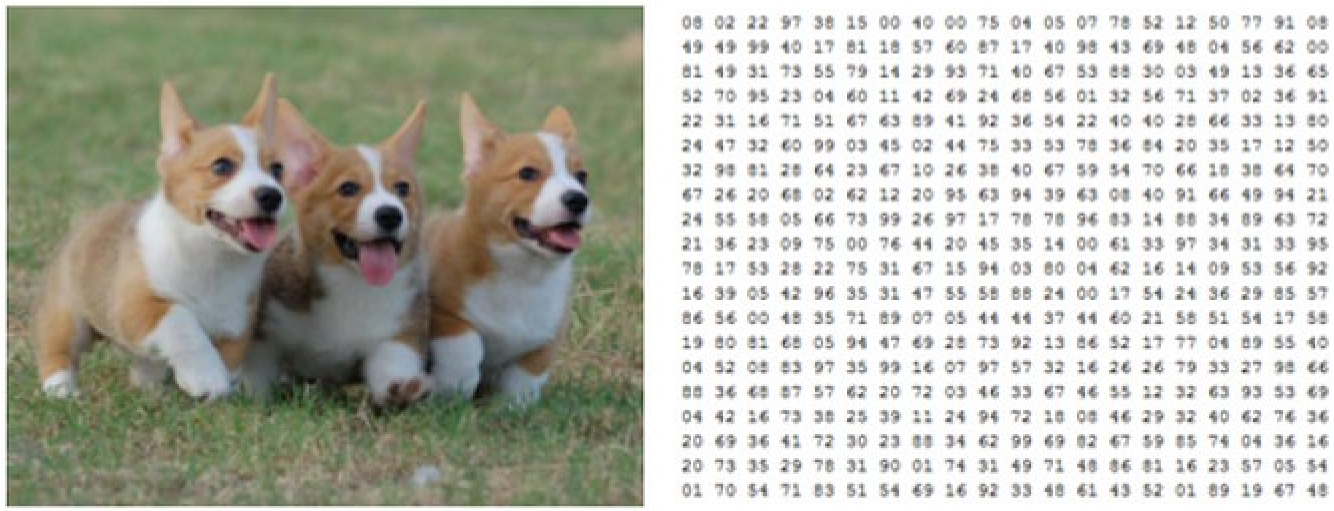
Feature Engineering (Vectorization)

PREPARED BY: AHMAD ALAA ALDINE

PRESENTED BY: AHMAD ALAA ALDINE



What We See

What Computers See

Definition

Feature engineering in Natural Language Processing (NLP) involves **transforming** raw text data into a **numerical** format that can be effectively utilized by **machine learning** algorithms.

I. Document to vector (Document Embedding)
   a. Bag of Words – Count Vectorization
   b. Bag of Words – TF-IDF
II. Word to vector (Word Embedding)
   a. Distributional Word Representation
   b. Word2vec

Bag of Words

Bag of Words (1/3)

The bag-of-words model is a way of representing text data when modeling text with machine learning algorithms.

The bag-of-words model is simple to understand and implement and has seen great success in problems such as language modeling, document classification, sentiment analysis, and others.

A bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things:

1. A vocabulary of known words
2. A measure of the presence of known words

Bag of Words (2/3)

BoW Steps:

1. Collect Dataset
    - Set of documents
2. Desing the vocabulary
    - Set of unique words in the dataset (ignoring stopwords and punctuations)
3. Create document vectors
    - Turn each document into a vector that we can use in machine learning

Bag of Words (3/3)

Limitations:

1. Semantic meaning
   - It does not consider the semantic meaning of a word
   - It ignores the context in which the word is used
2. Sparse Vectorization
   - Large feature space with many zeros
   - Higher computational time

Bag of Words – Count Vectorization

Example:

Dataset

D1	The dog barked in the park on another dog.
D2	The owner of the dog put him on the leash since he barked.
D3	My dog is barking and chasing its tail.

Vocabulary

{'chasing', 'leash', 'barked', 'tail', 'barking', 'park', 'owner', 'dog'}

Count Vectorization

	chasing	leash	barked	tail	barking	park	owner	dog
D1	0	0	1	0	0	1	0	2
D2	0	1	1	0	0	0	1	1
D3	1	0	0	1	1	0	0	1

Bag of Words – TF-IDF (1/3)

TF-IDF stands for **Term Frequency — Inverse Document Frequency**

If a particular word appears multiple times in a document, then it might have higher importance than the other words that appear fewer times (TF).

At the same time, if a particular word appears many times in a document, but it is also present many times in some other documents, then maybe that word is frequent, so we cannot assign much importance to it. (IDF).

TF-IDF (2/3)

Formula:

$$TF - IDF = TF \ * IDF$$

$$TF = \frac{Frequency\ of\ the\ word\ in\ the\ document}{Total\ number\ of\ words\ in\ the\ document}$$

$$IDF = \log\left(\frac{Total\ number\ of\ documents}{Number\ of\ documents\ containing\ the\ word}\right)$$

TF-IDF(3/3)

Example:

Count Vectorization

	chasing	leash	barked	tail	barking	park	owner	dog
D1	0	0	1	0	0	1	0	2
D2	0	1	1	0	0	0	1	1
D3	1	0	0	1	1	0	0	1

TF

	chasing	leash	barked	tail	barking	park	owner	dog
D1	0	0	¼=0.25	0	0	¼=0.25	0	2/4=0.5
D2	0	¼=0.25	¼=0.25	0	0	0	¼=0.25	¼=0.25
D3	¼=0.25	0	0	¼=0.25	¼=0.25	0	0	¼=0.25

IDF

	chasing	leash	barked	tail	barking	park	owner	dog
IDF	Log(3/1)= 0.477	Log(3/1)= 0.477	Log(3/2)= 0.176	Log(3/1)= 0.477	Log(3/1)= 0.477	Log(3/1)= 0.477	Log(3/1)= 0.477	Log(3/3)=0

TF-IDF

	chasing	leash	barked	tail	barking	park	owner	dog
D1	0	0	0.25*0.176	0	0	0.25*0.477	0	0
D2	0	0.25*0.477	0.25*0.176	0	0	0	0.25*0.477	0
D3	0.25*0.477	0	0	0.25*0.477	0.25*0.477	0	0	0

Distributional Word Representation

Distributional Word Representation (1/2)

Represent words as **vectors** based on the distribution of their **context** words.

Words sharing the **same context** tend to have a similar **meaning**.

Steps:

1. Define what are the contexts of a word in the dataset
   - Window (surrounding words)
   - All words in a sentence

2. Count how many times each vocabulary word occurs in these contexts

3. Build vectors

Distributional Word Representation (2/3)

Example:

Dataset

D1	The dog barked in the park on another dog.
D2	The owner of the dog put him on the leash since he barked.
D3	My dog is barking and chasing its tail.

Vocabulary

{'chasing', 'leash', 'barked', 'tail', 'barking', 'park', 'owner', 'dog'}

Vectors (Window = 5)

	chasing	leash	barked	tail	barking	park	owner	dog
chasing	0	0	0	1	1	0	0	0
leash	0	0	0	0	0	0	0	0
barked	0	0	0	0	0	0	0	1
tail	1	0	0	0	0	0	0	0
barking	1	0	0	0	0	0	0	1
park	0	0	0	0	0	0	0	0
owner	0	0	0	0	0	0	0	0
dog	0	0	0	0	1	0	0	0

Distributional Word Representation (3/3)

Example:

Dataset

D1	The dog barked in the park on another dog.
D2	The owner of the dog put him on the leash since he barked.
D3	My dog is barking and chasing its tail.

Vocabulary

{'chasing', 'leash', 'barked', 'tail', 'barking', 'park', 'owner', 'dog'}

Vectors (all sentence)

	chasing	leash	barked	tail	barking	park	owner	dog
chasing	0	0	0	1	1	0	0	1
leash	0	0	1	0	0	0	1	1
barked	0	1	0	0	0	1	1	3
tail	1	0	0	0	1	0	0	1
barking	1	0	0	1	0	0	0	1
park	0	0	1	0	0	1	0	1
owner	0	1	1	0	0	0	0	1
dog	1	1	2	1	1	0	0	0

Large dimension matrix with many zeros → Sparsity Problem

Word2Vec

Word2Vec

This approach was released back in 2013 by **Google** researchers, and it took the NLP industry by storm.

It uses the power of a simple **Neural Network** to generate word embeddings.

There are mainly two ways to implement Word2Vec:

1. Skip-Gram
2. CBOW

Word2Vec – Skip-Gram (1/4)

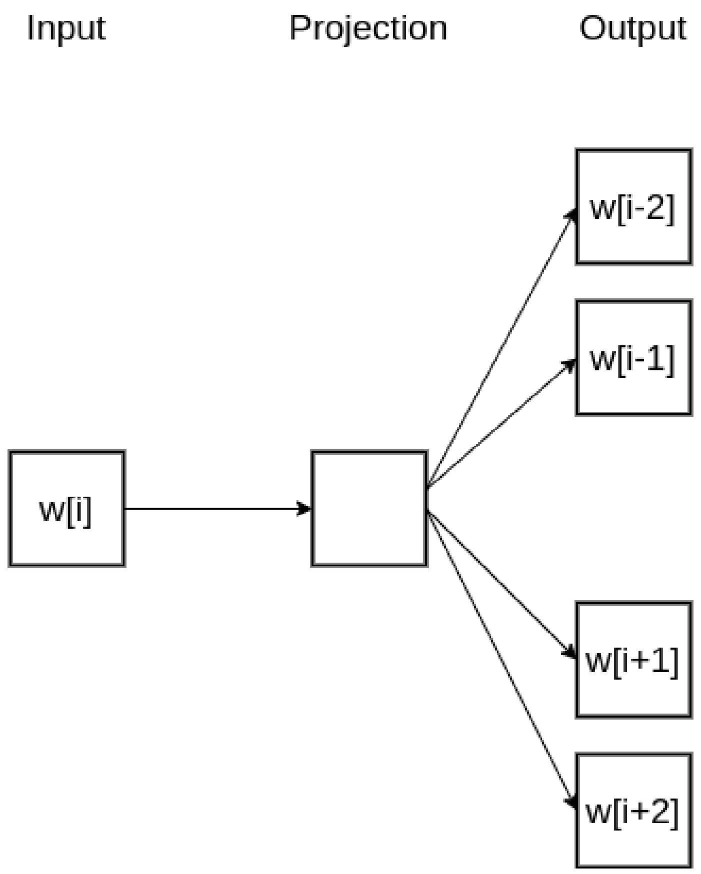
In Skip-Gram method, we provide a word to our Neural Network and ask it to predict the context.

Technically, it predicts the probabilities of a word being a context word for the given target word.

However, the interesting part is, we don't use this trained Neural Network.

Instead, the goal is just to learn the weights of the hidden layer while predicting the surrounding words correctly.

These weights are the **word embeddings**.



Word2Vec – Skip-Gram (2/4)

Input Layer:

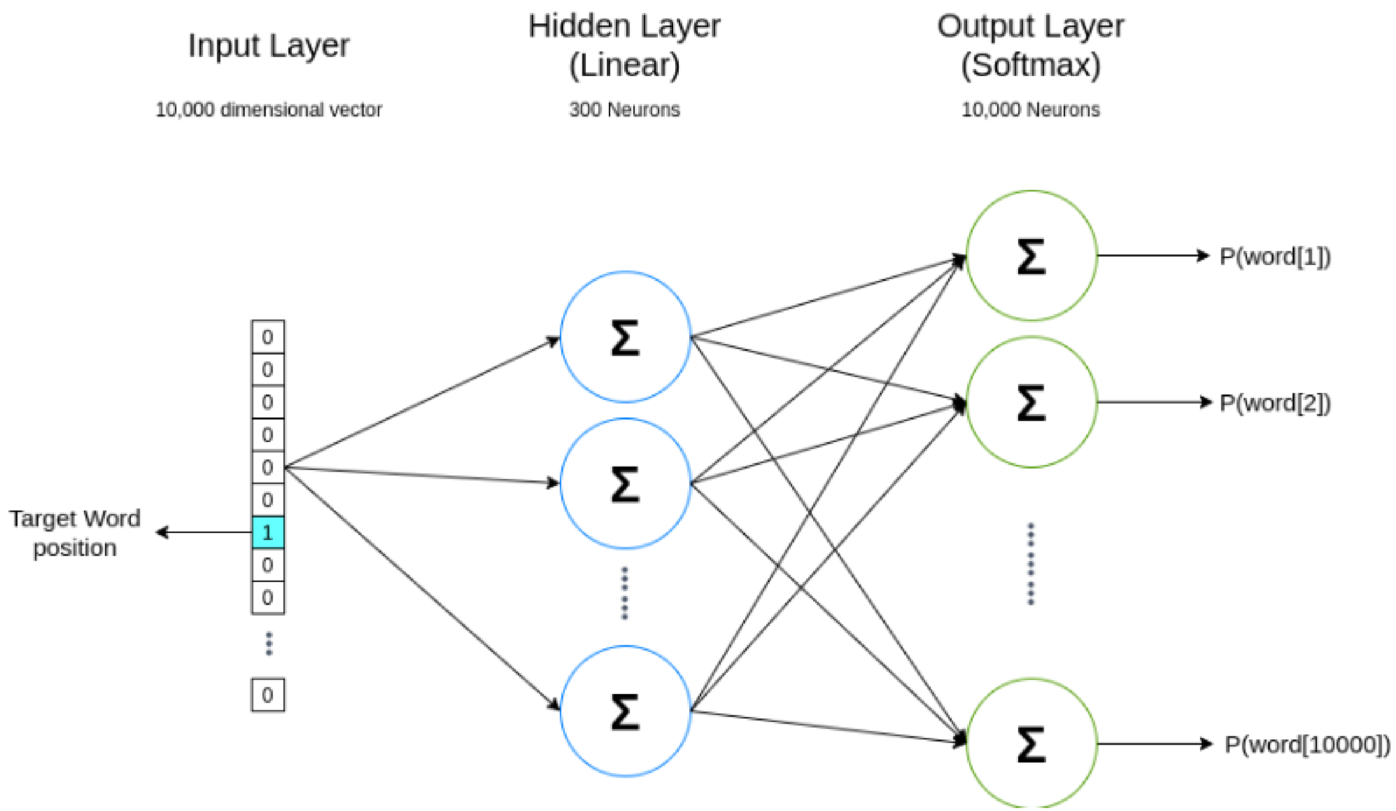
- Dataset vocabulary words

Hidden Layer:

- Hyperparameter
- tuned to obtain the best results

Output Layer:

- Dataset vocabulary words

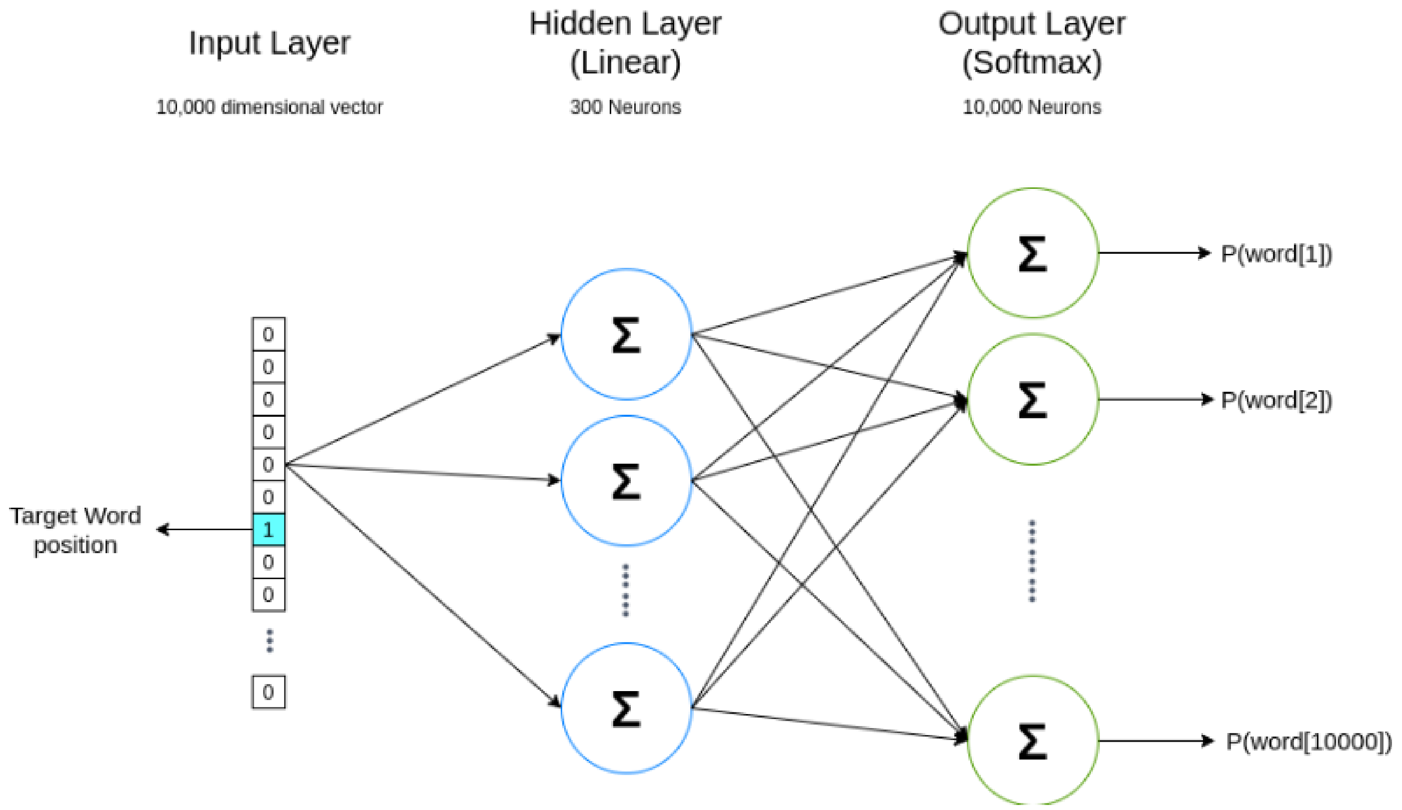


Word2Vec – Skip-Gram (3/4)

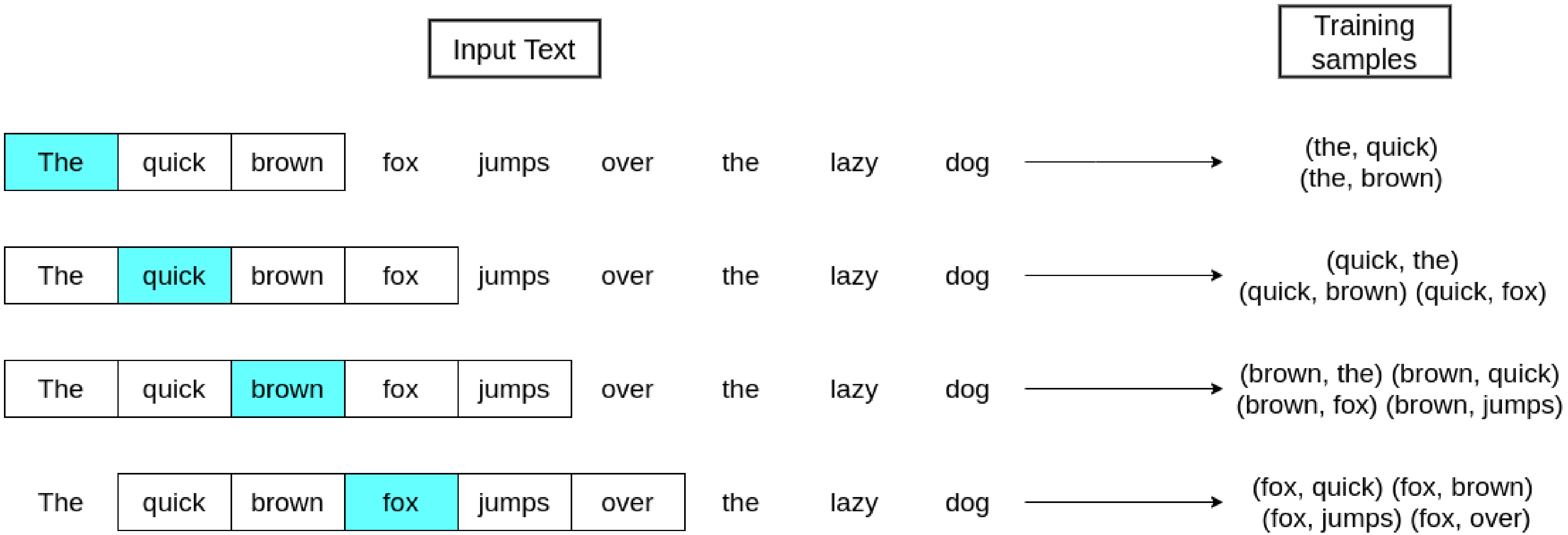
Input:
- One-Hot Encoding for a target word
- 1 in the position corresponding to the target word and 0 everywhere

Hidden Layer:
- Linear (No activation function)

Output:
- The probability of each word being the context word for our input target word
- Softmax: $S(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{N} e^{y_j}}$



Word2Vec – Skip-Gram (4/4)



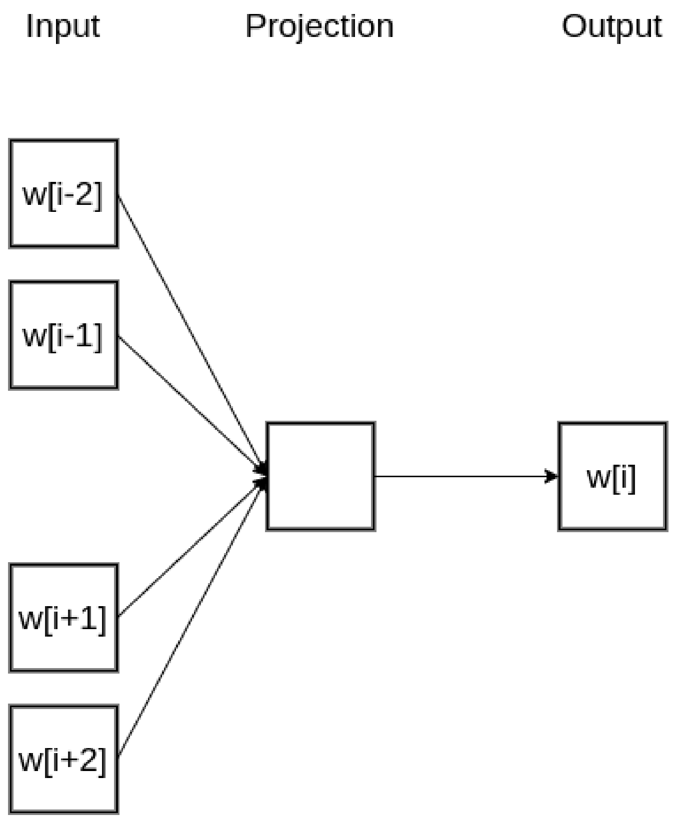
Once the model gets trained, the final word embedding for each word will be given by the following calculation: **1×10000 input vector * 10000×300 weights matrix = 1×300 vector**

Word2Vec – CBOW (1/2)

CBOW stands for Continuous Bag of Words.

Instead of predicting the context words, we input them into the model and ask the network to predict the current word.

CBOW is the mirror image of the skip-gram approach.



Word2Vec – CBOW (2/2)

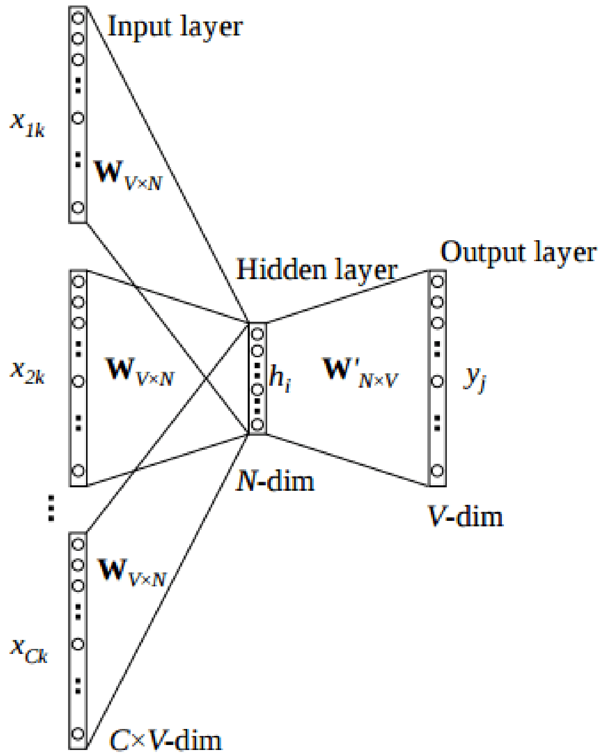
The dimension of our hidden layer and output layer stays the same as the skip-gram model.

The input is C context words in the form of a one-hot encoded vector of size 1xV (V = size of vocabulary).

C vectors will be multiplied by the Weights of our hidden layer of shape VxN (N = number of neurons in the hidden layer).

This will result in C, 1xN vectors, and all of these C vectors will be averaged element-wise to obtain our final activation for the hidden layer, which then will be fed into our output softmax layer.

The learned weight between the hidden and output layer makes up the word embedding representation.

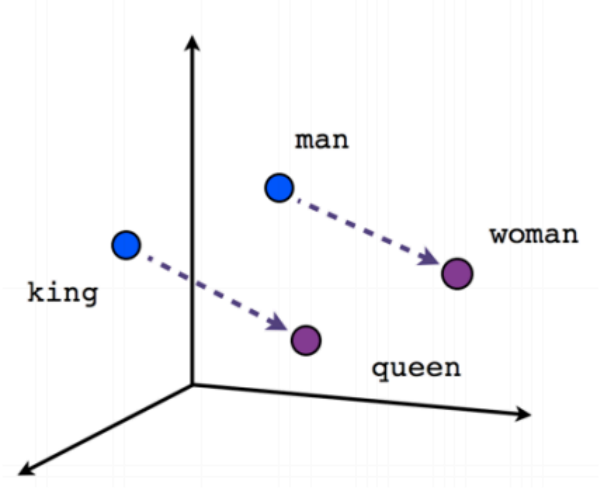


Word2Vec – Skip-Gram vs CBOW

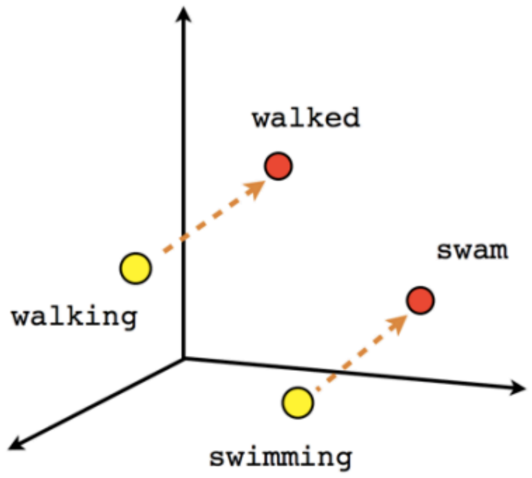
When to use the skip-gram model and when to use CBOW?

- Skip-gram works well with small datasets and can better represent rare words.
- CBOW is found to train faster than skip-gram and can better represent frequent words.
- So the choice of skip-gram VS. CBOW depends on the kind of problem that we're trying to solve.

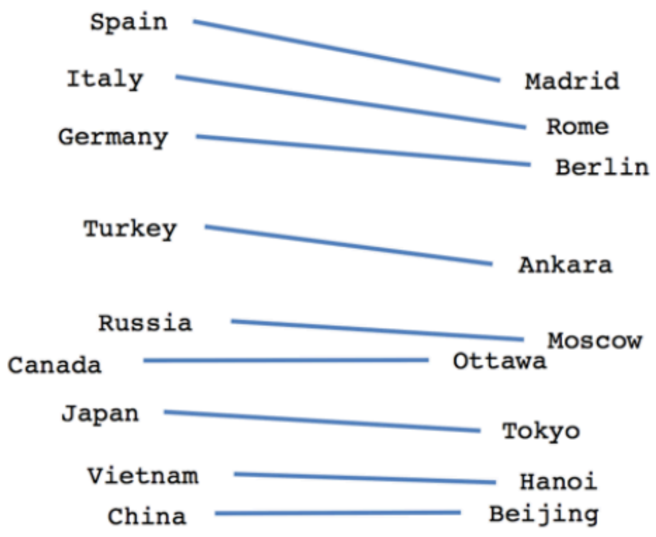
Word2Vec – Results



Male-Female

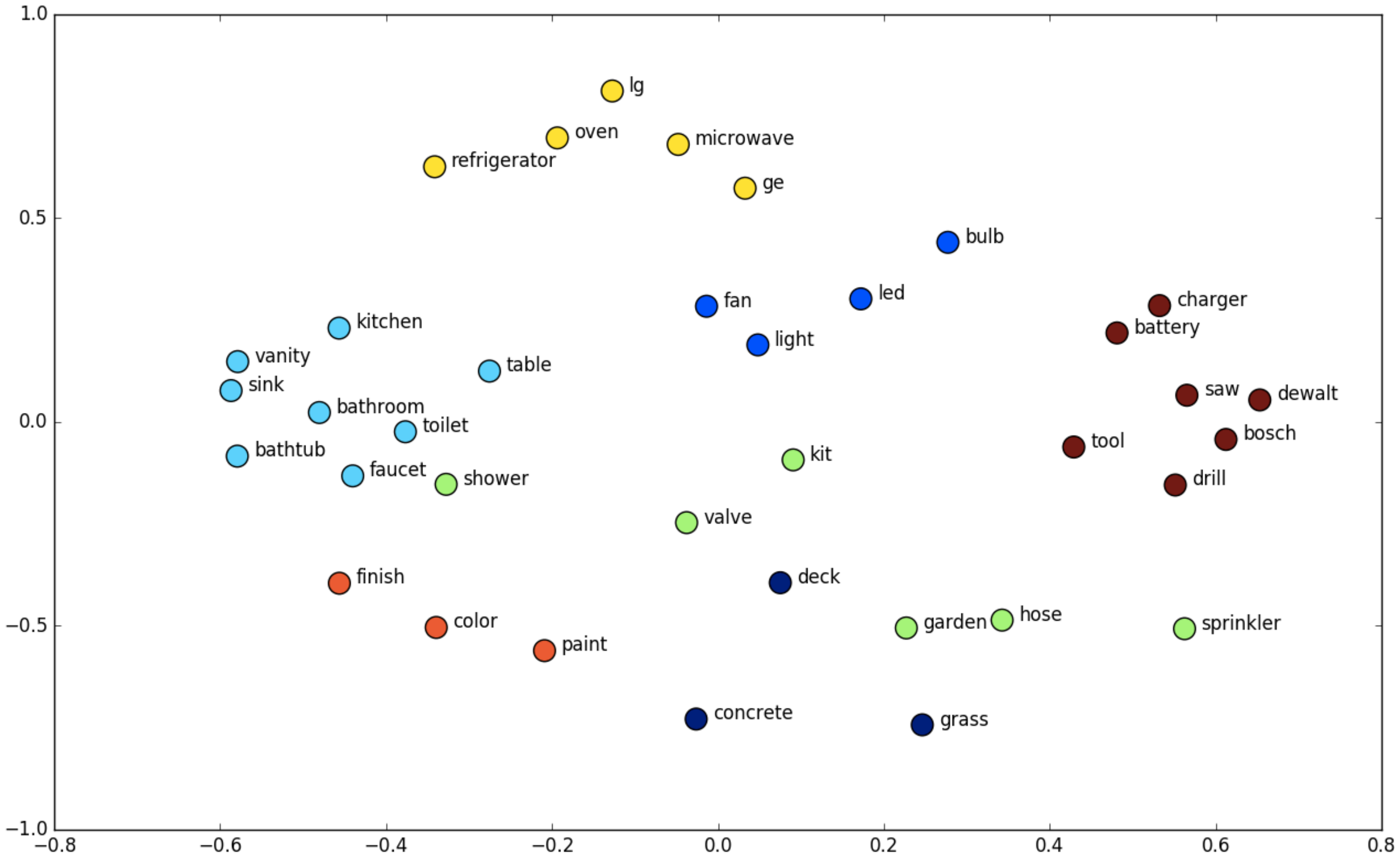


Verb tense



Country-Capital

Word2Vec – Results



Feature Engineering in Python

Count Vectorization using Sklearn

```
from sklearn.feature_extraction.text import CountVectorizer

sents = ['The dog barked in the park on another dog',
   'The owner of the dog put him on the leash since he barked.',
   'My dog is barking and chasing its tail.']

cv = CountVectorizer()

X = cv.fit_transform(sents)
X = X.toarray()

print(sorted(cv.vocabulary_.keys()))
print(X)
```

```
['and', 'another', 'barked', 'barking', 'chasing', 'dog', 'he', 'him', 'in', 'is', 'its', 'leash', 'my', 'of', 'on', 'owner',
'park', 'put', 'since', 'tail', 'the']
[[0 1 1 0 0 2 0 0 1 0 0 0 0 0 1 0 1 0 0 0 2]
 [0 0 1 0 0 1 1 1 0 0 0 1 0 1 1 1 0 1 1 0 3]
 [1 0 0 1 1 1 0 0 0 1 1 0 1 0 0 0 0 0 0 1 0]]
```

Count Vectorization without Stopwords using Sklearn

```
from sklearn.feature_extraction.text import CountVectorizer

sents = ['The dog barked in the park on another dog',
   'The owner of the dog put him on the leash since he barked.',
   'My dog is barking and chasing its tail.']

cv = CountVectorizer(stop_words='english')

X = cv.fit_transform(sents)
X = X.toarray()

print(sorted(cv.vocabulary_.keys()))
print(X)
```

```
['barked', 'barking', 'chasing', 'dog', 'leash', 'owner', 'park', 'tail']
[[1 0 0 2 0 0 1 0]
 [1 0 0 1 1 1 0 0]
 [0 1 1 1 0 0 0 1]]
```

N-Gram Count Vectorization without Stopwords using Sklearn ¶

```
from sklearn.feature_extraction.text import CountVectorizer

sents = ['The dog barked in the park on another dog',
   'The owner of the dog put him on the leash since he barked.',
   'My dog is barking and chasing its tail.']

cv = CountVectorizer(ngram_range=(1,2), stop_words='english')

X = cv.fit_transform(sents)
X = X.toarray()

print(sorted(cv.vocabulary_.keys()))
print(X)
```

```
['barked', 'barked park', 'barking', 'barking chasing', 'chasing', 'chasing tail', 'dog', 'dog barked', 'dog barking', 'dog lea
sh', 'leash', 'leash barked', 'owner', 'owner dog', 'park', 'park dog', 'tail']
[[1 1 0 0 0 0 2 1 0 0 0 0 0 0 1 1 0]
 [1 0 0 0 0 0 1 0 0 1 1 1 1 1 0 0 0]
 [0 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 1]]
```

TF-IDF Vectorization using Sklearn

```
from sklearn.feature_extraction.text import TfidfVectorizer

sents = ['The dog barked in the park on another dog',
   'The owner of the dog put him on the leash since he barked.',
   'My dog is barking and chasing its tail.']

cv = TfidfVectorizer(stop_words='english')

X = cv.fit_transform(sents)

print(X.toarray())

import pandas as pd
df = pd.DataFrame(X[0].T.todense(), index=cv.get_feature_names(), columns=["TF-IDF"])
df = df.sort_values('TF-IDF', ascending=False)

print()
print(df)
```

```
[[0.44102652 0.         0.         0.68499287 0.         0.
  0.57989687 0.        ]
 [0.44451431 0.         0.         0.34520502 0.5844829  0.5844829
  0.         0.        ]
 [0.         0.54645401 0.54645401 0.32274454 0.         0.
  0.         0.54645401]]

           TF-IDF
dog      0.684993
park     0.579897
barked   0.441027
barking  0.000000
chasing  0.000000
leash    0.000000
owner    0.000000
tail     0.000000
```

Word2Vec: using Pre-trained model in Gensim

```
from gensim import models
w2v = models.KeyedVectors.load_word2vec_format(
'./GoogleNews-vectors-negative300.bin', binary=True)

vect = w2v['healthy']
w2v.most_similar('happy')
```

Word2Vec: Train our model in Gensim

```
from gensim import models

sents = ['The dog barked in the park on another dog',
   'The owner of the dog put him on the leash since he barked.',
   'My dog is barking and chasing its tail.']

# Word2vec requires the training dataset in form of a list of lists
# of tokenized sentences, so we'll preprocess and convert sents to

sents = [sent.split() for sent in sents]
custom_model = models.Word2Vec(sents, min_count=1,vector_size=300,workers=4)

vect = custom_model.wv['dog']
print(vect[:5])
result = custom_model.wv.most_similar('barked')
print(result)
```

```
[-1.7874241e-04  7.8810059e-05  1.7011166e-03  3.0030911e-03
 -3.1009833e-03]
[('and', 0.11018942296504974), ('leash', 0.09745844453573227), ('him', 0.08689301460981369), ('barking', 0.07596393674612045),
('he', 0.07124919444322586), ('put', 0.06887509673833847), ('owner', 0.053101178258657455), ('its', 0.03030511364340782), ('do
g', 0.026307500898838043), ('is', 0.023843316361308098)]
```