Applied Machine Learning for Business Analytics

Lecture 2: From BoW to Word2Vec

Lecturer: Zhao Rui

Agenda

1. Representation Learning in NLP
2. Word Embeddings
3. Neural Networks for NLP
4. Tokens and Embeddings

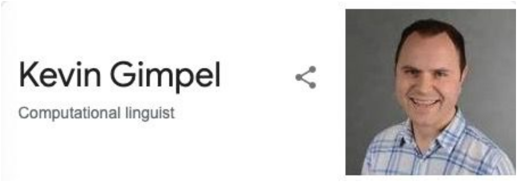
1. Representation Learning

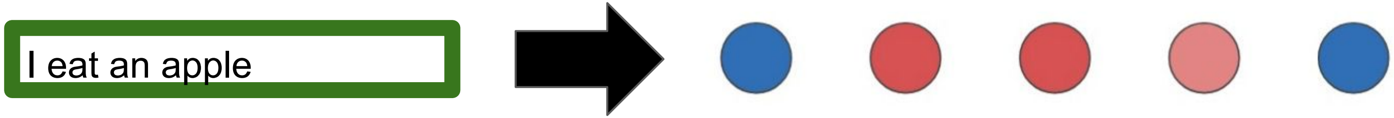
Representation learning

- We need to develop systems that read and understand text the way a person does, by forming a representation of the text, and other context information that humans create to understand a piece of text.

Representation learning

- We need to develop systems that read and understand text the way a person does, by forming a representation of the text, and other context information that humans create to understand a piece of text.





Text

Numerical Representation

The learned representation should capture high-level semantic and syntactic information.

History of NLP

- Now, neural nlp models are able to achieve state-of-arts results in all tasks.
- Before neural nlp:
  - Symbolic NLP: rule-based system (derived from linguistic)
  - Statistical NLP: data-driven and use statistical methods

Statistical NLP

- Starting from Document-Term Matrix
  - It contains the co-occurrence information
  - Bag-of-Words: n-gram as features
  - TF-IDF: frequency of words to measure importance
  - Matrix Decomposition:
    - SVD->Latent Semantic Analysis
    - Probabilistic model-> Topic Model

Corpus

D0: I eat an apple every day
D1: I eat an orange every day
D2: I like driving my car to work

Document-Term Matrix

	an	apple	car	day	driving	eat	every	like	my	orange	to	work
0	1	1	0	1	0	1	1	0	0	0	0	0
1	1	0	0	1	0	1	1	0	0	1	0	0
2	0	0	1	0	1	0	0	1	1	0	1	1

Bag-of-Words

TF-IDF

Latent Semantic Analysis

Topic Models

Bag-of-Words

Building Vocabulary: Tokenization -> Count unique set

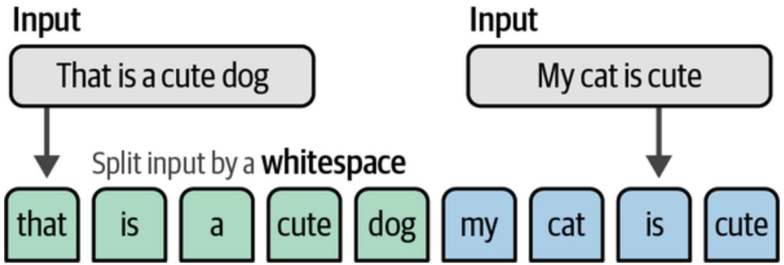


Figure 1-3. Each sentence is split into words (tokens) by splitting on a whitespace.

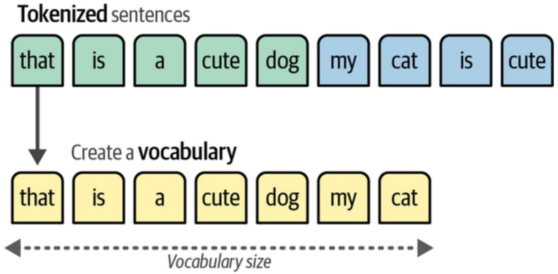


Figure 1-4. A vocabulary is created by retaining all unique words across both sentences.

Encoding Sentences into Vectors

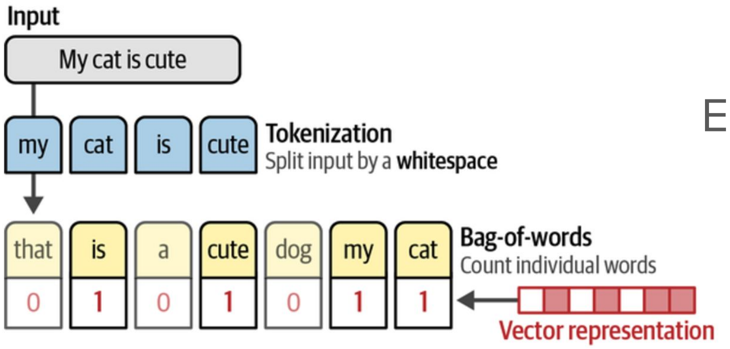


Figure 1-5. A bag-of-words is created by counting individual words. These values are referred to as vector representations.

Source: Hands-On Large Language Models

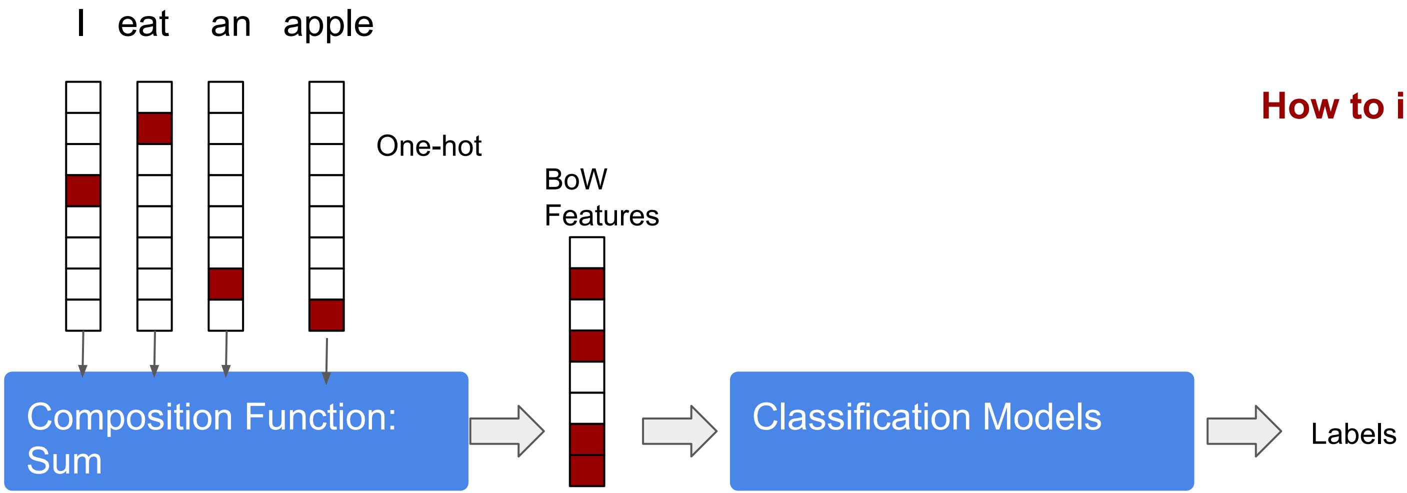
Limitations of BoW Vectors

- Too strong assumption: all words are independent of each other
    - | *orange* - *peach* | < | *orange* - *car* |
- Can not capture the order information in the sequence
- High dimensionality due to large size of vocabulary

	an	apple	car	day	driving	eat	every	like	my	orange	to	work
0	1	1	0	1	0	1	1	0	0	0	0	0
1	1	0	0	1	0	1	1	0	0	1	0	0
2	0	0	1	0	1	0	0	1	1	0	1	1

A new perspective on BoW

- Each word in vocab is represented in one-hot embedding
- Sum one-hot vectors of the words in a sentence
- The final vector is the representation for the given sentence and then fed into a classifier.



How to improve?

Statistical NLP

- D3: apple car
  - Word vector: one-hot ones
    - Apple: 0 1 0 0 0 0 0 0 0 0 0 0
    - Car: 0 0 1 0 0 0 0 0 0 0 0 0
  - Sum of two word vectors
    - apple vec + car vec
  - Document vector:
    - 0 1 1 0 0 0 0 0 0 0 0 0

Corpus

D0: I eat an apple every day
D1: I eat an orange every day
D2: I like driving my car to work

Document-Term Matrix

	an	apple	car	day	driving	eat	every	like	my	orange	to	work
0	1	1	0	1	0	1	1	0	0	0	0	0
1	1	0	0	1	0	1	1	0	0	1	0	0
2	0	0	1	0	1	0	0	1	1	0	1	1

Neural NLP

2001	Neural language models
2008	Multi-task learning
2013	Word embeddings
2013	Neural networks for NLP
2014	Sequence-to-sequence models
2015	Attention
2015	Memory-based networks
2018	Pretrained language models

https://www.kamperh.com/slides/ruder+kamper_indaba2018_talk.pdf

2. Word Embeddings

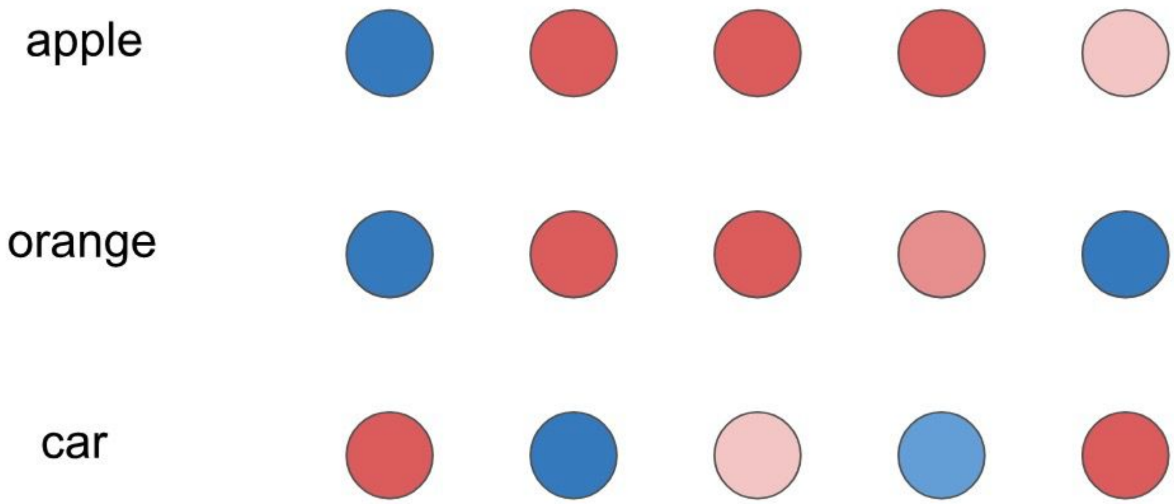
Word representation

- How to represent words in a vector space

apple	[0 0 0 0 0 **1** 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0]
orange	[0 0 0 0 0 0 0 0 0 0 0 0 0 **1** 0 0 0 0 ... 0 0 0 0 0 0]
car	[0 0 0 0 0 0 0 **1** 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0]

Distributed representation

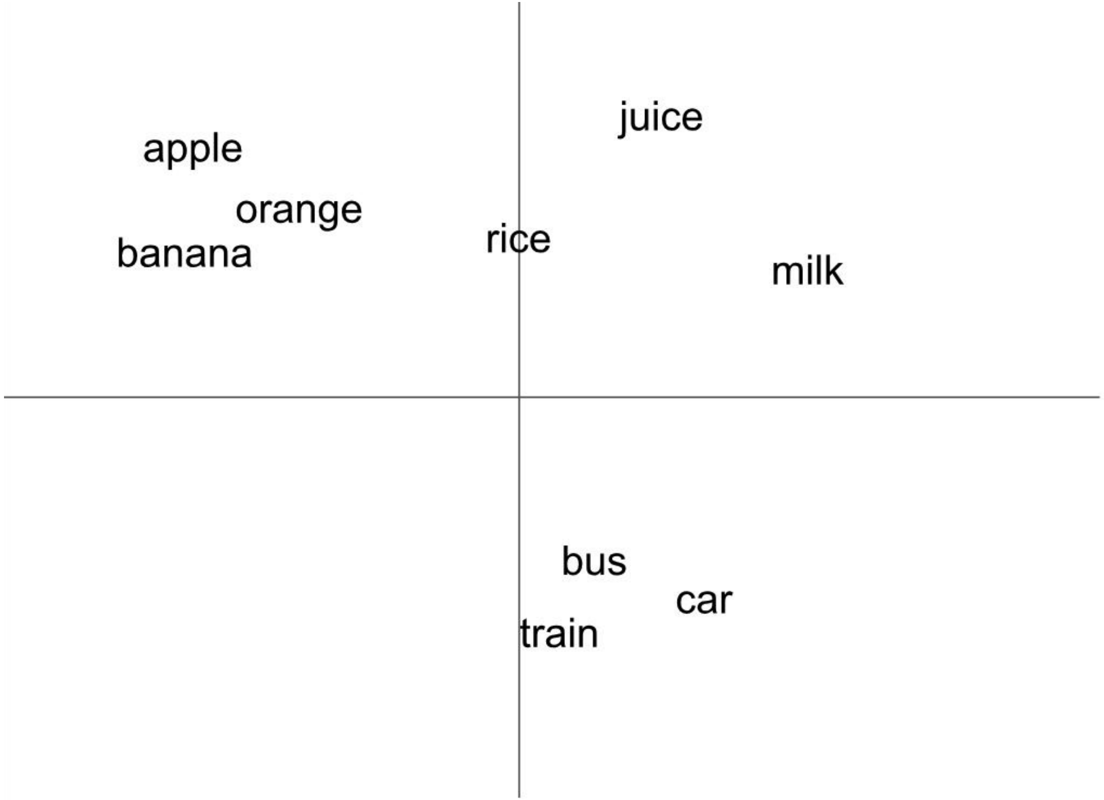
- Words should be encoded into a low-dimensional and dense vector



Word vectors

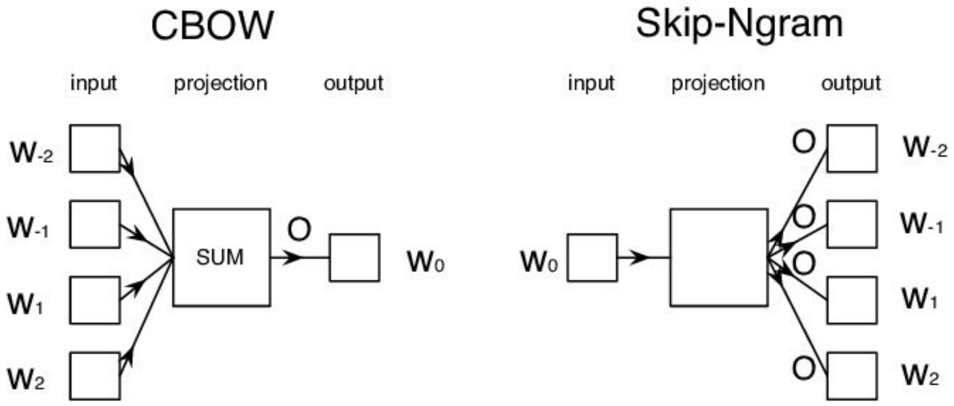
Project word vectors in a two-dimensional space. And visualize them!

Similar words are close to each other.

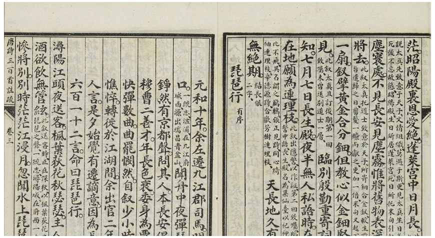


Word2Vec

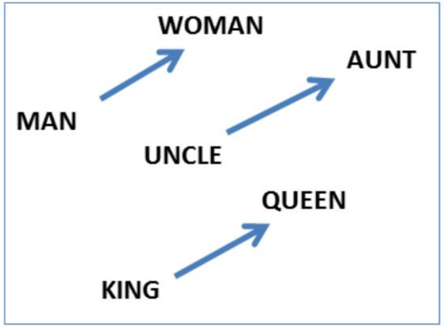
- A method of computing vector representation of words developed by Google.
- Open-source version of Word2Vec hosted by Google (in C)
- Train a simple neural network with a single hidden layer to perform word prediction tasks.
- Two structures proposed Continuous Bag of Words (CBoW) vs Skip-Gram



Word2Vec as BlackBox



input, output



Corpus

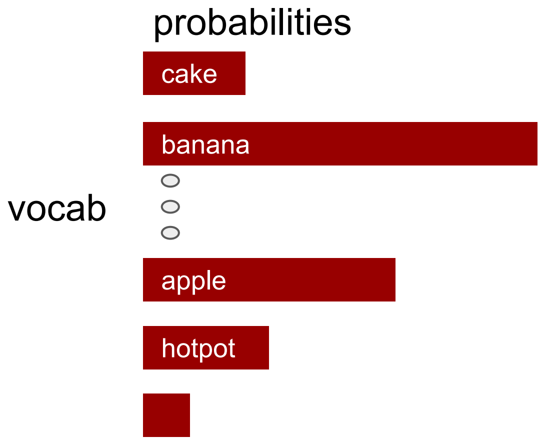
Word2Vec Tool

Word Embeddings

Use NN to predict word

Input words → Neural Network → Output words

Eat



Self-supervised learning

A Good Visualization for Word2Vec

https://ronxin.github.io/wevi/

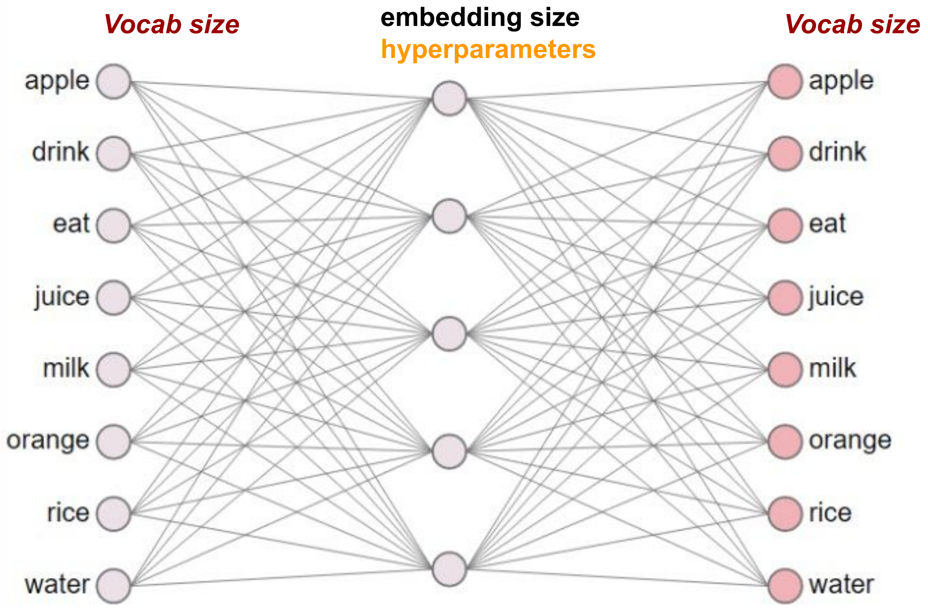
Target

- Given a training corpus, we prepare a list of N (input_word, output_word).
- Objective Function: Maximize probability of all the output words given the corresponding input words.

$$\mathbf{J}(\theta) = \prod_{i=1}^{N} p(w_{output}^{i} | w_{input}^{i}, \theta)$$

Neural network parameters that will be optimized

Model architecture



Structure Highlights:

- input layer
  - one-hot vector
- hidden layer
  - linear (identity)
- output layer
  - softmax

From Xin Rong 2016

Input layer

Give the training pair: eat -> apple (given eat, predict apple)

- 8 unique words are in the corpus so that the input layer has 8 neurons
- The index of eat is 3 in the vocab
- The input vector of the x(eat) would be:

One-hot vector

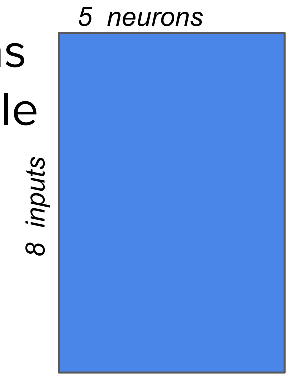
[0,0,1,0,0,0,0, 0]

Index of eat

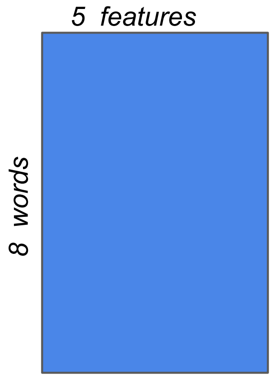
Hidden layer

- **Linear-activation** function here
- **5** neurons are the word vec. dimensions
- This layer is operating as a ‘lookup’ table
- Input word matrix denoted as **IVec**

Hidden Layer Weights Matrix



Word Vector Look Up Table



One-hot vector

[0,0,1,0,0,0,0, 0] ✗

Index of eat

1.06	2.91	0.29	1.39	0.33
1.60	1.12	0.29	0.74	0.21
0.96	1.50	1.37	0.34	1.04
0.53	2.11	0.76	2.51	0.20
0.31	0.64	2.08	0.24	1.23
1.40	1.36	0.01	1.69	1.95
2.97	2.13	0.86	0.90	2.21
1.05	0.80	2.18	2.43	1.57

=

Word vector for “eat”

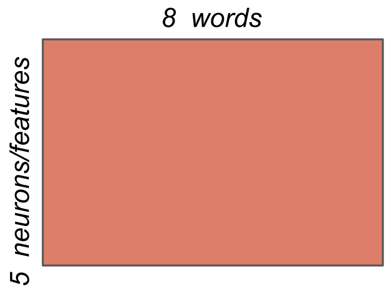
0.96, 1.5, 1.37, 0.34, 1.04

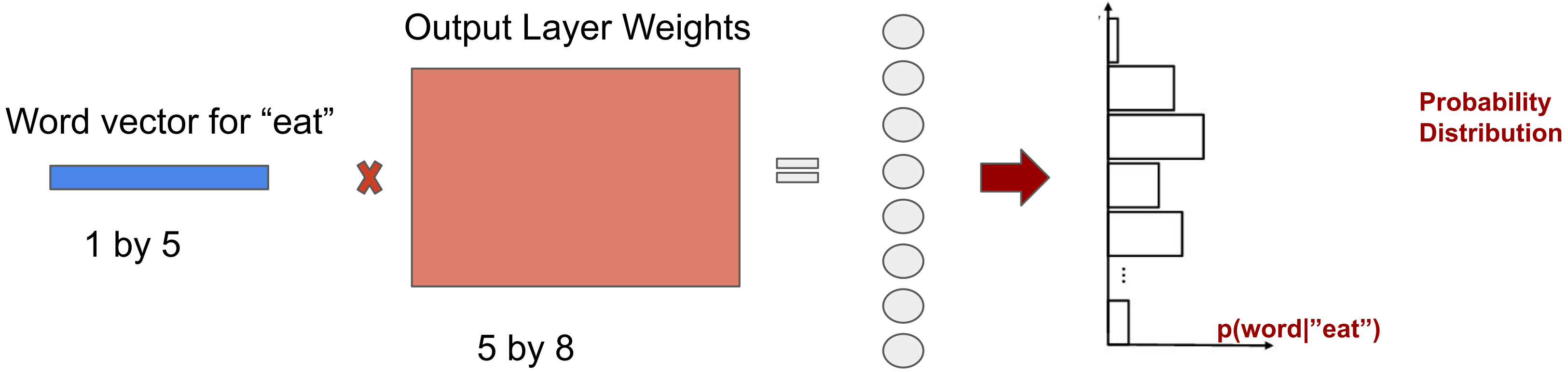
This is a **projection/look up** process: given the index of the word, we take the ith row in the word vector matrix out

Output layer

- Softmax Classifier
- Output word matrix denoted as **OVec**

Output Layer Weights Matrix
A.K.A Output word vectors



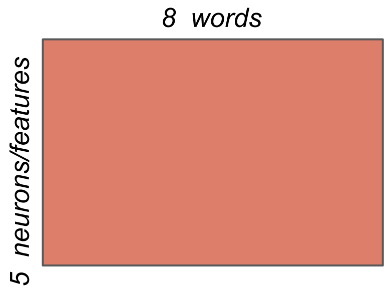


Scores over 8 words

Output layer

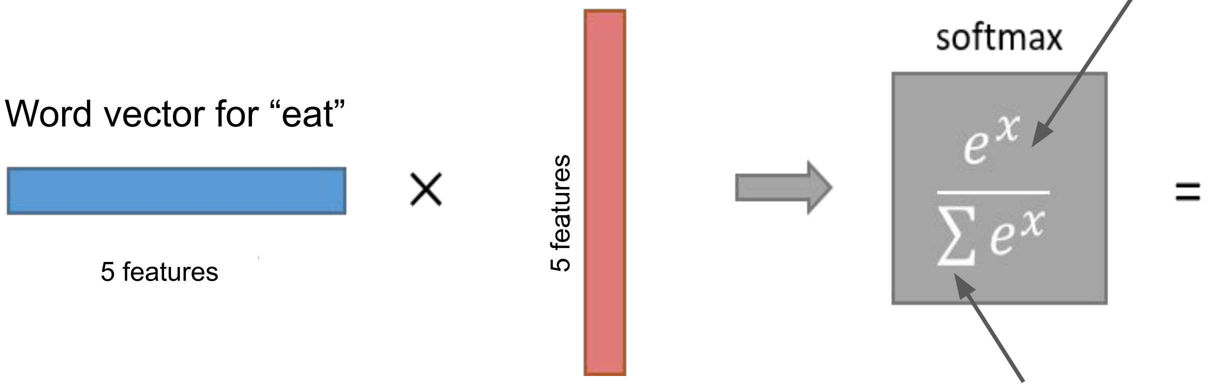
- Softmax Classifier
- Output word matrix denoted as **OVec**

Output Layer Weights Matrix
A.K.A Output word vectors



Output weights for “apple”

e^(IVec[eat]*OVec[apple])



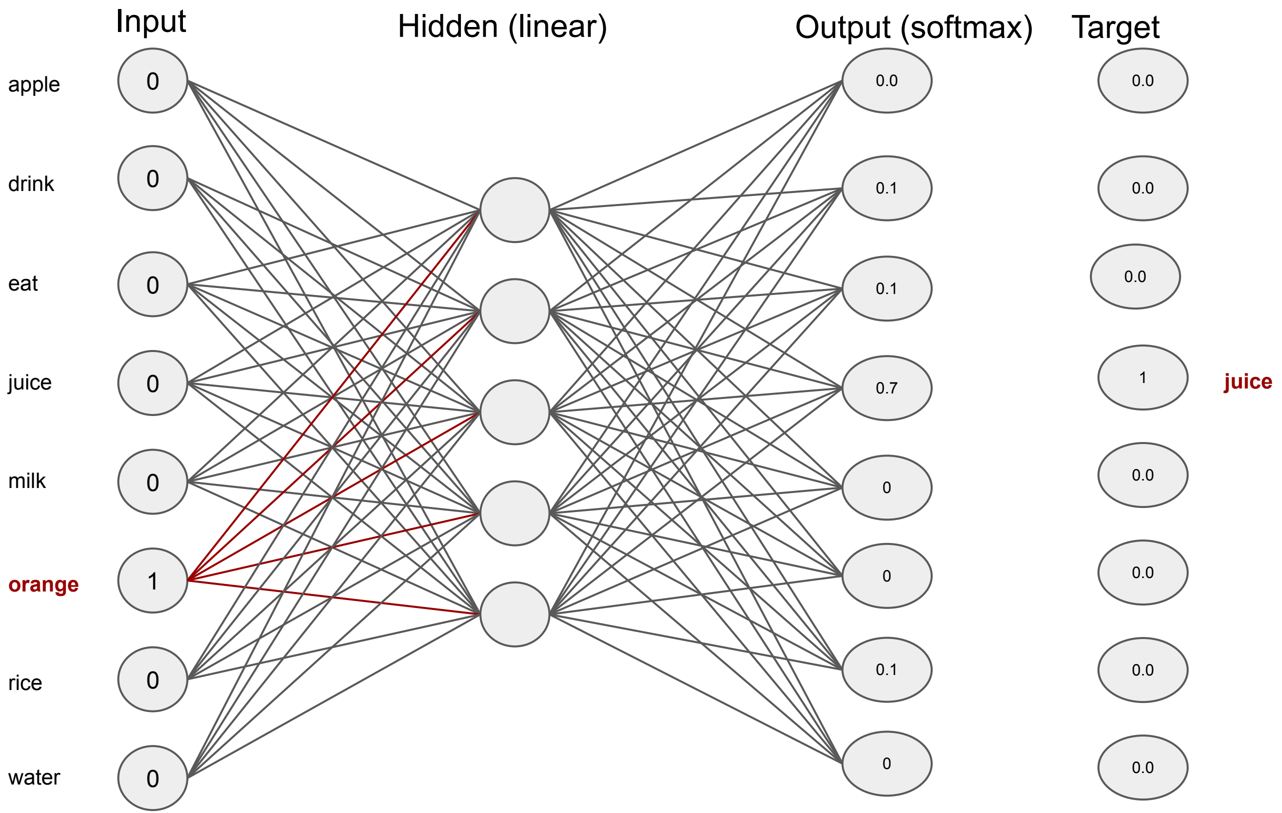
$$\frac{e^x}{\sum e^x} =$$

Probability that if you randomly pick a word nearby “eat”, that it is “apple”

p(apple|eat)

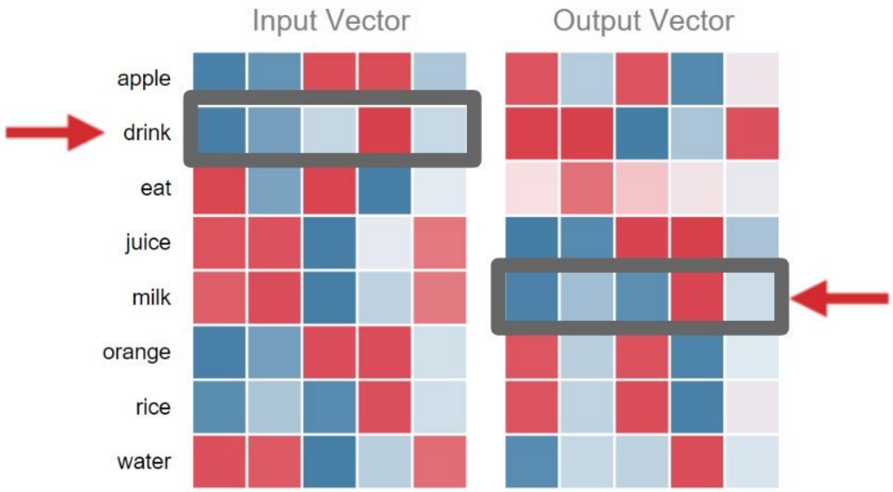
e^(IVec[eat]*OVec[apple]) + e^(IVec[eat]*OVec[juice]) + e^(IVec[eat]*OVec[drink])+...+e^(IVec[eat]*OVec[water])

Word2Vec



Then, we can compute the loss and call gradient descent to update model parameters.

Updating word vectors



From Xin Rong 2016

Input vs output word vectors

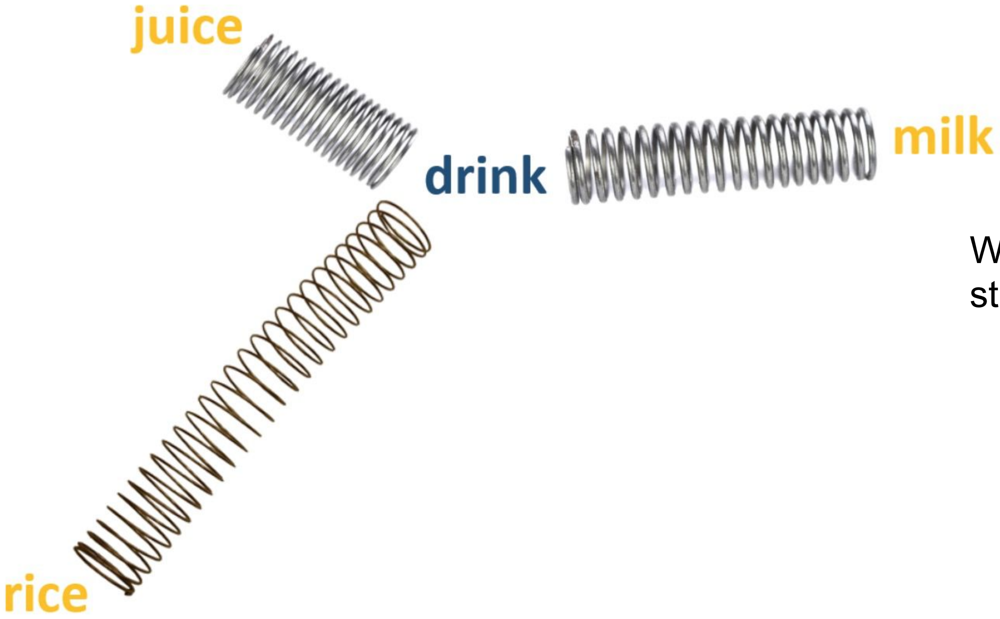
- Input matrix: semantics **encoder** from word index to semantics
- Output matrix: semantics **decoder** from semantics to probability distributions over words
- In most cases, input word vectors are used. Some have observed that combinations of these two vectors may perform better

	Vector size	Overall	Semantic	Syntactic
DVRS	300	0.41	0.59	0.26
DVRS	1024	0.43	0.62	0.28
SG	300	**0.64**	**0.69**	**0.60**
SG	1024	0.57	0.60	0.55
Add 300-DVRS, 300-SG	300	0.64	0.72	0.58
Concatenate 300-DVRS, 300-SG	600	**0.67**	**0.74**	**0.60**
Add 1024-DVRS, 1024-SG	1024	0.60	0.66	0.55
Concatenate 1024-DVRS, 1024-SG	2048	0.61	0.68	0.55
Concatenate DVRS-1024, SG-300	1324	0.66	0.73	**0.60**
Oracle DVRS-1024, SG-300	1024/300	0.70	0.79	0.62

Table 2: Performance on word analogy problems with vectors trained against the first 10^9 bytes of Wikipedia.

Garten, 2014

A force-directed graph



What decides the strength of the string?

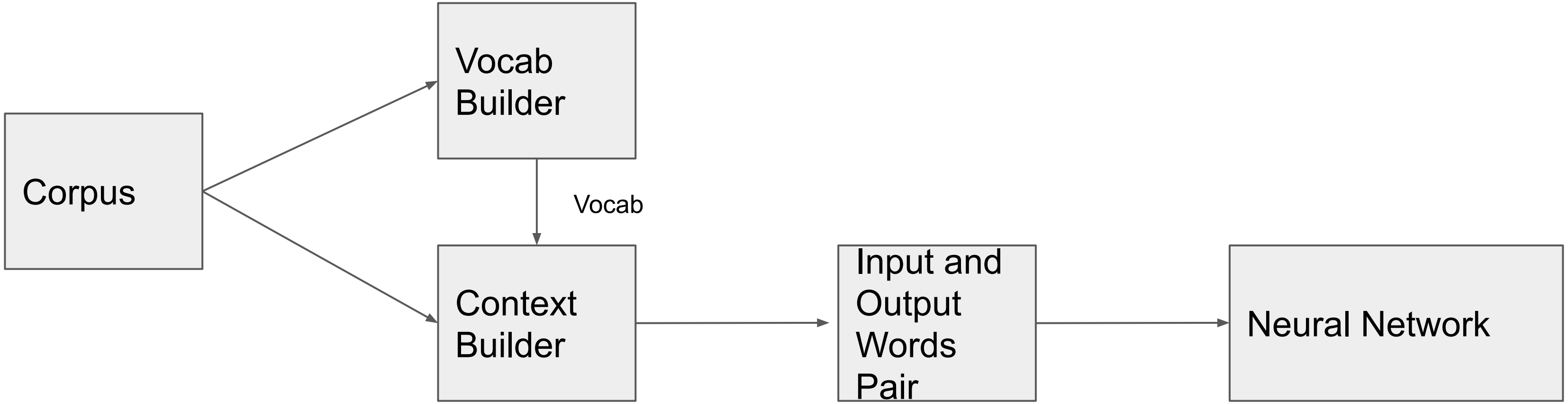
Idea behind Word2Vec

- Feature vector assigned to a word will be adjusted if it can not be used for accurate prediction of that word's context.
- Each word's context in the corpus is the teacher sending error signals back to modify the feature vector.
- It means that words with similar **context** will be assigned similar **vectors**!

"You shall know a word by the company it keeps" - by Firth (1957)

Input and output words

- How to select them from corpus
- Skip-gram and CBoW differ here



Skip-Gram

- Task Definition: given a specific word, predict its nearby word (probability output)
- Model input: source word, Model output: nearby word
- Input is one word, output is one word
- The output can be interpreted as prob. scores, which are regarded as how likely it is that each vocabulary word can be nearby your input word.

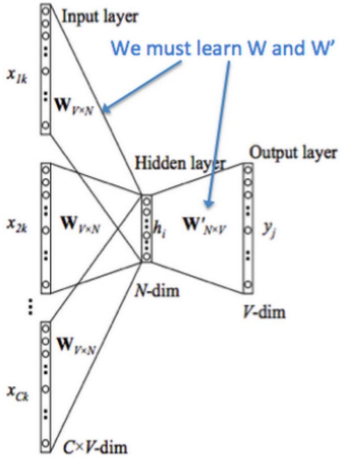
give a talk at the

Input x	Target y
talk	give
talk	a
talk	at
talk	the

CBoW

- Task Definition: given context, predict its target word
- Model input: context (several words), Model output: center word
- Input is several words, output is one word
- Core Trick: **average** these context vectors for prob. score computing

give a **talk** at the → **Input x (give,a,at,the)** **Target y talk**



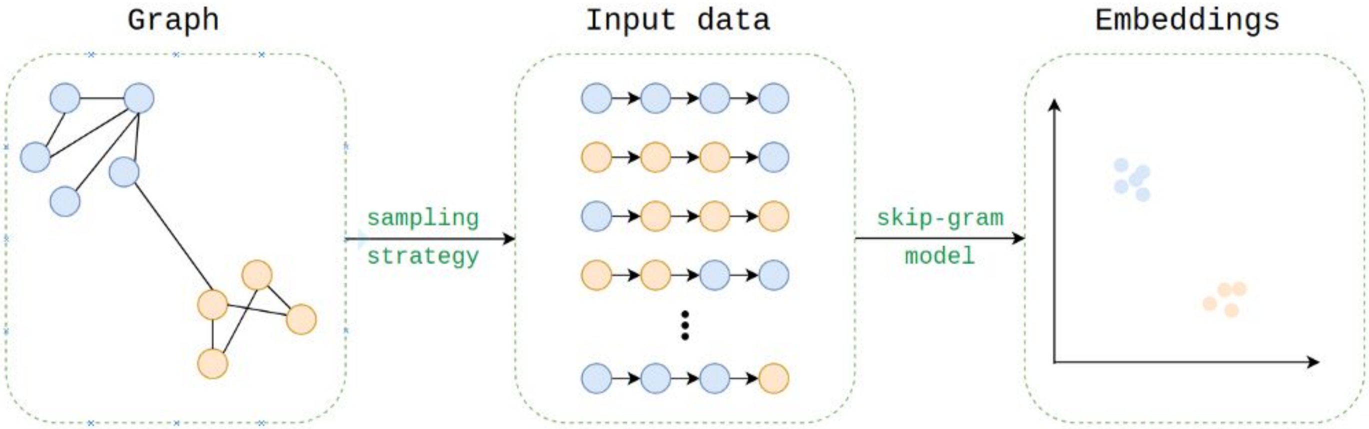
Skip-Gram vs CBoW

- Skip-gram:
  - Learning to predict the context by the center word
- CBoW:
  - Learning to predict the word by the context

- ? : several times faster to train the ?
- ? : works well with small amount of the training data, represents well even rare words or phrases.

Embedding for graph data

- Embeddings can be extended beyond NLP domain
- Embeddings can be learned for any nodes in a graph
- Nodes can be items, web pages and so on in user clicked stream data
- Embeddings can be learned for any group of discrete and co-occurring states.

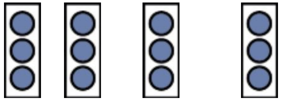


3. Neural Networks for NLP

Sequence of words

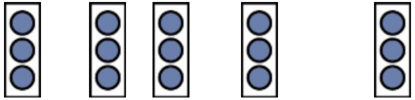
- Each sentence or document can be regarded as a sequence of vectors.
- The shape of matrix depends on the length of sequence. However, the majority of ML systems need fixed-length feature vectors.
- One simple solution: average the sequence of vectors, just like bag-of-words (abandon order information).

I hate this movie



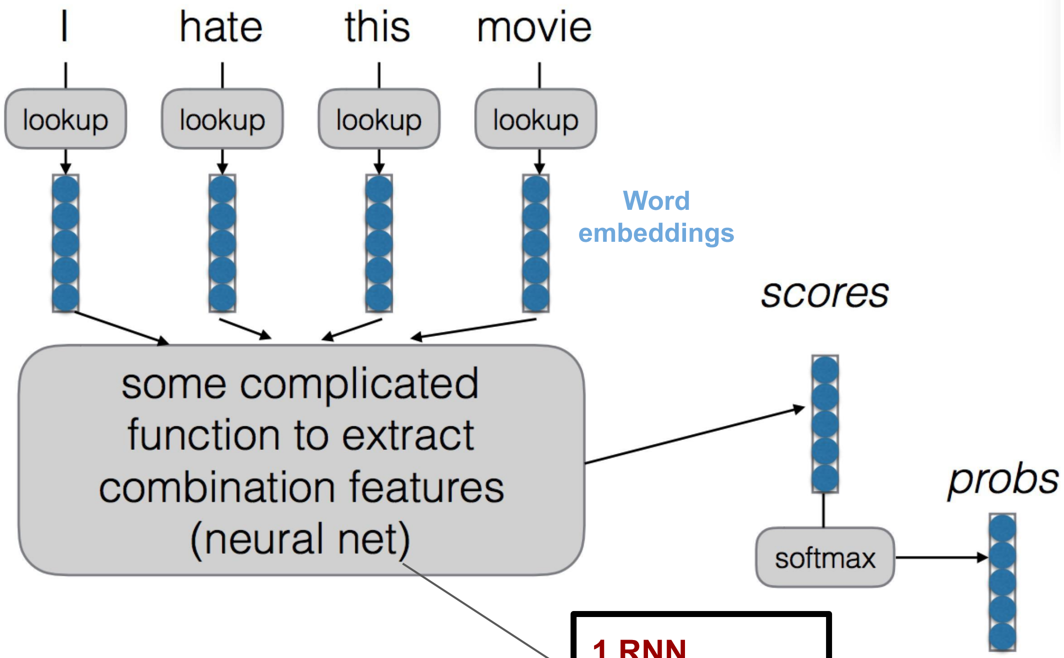
4 by d

This is my favorite movie.



5 by d

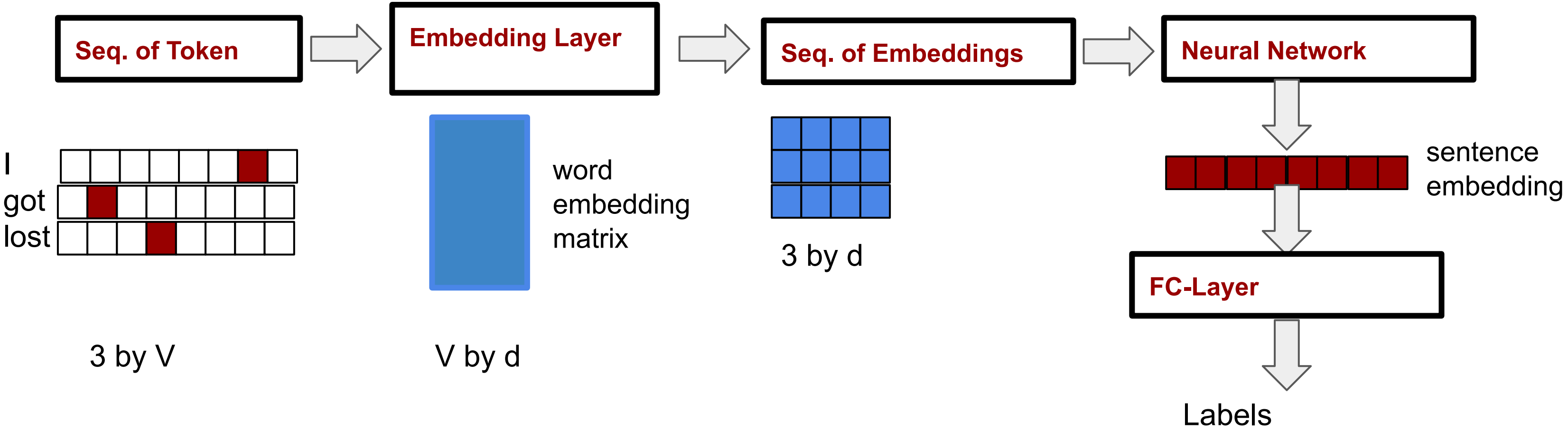
Complex semantic



1 RNN
2 CNN
3 Attention
etc

Word Embeddings is the foundation for deep learning's applications on NLP

Neural networks for NLP



Embedding layer: the fully-connected layer via one-hot encoding (no bias and no activation)

Is Word2Vec good enough?

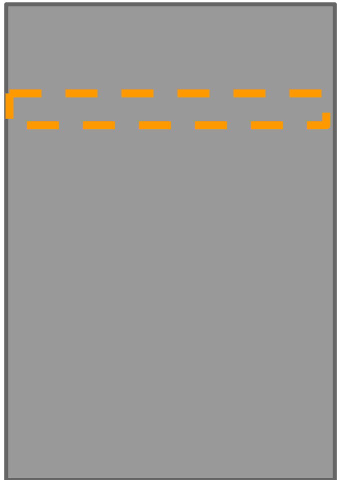
- Can not capture different senses of words (context independent)
  - Solution: Take the word order into account->context dependent
- Can not address Out-of-Vocabulary words
  - Solution: Use characters or **subwords**

Multi-sense of Words

- It is safest to deposit your money in the **bank**.
- All the animals lined up along the river **bank**.
- Today, blood **banks** collect blood.

The third sense of not?

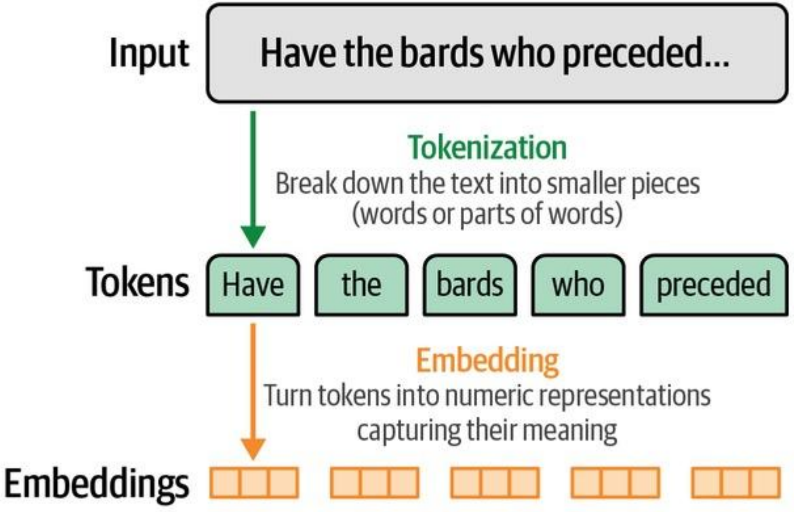
Word2Vec, Fasttext,Glove and other word embedding models



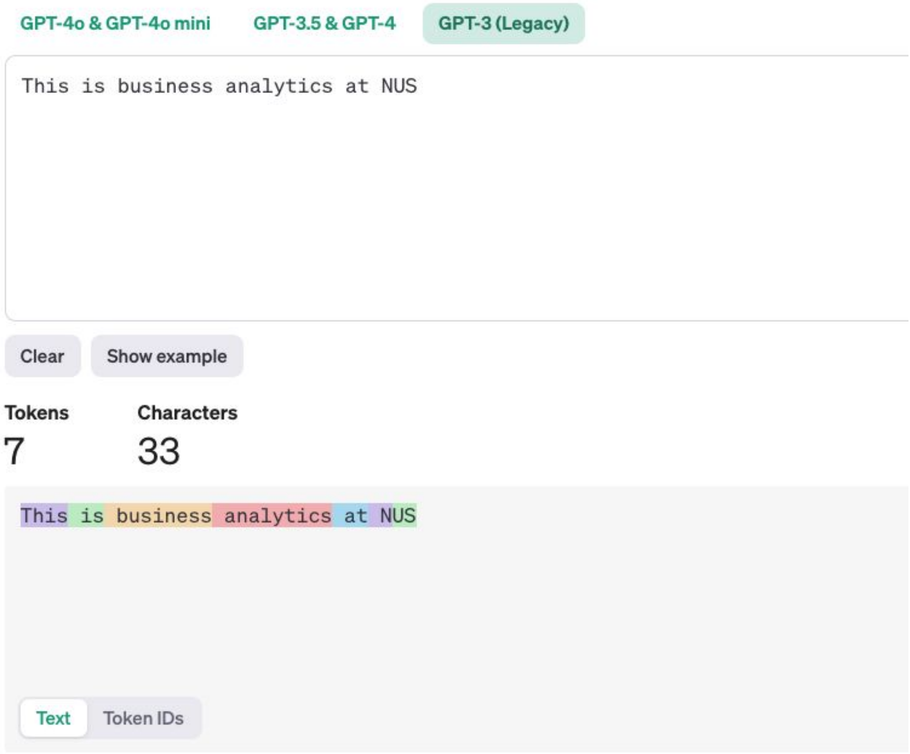
4. Tokens and Embeddings

Tokens and Embeddings

- Tokenization
  - LLM deal with text in small chunks called tokens.
- Embeddings:
  - The numeric representation for tokens



Tokenization



https://platform.openai.com/tokenizer

Tokenization Approach

- Word tokens
  - Used in word2vec
  - Unable to deal with new words
  - Result in a vocabulary that has a lot of tokens with minimal differences
    - Apology, Apologize, Apologetic, Apologist
- Subword tokens
  - Contains full and partial words
  - Able to represent new words by breaking down the new token into smaller characters
    - Apolog
      - Suffix tokens: -y, -ize, -etic, -ist

Tokenization: word level

Load tokenizers

Tokenize the sentences: split the sentence into a list of word and subwords defined in the predefined Vocab. The process is given as:

1. Tokenization: breaking down of the sentence into tokens
2. Adding the [CLS] token at the beginning of the sentence
3. Adding the [SEP] token at the end of the sentence
4. Converting each token into their corresponding IDs in the model

For some words, the BERT tokenization function will first breaks the word into two subwoards, namely characteristic and ##ally, where the first token is a more commonly-seen word (prefix) in a corpus, and the second token is prefixed by two hashes ## to indicate that it is a suffix following some other subwords.

```
text = "Hello! How are you doing? 👋 I'm at BT5153 to learn LLM"

# Basic tokenization
tokens = tokenizer.tokenize(text)
print(f"Original text: {text}")
print(f"Tokens: {tokens}")

# Convert to token IDs
token_ids = tokenizer.encode(text)
print(f"\nToken IDs: {token_ids}")

# Convert back to text
decoded_text = tokenizer.decode(token_ids)
print(f"Decoded text: {decoded_text}")

# Show special tokens
print("\nSpecial tokens in BERT:")
print(f"[CLS] token: {tokenizer.cls_token} (ID: {tokenizer.cls_token_id})")
print(f"[SEP] token: {tokenizer.sep_token} (ID: {tokenizer.sep_token_id})")
print(f"[PAD] token: {tokenizer.pad_token} (ID: {tokenizer.pad_token_id})")
print(f"[UNK] token: {tokenizer.unk_token} (ID: {tokenizer.unk_token_id})")
```

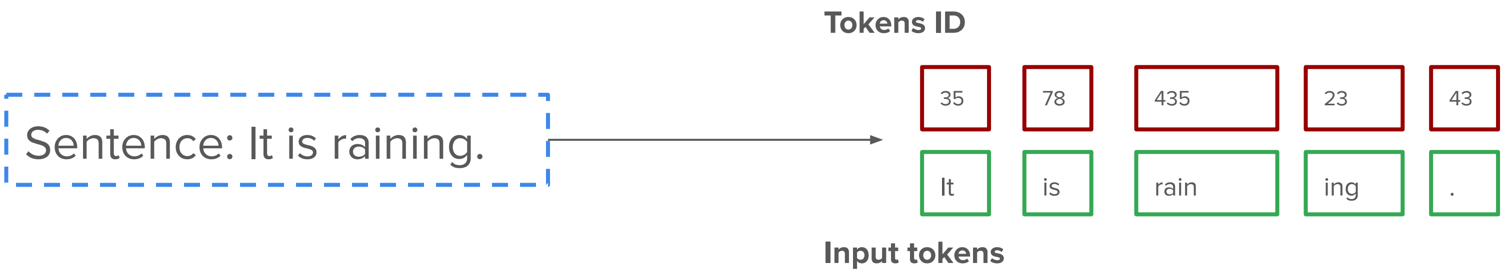
```
Original text: Hello! How are you doing? 👋 I'm at BT5153 to learn LLM
Tokens: ['hello', '!', 'how', 'are', 'you', 'doing', '?', '[UNK]', 'i', "'", 'm', 'at', 'bt', '##51', '##53', 't
o', 'learn', 'll', '##m']

Token IDs: [101, 7592, 999, 2129, 2024, 2017, 2725, 1029, 100, 1045, 1005, 1049, 2012, 18411, 22203, 22275, 2000,
4553, 2222, 2213, 102]
Decoded text: [CLS] hello! how are you doing? [UNK] i ' m at bt5153 to learn llm [SEP]

Special tokens in BERT:
[CLS] token: [CLS] (ID: 101)
[SEP] token: [SEP] (ID: 102)
[PAD] token: [PAD] (ID: 0)
[UNK] token: [UNK] (ID: 100)
```

Relies on a predefined vocabulary -> **Out-of-vocabulary issues**

Tokenization: subword level

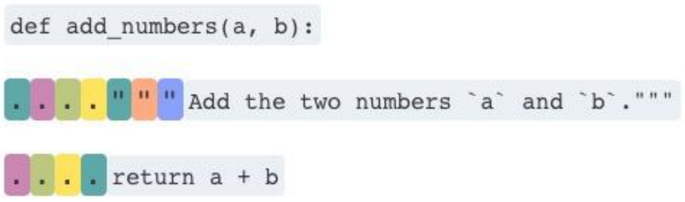


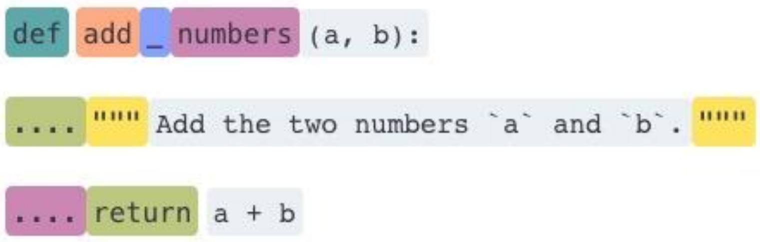
- Can represent out-of-vocabulary words by composing them from subword units
- The subword algorithms have two main modules:
  - A token learner: this takes a corpus as input and creates a vocabulary containing tokens
    - When should we decompose word into subwords and index those subwords
  - A token segmenter
    - Takes a piece of text and segments it into tokens

[Byte-pair Encoding tokenization]

Tokenizer Properties

- Tokenization methods
  - How to choose an appropriate set of tokens to represent a dataset
- Tokenizer parameters
  - Vocab size
  - Special tokens
  - Capitalization
- The domain of the data
  - Before the model training, the tokenization method optimized the vocabulary to represent a specific dataset





Tokenizer

- Tokenization:
  - Split text into tokens (words, subwords, punctuation, etc.) using **model-specific rules** to match the pretrained model.
- Numerical conversion
  - Convert tokens to number using the **model-specific vocabulary** (indexes), ensuring alignment with the pretrained models

If you do not want to re-train the model, you have to use its associated tokenizers.

```
print("Setting up tokenizer and model...")
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
model = AutoModel.from_pretrained("bert-base-uncased")
```

SAME

Bert Tokenizers

Load tokenizers

Tokenize the sentences: split the sentence into a list of word and subwords defined in the predefined Vocab. The process is given as:

1. Tokenization: breaking down of the sentence into tokens
2. Adding the [CLS] token at the beginning of the sentence
3. Adding the [SEP] token at the end of the sentence
4. Converting each token into their corresponding IDs in the model

For some words, the BERT tokenization function will first breaks the word into two subwoards, namely characteristic and ##ally, where the first token is a more commonly-seen word (prefix) in a corpus, and the second token is prefixed by two hashes ## to indicate that it is a suffix following some other subwords.

```
text = "Hello! How are you doing? 👋 I'm at BT5153 to learn LLM"

# Basic tokenization
tokens = tokenizer.tokenize(text)
print(f"Original text: {text}")
print(f"Tokens: {tokens}")

# Convert to token IDs
token_ids = tokenizer.encode(text)
print(f"\nToken IDs: {token_ids}")

# Convert back to text
decoded_text = tokenizer.decode(token_ids)
print(f"Decoded text: {decoded_text}")

# Show special tokens
print("\nSpecial tokens in BERT:")
print(f"[CLS] token: {tokenizer.cls_token} (ID: {tokenizer.cls_token_id})")
print(f"[SEP] token: {tokenizer.sep_token} (ID: {tokenizer.sep_token_id})")
print(f"[PAD] token: {tokenizer.pad_token} (ID: {tokenizer.pad_token_id})")
print(f"[UNK] token: {tokenizer.unk_token} (ID: {tokenizer.unk_token_id})")
```

```
Original text: Hello! How are you doing? 👋 I'm at BT5153 to learn LLM
Tokens: ['hello', '!', 'how', 'are', 'you', 'doing', '?', '[UNK]', 'i', "'", 'm', 'at', 'bt', '##51', '##53', 't
o', 'learn', 'll', '##m']

Token IDs: [101, 7592, 999, 2129, 2024, 2017, 2725, 1029, 100, 1045, 1005, 1049, 2012, 18411, 22203, 22275, 2000,
4553, 2222, 2213, 102]
Decoded text: [CLS] hello! how are you doing? [UNK] i ' m at bt5153 to learn llm [SEP]

Special tokens in BERT:
[CLS] token: [CLS] (ID: 101)
[SEP] token: [SEP] (ID: 102)
[PAD] token: [PAD] (ID: 0)
[UNK] token: [UNK] (ID: 100)
```

Next Class: From Word2Vec to Transformers

Suggested Reading: The illustrated transformer