Feature Engineering (Vectorization)

PREPARED BY: AHMAD ALAA ALDINE

PRESENTED BY: AHMAD ALAA ALDINE

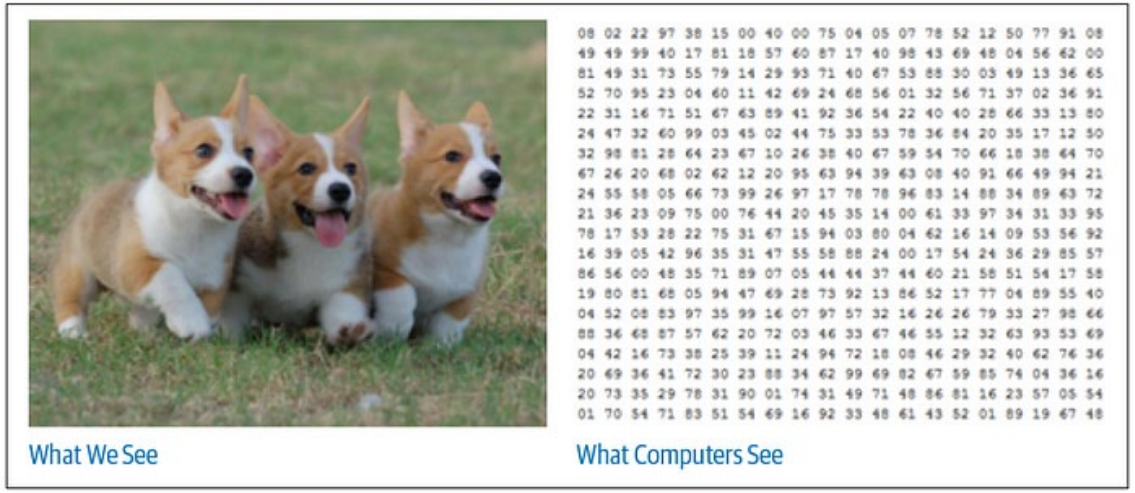
Why Vectorize?

Unstructured Data

Natural Language is inherently unstructured. Machines cannot directly perform mathematical operations on raw strings or characters without a mapped numerical representation.

Feature Engineering

Vectorization transforms text into a "Feature Space" where algorithms can apply linear algebra for classification, clustering, and information retrieval.



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.

Vectorization Techniques

FREQUENCY-BASED TECHNIQUES

These methods are simple but often result in high-dimensional, sparse vectors that ignore word order.

They are lexical techniques that fails to capture semantic relationships.

- **One-hot encoding** → word to vector
- **Bag-of-words** → document to vector
- **Bag-of-N-grams** → document to vector
- **TF-IDF** → document to vector
- **Distributional Word Representation** → word to vector

EMBEDDING-BASED TECHNIQUES

These techniques generate dense, low-dimensional vectors where similar words have similar numerical representations.

They are semantic techniques so they capture semantic relationships.

- **Word2Vec** → word embedding
- **GloVe** → word embedding
- **FastText** → word embedding
- **BERT** → contextual embedding

Frequency-Based Technique

ONE-HOT ENCODING, BAG OF WORDS , BAG OF NGRAMS, TF-IDF, DISTRIBUTION WORD REPRESENTATION

Frequency-Based Technique Pipeline



1. Tokenization

Breaking sentences into individual words or N-grams.



2. Vocabulary

Collecting all unique tokens across the entire corpus.

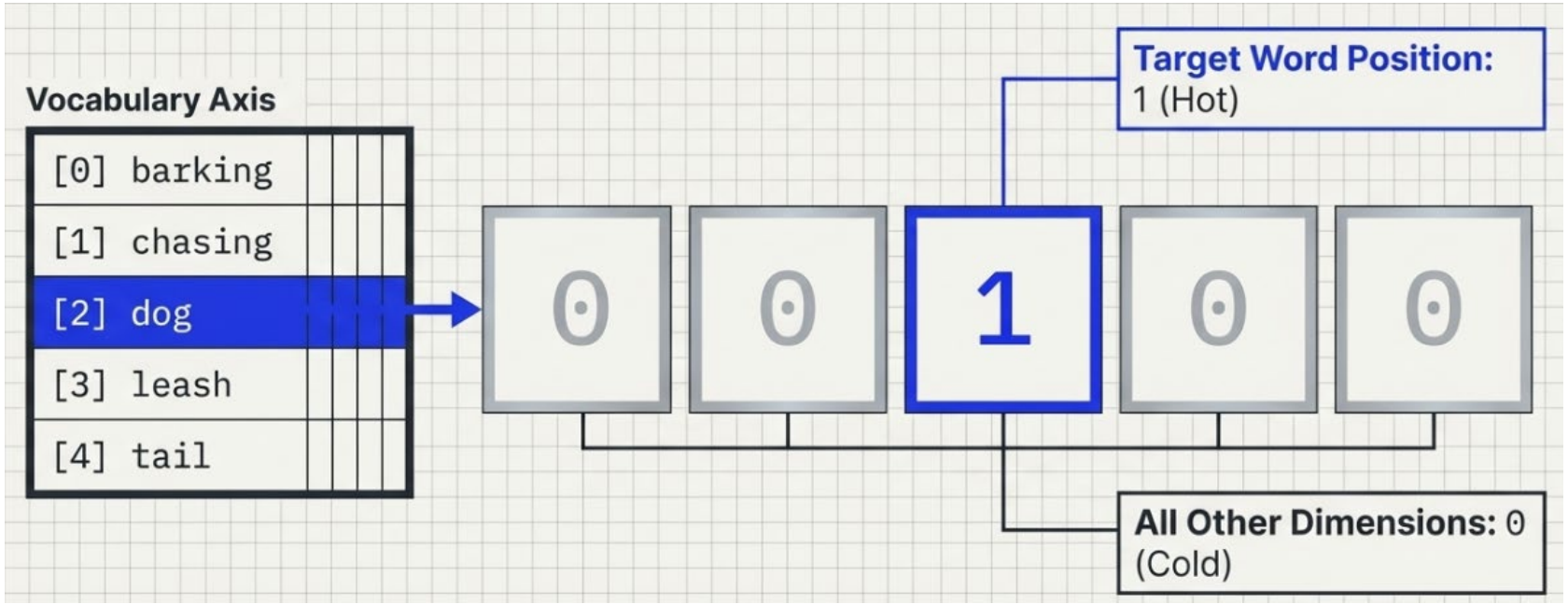


3. Encoding

Represent each word or document as a row vector

One Hot Encoding

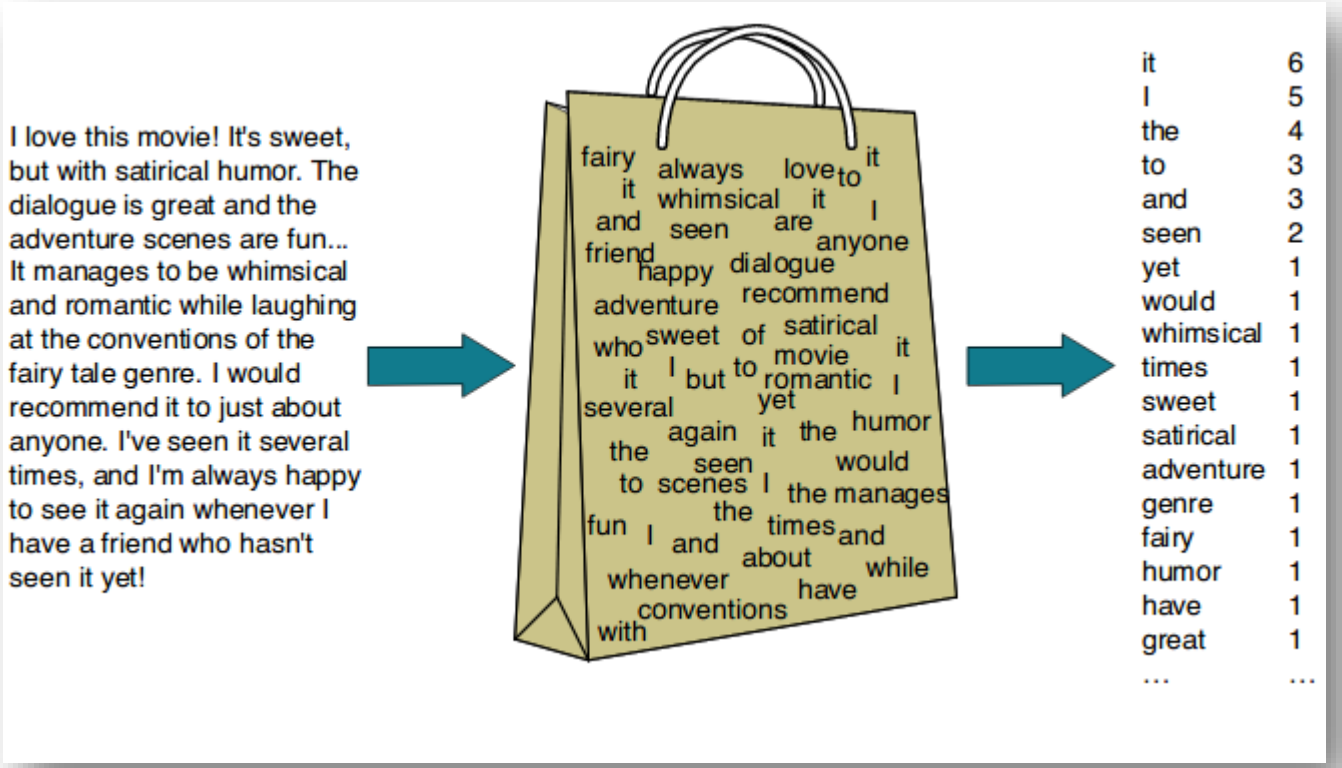
A word is represented entirely by its position in the vocabulary in a binary vector (vector of many 0s and one 1).
The dimension size matches the total vocabulary size.



Bag-of-Words (BoW)

The simplest method to represent text as numbers. It treats each document as a "bag" of its constituent tokens, discarding structure and sequence.

- Disregards Grammar & Word Order
- Focuses purely on Frequency
- Produces High-Dimensional Vectors



Bag-of-Words (BoW) – Example

- The Corpus

ID	Original Text
Doc 1	"The cat sat on the mat."
Doc 2	"The dog barked at the cat."
Doc 3	"The mat was sat on by the cat."

- Step 1: **Tokenization**
  - Split docuements into individual word units (tokens) based on whitespace.
  - Document 1: ["the", "cat", "sat", "on", "the", "mat"]
- Step 2: **Vocabulary**
  - We extract every unique word from across all documents to form our Feature Space.
  - [at, barked, by, cat, dog, mat, on, sat, the, was]

Bag-of-Words (BoW) – Example

Step 3: **Encoding (Vectorization)**

- Document-Term Matrix (DTM)

Doc / Vocab	at	barked	by	cat	dog	mat	on	sat	the	was
Doc 1	0	0	0	1	0	1	1	1	2	0
Doc 2	1	1	0	1	1	0	0	0	2	0
Doc 3	0	0	1	1	0	1	1	1	2	1

Why simple Bag-of-Words fails

Loss of Word Order

Standard BoW (Unigrams) treats "not good" and "good not" as identical. By discarding word order, the model loses the semantic meaning of phrases.

Local Context

N-grams preserve the immediate neighbors of a word. This allows the model to capture negation, idioms, and multi-word expressions like "Natural Language Processing".

Bag-of-n-grams: Capturing Local Syntax in Statistical NLP

Types of N-grams

1

Unigram (n=1)

Individual words.

Ex: "the", "cat", "sat".

2

Bigram (n=2)

Pairs of consecutive words.

Ex: "the cat", "cat sat".

3

Trigram (n=3)

Groups of three words.

Ex: "the cat sat".

The Context-Complexity Trade-off

- As n increases, the model captures more semantic context and linguistic structure.
- However, this leads to an exponential explosion in the vocabulary size (feature space).
- This creates extreme data sparsity, as most possible word combinations never appear in the training set.

Bigrams (n = 2) – Example

- The Corpus

ID	Original Text
Doc 1	"The cat sat on the mat."
Doc 2	"The dog barked at the cat."
Doc 3	"The mat was sat on by the cat."

- Unique Bigrams
  - While our unigram vocabulary had 10 words, our bigram vocabulary expands to .
  - "the cat", "cat sat", "sat on", "on the", "the mat“, “the dog”, “dog barked”, “barked at”, “at the”, “mat was”, “was sat”, “on by”, “by the”.
- Bigram Feature Matrix

Doc / Bigram	the cat	cat sat	sat on	on the	the mat	the dog	dog barked	barked at	at the	mat was	was sat	on by	by the
Doc 1	1	1	1	1	1	0	0	0	0	0	0	0	0
Doc 2	1	0	0	0	0	1	1	1	1	0	0	0	0
Doc 3	1	0	1	0	1	0	0	0	0	1	1	1	1

Why Word Counts are Not Enough

1. Stopword Dominance

Words like "the", "is", and "at" appear frequently in almost every document. In BoW, these words get the highest weights, even though they carry the least semantic meaning.

2. Lack of Uniqueness

BoW treats a word that appears 10 times in every document exactly the same as a word that appears 10 times in only one document. The latter is far more informative for classification.

TF-IDF: Term Frequency – Inverse Document Frequency

- Smarter Term Weighting
- Assigning importance by penalizing commonality.

TF-IDF (t, d)= $TF(t,d) * IDF(t)$
Where t is a term and d is a document

Term Frequency (TF)

Measures how frequently a term occurs in a specific document. Since documents vary in length, we normalize the raw count by the total number of words.

$$\text{TF (t, d)} = \frac{Count\ of\ t\ in\ d}{Total\ terms\ in\ d}$$

Example (Doc 1): "The cat sat on the mat."

Total Words: 6

TF("the") = 2/6 = 0.33

TF("cat") = 1/6 = 0.16

Inverse Document Frequency (IDF)

Measures how important a term is. While computing TF, all terms are considered equally important. However, certain terms, such as "is", "of", and "that", may appear a lot of times but have little importance.

$$\text{IDF (t)} = log\frac{N}{df(t)}$$

Where N is total docs

and df(t) is number of docs containing the term.

Word	df	idf
Romeo	1	1.57
salad	2	1.27
Falstaff	4	0.967
forest	12	0.489
battle	21	0.246
wit	34	0.037
fool	36	0.012
good	37	0
sweet	37	0

TF-IDF – Example

- The Corpus

ID	Original Text
Doc 1	"The cat sat on the mat."
Doc 2	"The dog barked at the cat."
Doc 3	"The mat was sat on by the cat."

- Document Frequencies

Term	Documents Present	df(t)	IDF (log10(3/df))
the / cat	Doc 1, 2, 3	3	**0.00** (Common)
mat / sat / on	Doc 1, 3	2	**0.176**
dog / barked / was / at / by	Doc 2 or 3	1	**0.477** (Rare)

Note: "the" and "cat" are penalized to zero because they appear in every document.

TF-IDF – Example

- Term Frequency

Doc \ TF	the	Cat	Sat	on	mat	dog	barked	was	at	by
Doc 1 (6 terms)	2/6	1/6	1/6	1/6	1/6	0	0	0	0	0
Doc 2 (6 terms)	2/6	1/6	0	0	0	1/6	1/6	0	1/6	0
Doc 3 (8 terms)	2/8	1/8	1/8	1/8	1/8	0	0	1/8	0	1/8

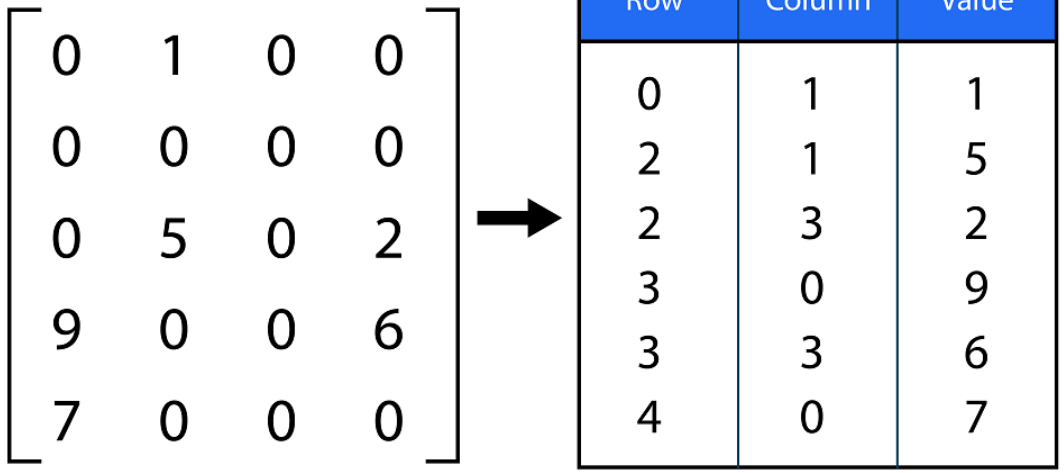
- Term Frequency – Inverse Document Frequency (Final Matrix)

Doc \ TF-IDF	the	Cat	Sat	on	mat	dog	barked	Was	at	by
Doc 1	2/6 * 0 = 0	1/6 * 0 = 0	1/6 * 0.176 = 0.03	1/6 * 0.176 = 0.03	1/6 * 0.176 = 0.03	0 * 0.477 = 0	0 * 0.477 = 0	0* 0.477 = 0	0* 0.477 = 0	0* 0.477 = 0
Doc 2	2/6 * 0 = 0	1/6 * 0 = 0	0 * 0.176 = 0	0 * 0.176=0	0 * 0.176=0	1/6 * 0.477 = 0.08	1/6 * 0.477 = 0.08	0* 0.477 = 0	1/6* 0.477 = 0.08	0* 0.477 = 0
Doc 3	2/8 * 0 = 0	1/8 * 0 = 0	1/8 * 0.176=0.02	1/8 * 0.176=0.02	1/8 * 0.176=0.02	0 * 0.477 = 0	0 * 0.477 = 0	1/8* 0.477 = 0.06	0* 0.477 = 0	1/8* 0.477 = 0.06

The Sparsity Problem

In a typical corpus, vocabularies exceed 50,000 words. However, a single document may only use 100 unique words.

This results in Sparse Matrices where large number of entries are zero.
This leads to massive memory overhead and "distance saturation" in vector space.



$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 2 \\ 9 & 0 & 0 & 6 \\ 7 & 0 & 0 & 0 \end{bmatrix}$$

Row	Column	Value
0	1	1
2	1	5
2	3	2
3	0	9
3	3	6
4	0	7

Critical Limitations

No Context

Ignores word order. "Dog bites man" and "Man bites dog" result in identical vectors.

No Semantics

Cannot detect synonyms. "Stellar" and "Star" are treated as completely unrelated dimensions.

Polysemy

Fails to distinguish meanings for the same word (e.g., "Bank" as a river edge vs. financial institution).

Distributional Word Representation

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 - 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 = 2)

	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	1	0	1	0	0	0

Distributional Word Representation – 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	3	1	1	1	1	1

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)

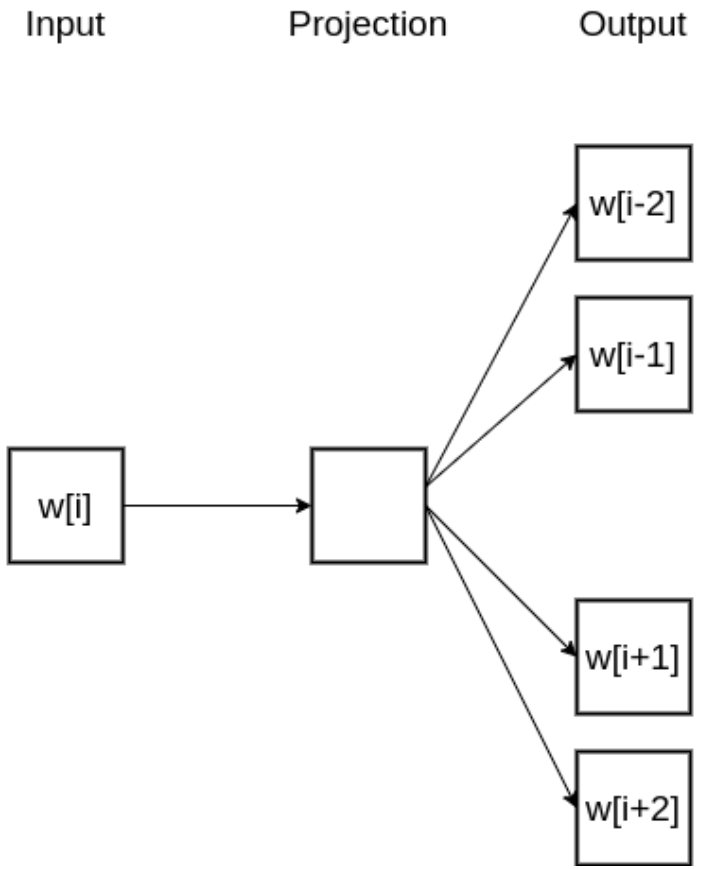
In Skip-Gram method, we provide a word to our Neural Network and ask it to predict the context.

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:

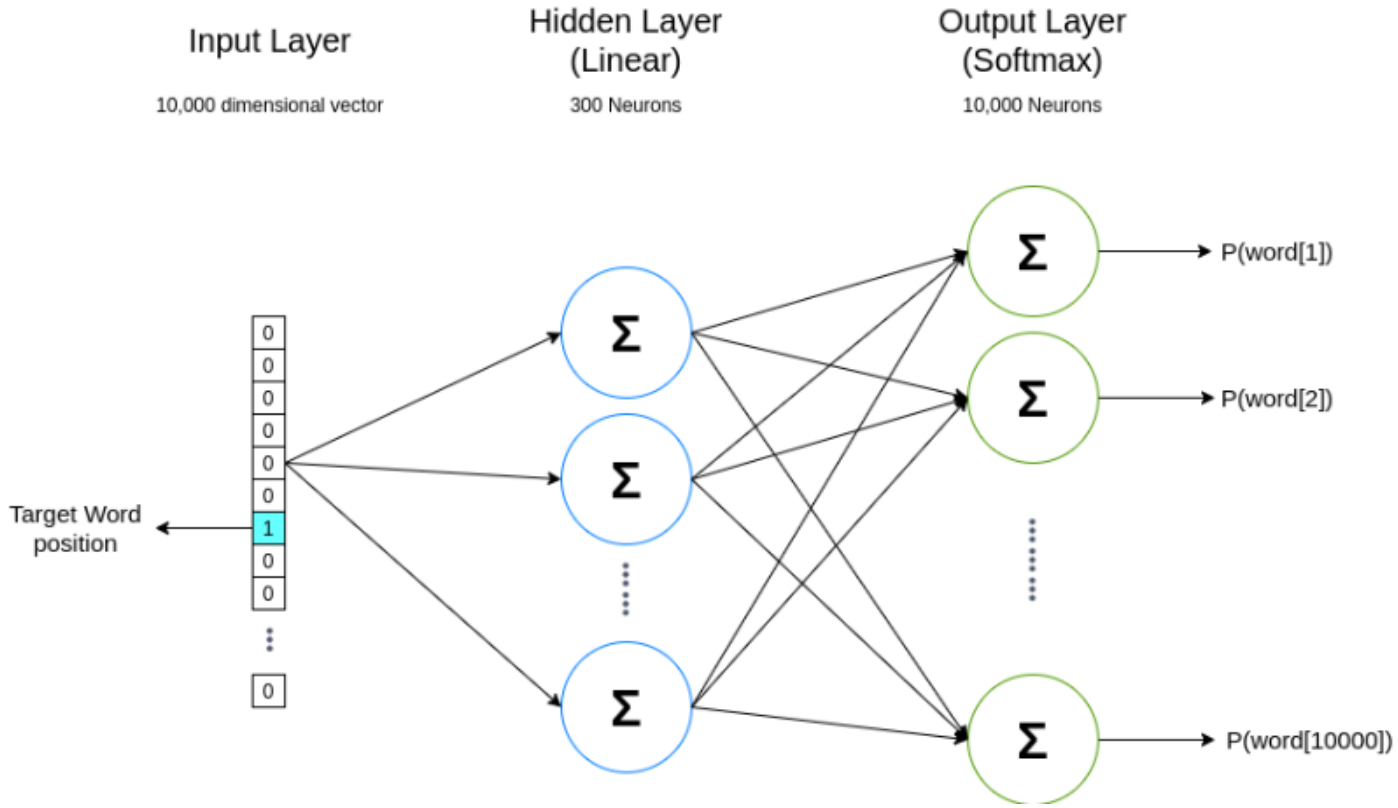
- Dataset vocabulary words

Hidden Layer:

- Hyperparameter
- tuned to obtain the best results

Output Layer:

- Dataset vocabulary words



Word2Vec – Skip-Gram (3/4)

Input:

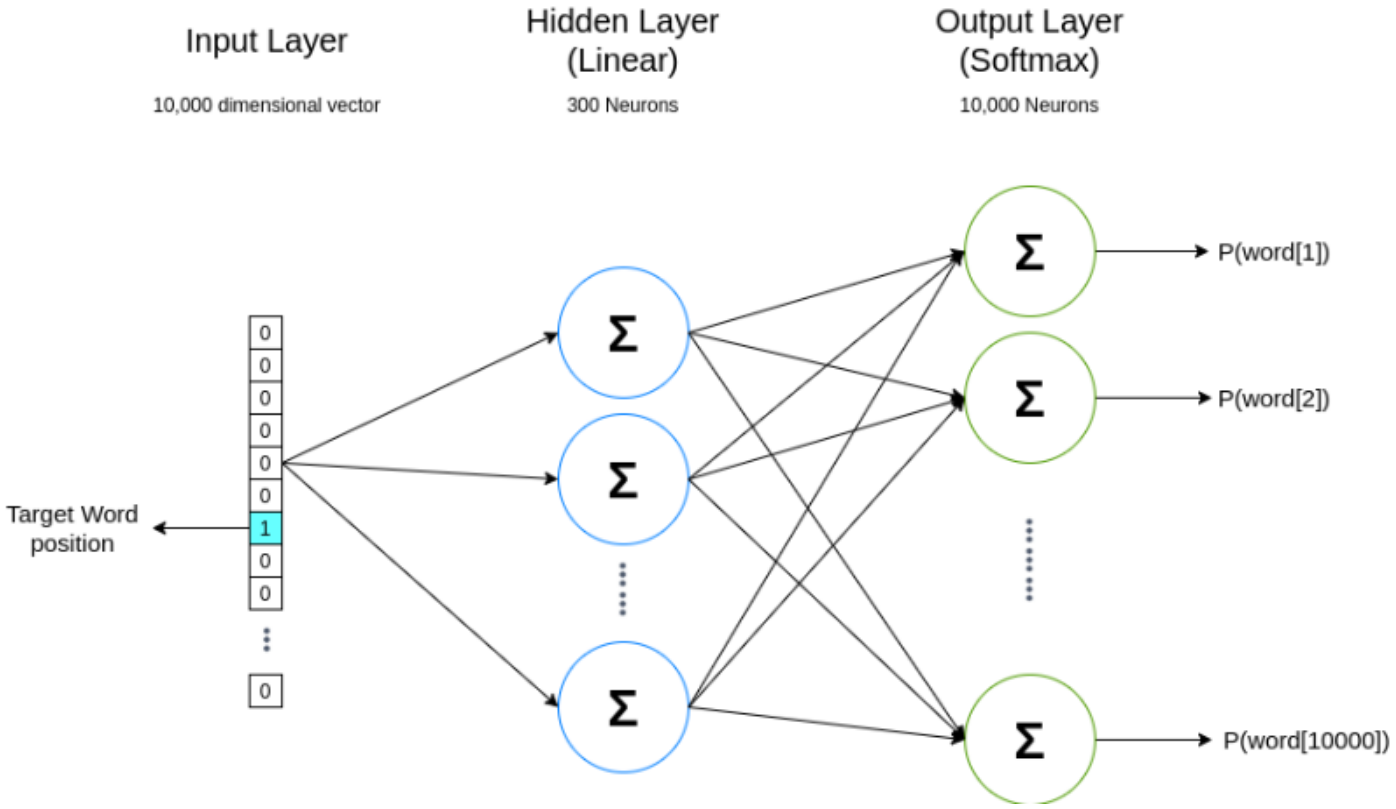
- One-Hot Encoding for a target word
- 1 in the position corresponding to the target word and 0 everywhere

Hidden Layer:

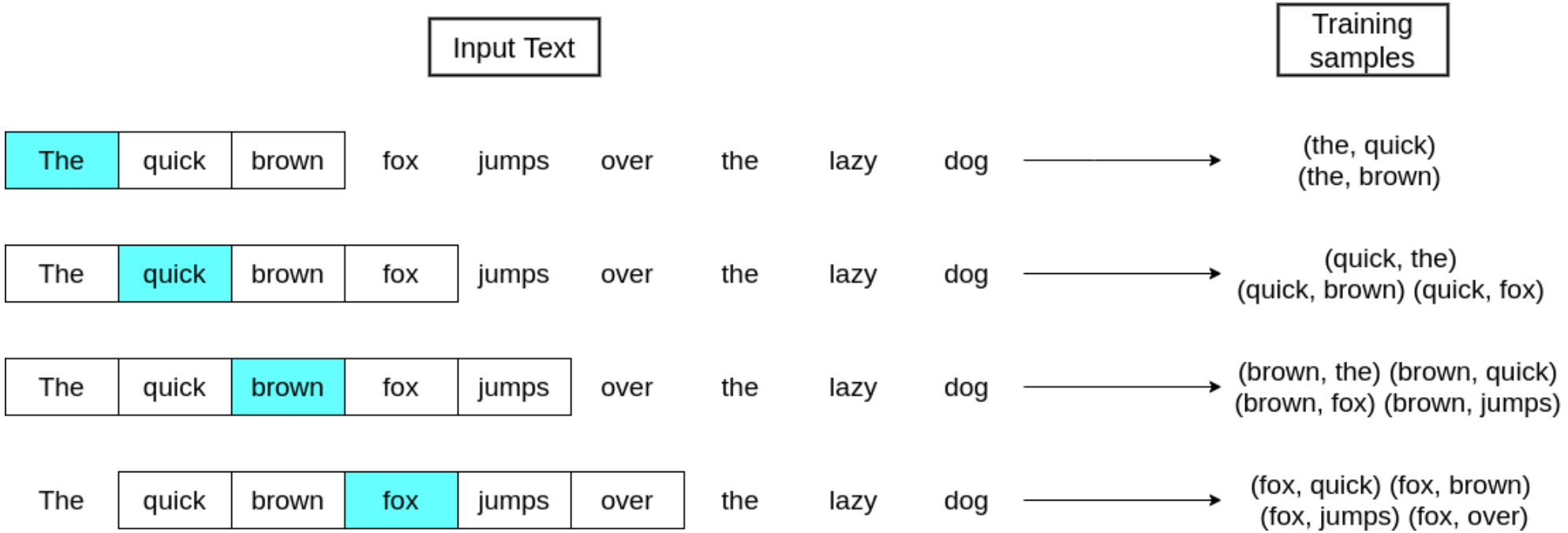
- Linear (No activation function)

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)



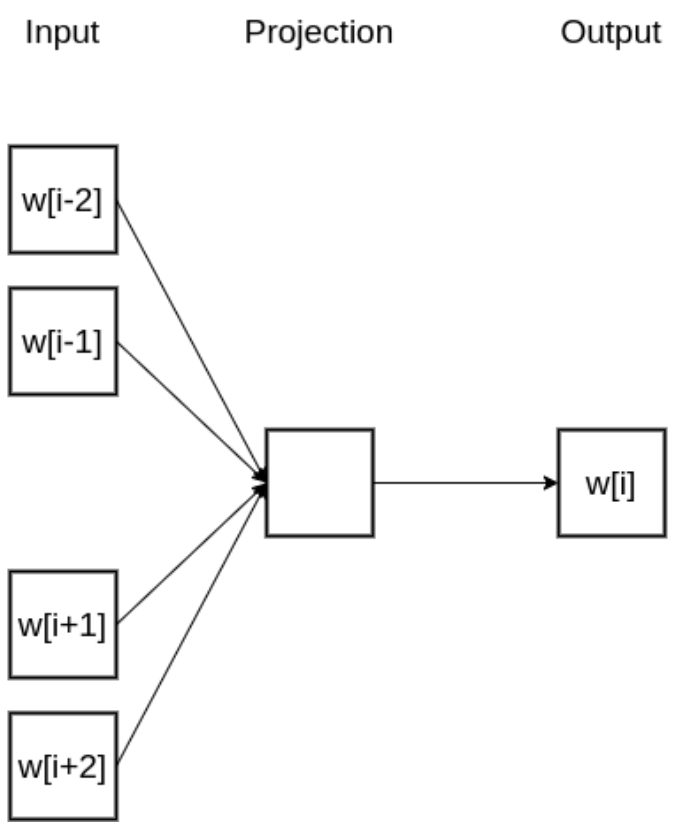
Once the model gets trained, the final word embedding for each word will be given by the following calculation: **1×10000 input vector * 10000×300 weights matrix = 1×300 vector**

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)

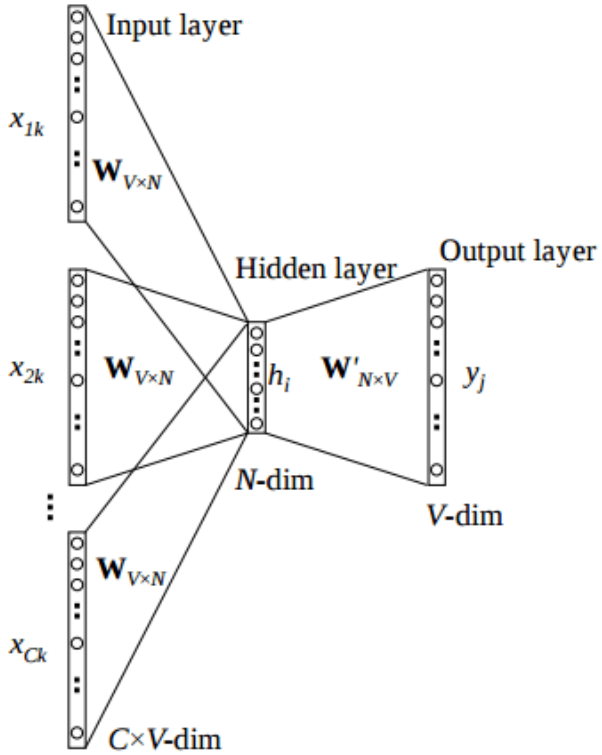
The dimension of our hidden layer and output layer stays the same as the skip-gram model.

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.

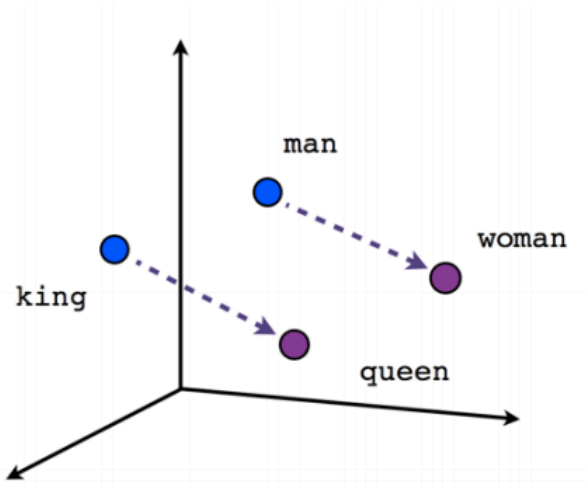


Word2Vec – Skip-Gram vs CBOW

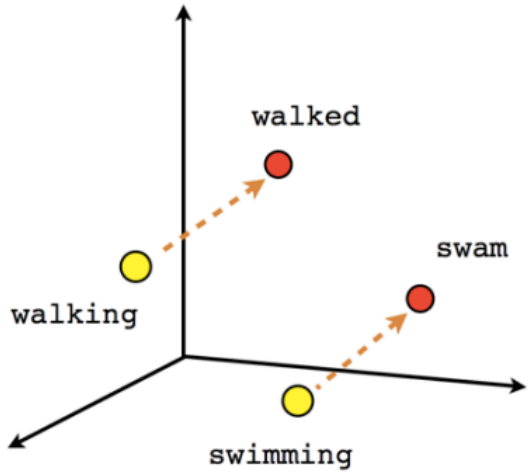
When to use the skip-gram model and when to use CBOW?

- 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.

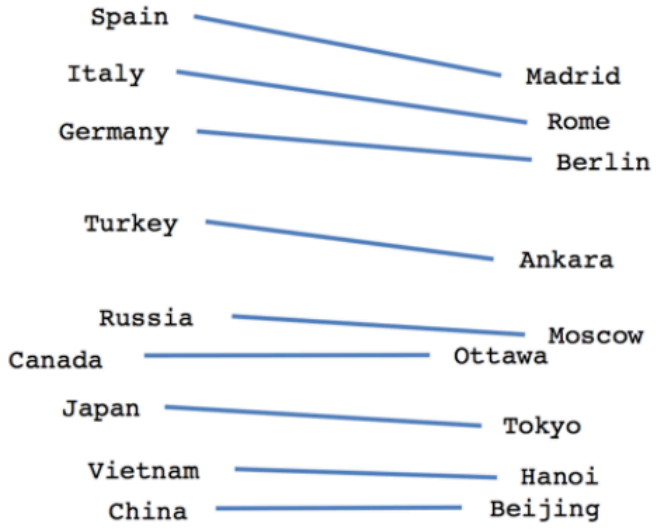
Word2Vec – Results



Male-Female

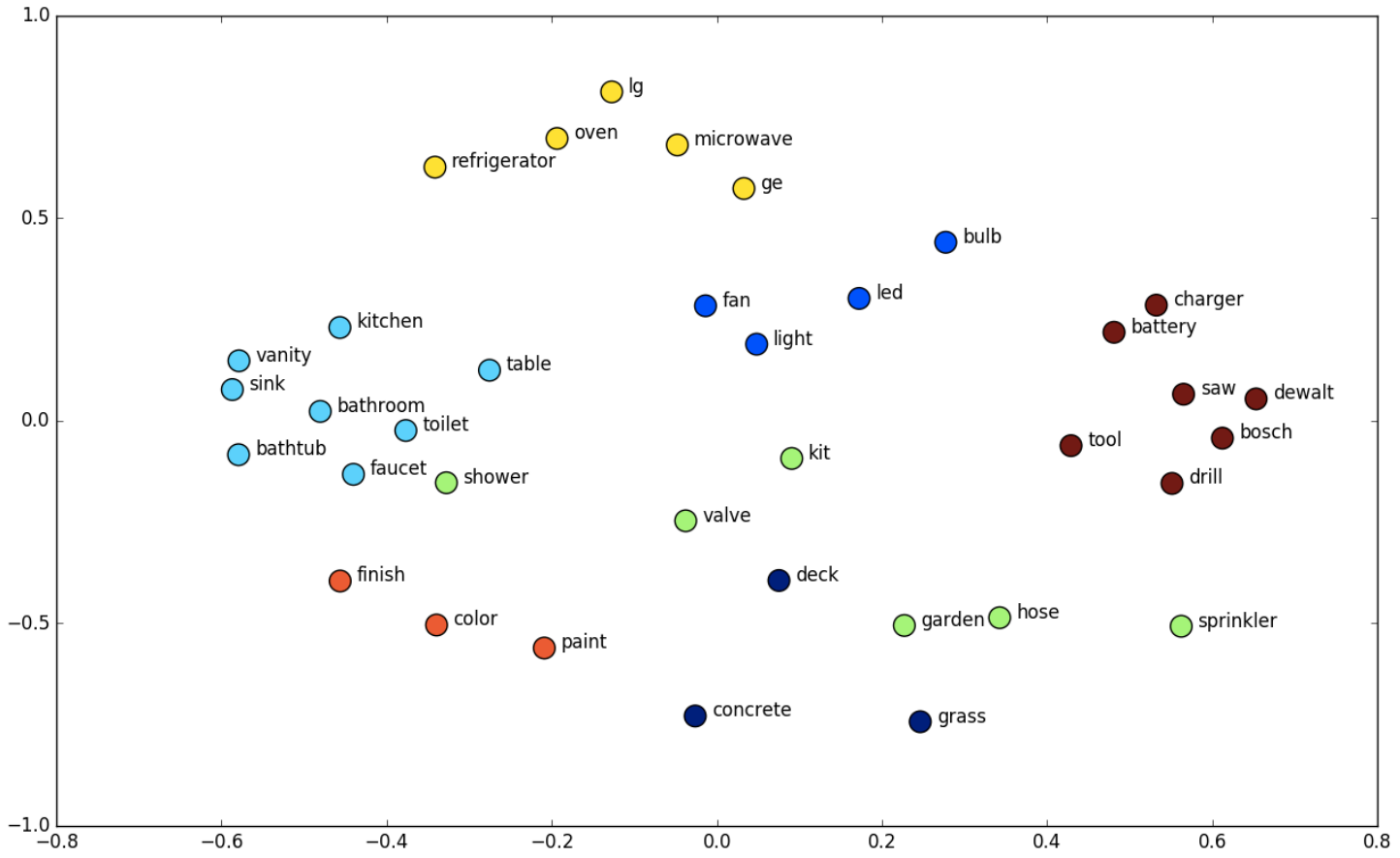


Verb tense



Country-Capital

Word2Vec – Results



FastText

The main principle behind fastText is that the morphological structure of a word carries important information about the meaning of the word.

Such structure is not taken into account by traditional word embeddings like Word2Vec.

FastText attempts to solve this by treating each word as the aggregation of its subwords.

Subwords are taken to be the n-grams character of the word.

The vector for a word is simply taken to be the sum of all vectors of its component char-ngrams.

FastText can obtain vectors even for out-of-vocabulary (OOV) words.

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)]
```

FastText: Train our model

```
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.']

sents = [sent.split() for sent in sents]

model = models.FastText(sentences=sents, vector_size=100, window=5, min_count=1, workers=4, sg=1)

word_embedding = model.wv['dog']
print(word_embedding[:5])

# Most similar words to a given word
similar_words = model.wv.most_similar('barked')

print("Most similar words to 'barked':", similar_words)
```

```
[-0.00260617  0.00086763 -0.00203372  0.00195915 -0.00396831]
Most similar words to 'barked': [('barked.', 0.6791878342628479), ('barking', 0.3493046164512634), ('its', 0.1485029011964798),
('park', 0.10157518088817596), ('is', 0.09532742202281952), ('My', 0.04805697500705719), ('dog', 0.04705512896180153), ('and',
0.04276891052722931), ('in', 0.03834148868918419), ('him', 0.02182852476835251)]
```