rate:**
  - Controls the step size during optimization.
  - Needs careful tuning to ensure convergence without overshooting.
- **Number of Iterations:**
  - Specifies how many times the model processes the data.
  - More iterations can improve accuracy but increase training time.
- **Weighting Function Parameters:**
  - Adjusts how co-occurrence frequencies influence training.
  - Balances the contribution of frequent and rare pairs.

Advantages of GloVe over Word2Vec

- **Global context:** Uses the entire corpus, capturing broader semantic relationships.
- **Efficiency:** Can be trained faster than Word2Vec for large datasets.
  - because it uses a precomputed co-occurrence matrix, leveraging the entire corpus efficiently
  - GloVe's optimization involves just solving a matrix factorization problem
  - this reduces data processing and simplifies optimization compared to Word2Vec's iterative neural network approach
- **Improved performance:** Often shows better performance on word analogy tasks.

GloVe – Advantages & Disadvantages

Advantages

- It tends to perform better than word2vec in analogy tasks
- It considers word pair to word pair relationship while constructing the vectors & hence tend to add more meaning to the vectors when compared to vectors constructed from word-word relationships
- GloVe is easier to parallelise compared to Word2Vec hence shorter training time

Disadvantages

- Because it uses a co-occurrence matrix & global information, memory cost is more in GloVe compared to word2vec
- Similar to word2vec, it does not solve the problem of polysemous words since words & vectors have a one-to-one relationship

GloVe vs. Word2Vec: A Comparison Table

Feature	**GloVe**	**Word2Vec**
Context	Global	Local (sliding window)
Training Method	Weighted Least Squares	Negative Sampling or Hierarchical Softmax
Computational Cost	Relatively high for very large corpora	Can be computationally expensive
Performance	Often superior on analogy tasks	Good performance, but can be context-dependent

Glove – Example 1

```
import gensim.downloader as api
# Lets download a 25 dimensional GloVe representation of 2 Billion tweets
# Info on this & other embeddings : <https://nlp.stanford.edu/projects/glove/>
twitter_glove = api.load("glove-twitter-25")
# Example 1: Finding Similar Words
twitter_glove.most_similar("modi",topn=10)
# Example 1b: # To get the 25D vectors
twitter_glove['modi']
# Example 2: Word Analogy
result = glove_vectors.most_similar(positive=['woman', 'king'], negative=['man'],
topn=1)
print("\nResult of the analogy 'king - man + woman':")
print(result)
# Example 3: Similarity Between Words
twitter_glove.similarity("modi", "india")
# Example 4: Does Not Match
odd_one_out = glove_vectors.doesnt_match(["breakfast", "lunch", "dinner", "car"])
print(f"\nWord that doesn't match: {odd_one_out}")
```

OUTPUT

```
# twitter_glove.most_similar("modi",topn=10)
[('kejriwal', 0.9501368999481201),
 ('bjp', 0.9385530948638916),
 ('arvind', 0.9274109601974487),
 ('narendra', 0.9249324798583984),
 ('nawaz', 0.9142388105392456),
 ('pmln', 0.9120966792106628),
 ('rahul', 0.9069461226463318),
 ('congress', 0.904523491859436),
 ('zardari', 0.8963413238525391),
 ('gujarat', 0.8910366892814636)]
# twitter_glove['modi']
array([-0.56174 ,  0.69419 ,  0.16733 ,  0.055867, -0.26266 , -0.6303  ,
       -0.28311 , -0.88244 ,  0.57317 , -0.82376 ,  0.46728 ,  0.48607 ,
       -2.1942  , -0.41972 ,  0.31795 , -0.70063 ,  0.060693,  0.45279 ,
        0.6564  ,  0.20738 ,  0.84496 , -0.087537, -0.38856 , -0.97028 ,
       -0.40427 ], dtype=float32)
# twitter_glove.similarity("modi", "india")
0.73462856
# twitter_glove.similarity("modi", "India")
KeyError: "word 'India' not in vocabulary"
```

Glove Twiiter 25

- Pre-trained glove vectors based on 2B tweets, 27B tokens, 1.2M vocab, uncased.
- Read more:
- https://nlp.stanford.edu/projects/glove/
- https://nlp.stanford.edu/pubs/glove.pdf

- https://huggingface.co/Gensim/glove-twitter-25

Glove – Example 2

```
import gensim.downloader as api
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
import numpy as np

# Load the pre-trained GloVe model
glove_vectors = api.load("glove-wiki-gigaword-50")

# Sample data: sentences and their labels
sentences = [
    "I love programming in Python",
    "Python is great for data science",
    "I enjoy watching movies",
    "Movies are a great way to relax",
]
labels = [1, 1, 0, 0]  # 1: tech-related, 0: non-tech

# Function to compute sentence vectors by averaging word vectors
def sentence_vector(sentence, model):
    words = sentence.lower().split()
    word_vectors = [model[word] for word in words if word in model]
    return np.mean(word_vectors, axis=0)

# Convert sentences to vectors
sentence_vectors = [sentence_vector(sentence, glove_vectors) for sentence in sentences]

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(sentence_vectors, labels, test_size=0.25, random_state=42)

# Train a simple classifier
classifier = SVC(kernel='linear')
classifier.fit(X_train, y_train)

# Predict and evaluate
predictions = classifier.predict(X_test)
accuracy = accuracy_score(y_test, predictions)

print(f"Accuracy: {accuracy * 100:.2f}%")
```

Glove - Example 2

Explanation

- **Data Preparation:** Sentences are labeled based on whether they are tech-related.
- **Vectorization:** Each sentence is converted into a vector by averaging its word vectors.
- **Classification:** A Support Vector Machine (SVM) is trained on these vectors to classify sentences.
- **Evaluation:** The model's accuracy is tested on unseen data.


fastText

What is fastText?

- FastText is an open-source, free, lightweight library that allows users to learn text representations and text classifiers.
- It works on standard, generic hardware. Models can later be reduced in size to even fit on mobile devices.
- https://fasttext.cc/

fastText resources

- English word vectors
  - Trained from Wikepedia
  - Trained from Common Crawl
  - https://fasttext.cc/docs/en/english-vectors.html
- Word vectors for 157 languages
  - https://fasttext.cc/docs/en/crawl-vectors.html
- Language identification
  - https://fasttext.cc/docs/en/language-identification.html

FastText vs Word2Vec

- Word2Vec treats each word as a single unit
- FastText breaks words into subword units (n-grams), allowing it to:
  - Handle out-of-vocabulary words
  - Better understand morphologically rich languagesLearn representations for rare words
- Word2Vec learns representations for complete words only
- FastText learns representations for both words and subwords (character n-grams)
- Word2Vec: Faster training, smaller memory footprint
- FastText: Slower training, larger memory requirements due to subword storage
- Word2Vec: Better for languages with simple morphology (like English)
- FastText: Better for languages with complex morphology (Finnish, Turkish, German)

fastText usage example

- fastText Tutorial & Train Custom Word Vectors in fastText https://www.youtube.com/watch?v=Br-Ozg9D4mc

- Code
- https://github.com/codebasics/nlp-tutorials/blob/main/16_word_vectors_gensim_text_classification/gensim_w2v_google.ipynb

ADDITIONAL LINKS

Projects & Basic Applications

- Mini projects
- https://github.com/Storiesbyharshit/Natural-Language-Processing

- Wine reviews (Mini Projects)
- https://github.com/Storiesbyharshit/Natural-Language-Processing/blob/master/NLP-Wine-Reviews/Wine%20reviews.ipynb

- Sentiment Analysis with LSTM
- https://medium.com/@skillcate/sentiment-classification-using-neural-networks-a-complete-guide-1798aaf357cd

NLP - Projects

- Mini projects (Sentiment Analysis with BERT)
- https://github.com/Storiesbyharshit/Natural-Language-Processing/blob/master/Sentiment-Analysis-with-BERT-Transformers/Sentiment%20Analysis%20with%20BERT.ipynb

- NLP basic apps
- https://github.com/YanSte/NLP-LLM-Basic-Applications

- News Classification using GenSim
- https://www.youtube.com/watch?v=ZrgVlfNduj8&list=PLeo1K3hjS3uuvuAXhYjV2lMEShq2UYSwX

NLP – Tutorial Example Code

- https://github.com/codebasics/nlp-tutorials/tree/main/9_bag_of_words

NLP Course HuggingFace

- https://huggingface.co/learn/nlp-course/chapter1/1