CS 4063 – Natural Language Processing

Lecture Notes – week 4

Muhammad Hannan Farooq

Feature extraction from text

- Tokenization
- Bag of Words (BoW)
- Tf-IDF
- Word2Vec
- GloVe embeddings

Feature extraction from text

- Feature extraction from text is a critical step in Natural Language Processing (NLP) that involves transforming raw text data into a structured format suitable for machine learning models.
- The features extracted from text help capture the essence and meaning of the text, enabling models to perform tasks such as classification, sentiment analysis, or topic modeling.

Tokenization

- Tokenization is the process of breaking down text into smaller units, called tokens, which can be words, subwords, sentences, or even characters.\
- It is a fundamental step in Natural Language Processing (NLP), as it converts unstructured text data into structured input that can be processed by machine learning algorithms.

Types of Tokenization

1. Word Tokenization:
   - **Description:** In word tokenization, text is split into individual words. This is the most common form of tokenization.
   - **Example:** Input: "Natural Language Processing is fun!“
   - **Tokens:** ["Natural", "Language", "Processing", "is", "fun", "!"]
2. Sentence Tokenization:
   - **Description:** Sentence tokenization breaks down a text into individual sentences. This is useful when the task requires analyzing entire sentences or when punctuation plays a significant role.
   - **Example:** Input: "NLP is great. I love learning about it.“
   - **Tokens:** ["NLP is great.", "I love learning about it."]

Types of Tokenization(Cont.)

1. Subword Tokenization:
   - **Description:** In subword tokenization, words are broken down into smaller meaningful parts (subwords). This is useful for handling rare words, out-of-vocabulary words, or complex languages.
   - **Popular Methods:** Byte Pair Encoding (BPE), WordPiece, Unigram.
   - **Example (WordPiece):**
     - **Input:** "unbelievable“
     - **Tokens:** ["un", "##believe", "##able"]
     - The "##" symbol indicates that the token is a continuation of the previous word part.

Types of Tokenization(Cont.)

1. Character Tokenization:
   - **Description:** Character tokenization splits text into individual characters. This is useful in specific cases, such as language models for rare or unknown words or handling typos.
   - **Example:**
     - **Input:** "hello“
     - **Tokens:** ["h", "e", "l", "l", "o"]

Why Tokenization is Important?

1. **Text Preprocessing:** Tokenization is the first step in preparing text for machine learning models. It transforms raw text into manageable units.
2. **Text Representation:** Tokenization helps break text down into smaller pieces, making it easier to convert into numerical representations like Bag of Words, TF-IDF, or Word Embeddings.
3. **Handling Different Languages:** Tokenization must account for the complexity of different languages (e.g., handling spaces, punctuation, compound words, etc.).

Tokenization Challenges

1. **Punctuation Handling:** In some cases, punctuation should be preserved, while in others, it should be removed. This depends on the task, such as sentiment analysis or named entity recognition**.**
   - **Example:** "I'm happy!" → ["I", "'m", "happy", "!"]
2. **Handling Contractions and Special Characters:**
   - Tokenizers must handle contractions like "don't" or "we'll" appropriately, as well as numbers, symbols, and special characters.
   - **Example:** "Can't" → ["Ca", "n't"] (depending on tokenizer settings)
3. **Ambiguity in Tokenization for Different Languages:**
   - Languages like Chinese and Japanese do not use spaces between words, making word tokenization much more challenging. Special tokenizers are needed for such languages.

Applications of Tokenization

- **Sentiment Analysis:** Tokenization breaks down reviews or social media posts into words or sentences for sentiment classification.
- **Text Classification:** Tokenization prepares text data for machine learning models that classify documents or news articles.
- **Machine Translation:** Tokenization is used to split text into words or subwords before translating between languages.
- **Chatbots and Virtual Assistants:** Tokenization helps these systems break down user input to extract key commands and actions.

Bag of Words(BoW)

- Converts a collection of text documents into numerical feature vectors.
- grammar, word order, and context, and focuses only on the frequency or presence of words.

How BoW Works

- Tokenization.
- Vocabulary Creation
- Vector Representation

BoW Example

- "I love NLP.“
- "NLP is fun.“
- "I love learning NLP."

Final Bag of Words Matrix

Document	I	love	NLP	is	fun	learning
Doc 1	1	1	1	0	0	0
Doc 2	0	0	1	1	1	0
Doc 3	1	1	1	0	0	1

Anothrer Example

- "I love NLP. I love machine learning.“
- "NLP is fun and NLP is powerful.“
- "I enjoy learning NLP and NLP helps in AI."

Pros of BoW

- Simple and easy to implement.
- Effective for basic tasks like document classification or spam filtering.

Cons of BoW

- Ignores word order and semantics.
- High-dimensional representation, especially for large vocabularies.
- Unable to capture context or meaning of words (e.g., "bank" in "river bank" vs "money bank").

Type of BoW

- Binary Bag of Words
- Frequency Bag of Words (Standard BoW)
- TF-IDF Bag of Words
- N-gram Bag of Words
- Continuous Bag of Words (CBOW)

Term Frequency-Inverse Document Frequency (TF-IDF)

- **TF-IDF** is a more advanced feature extraction technique that takes into account not just the frequency of words (term frequency), but also how important a word is in the entire corpus (inverse document frequency).
- This helps reduce the impact of common words (like "the," "is") and emphasizes words that are more unique to each document.
- **How it Works:** TF measures how often a term occurs in a document, while IDF measures how unique or rare a term is across all documents. The TF-IDF score is the product of TF and IDF.
- **Example:** In a corpus, "NLP" may have a high TF-IDF score because it’s unique to one document, while common words like "the" will have a lower score.

Key Components of TF-IDF

- **TF-IDF is a combination of two metrics:**
  - Term Frequency (TF): Measures how frequently a term appears in a document.
  - Inverse Document Frequency (IDF): Measures how important a term is, based on how many documents in the corpus contain the term.

$$\text{TF-IDF}(w, d) = \text{TF}(w, d) \times \text{IDF}(w)$$

Term Frequency (TF)

TF measures how often a word appears in a document compared to the total number of words in the document.

$$\text{TF}(w, d) = \frac{\text{Number of times word } w \text{ appears in document } d}{\text{Total number of words in document } d}$$

Inverse Document Frequency (IDF)

IDF measures the importance of a word by considering how common or rare it is across all documents in the corpus. The more rare a word is, the higher its IDF score.

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

Final TF-IDF Calculation?

Example with Detailed Calculation

- **Let’s take a corpus with three documents:**
- Document 1: "I love machine learning“
- Document 2: "Machine learning is fun“
- Document 3: "Learning AI is exciting”

Word	TF (Doc 1)	IDF	TF-IDF
I	0.25	0.18	$0.25 \times 0.18 = 0.045$
love	0.25	1.10	$0.25 \times 1.10 = 0.275$
machine	0.25	0.18	$0.25 \times 0.18 = 0.045$
learning	0.25	0.00	$0.25 \times 0.00 = 0.00$

Word	TF in Doc 1	TF in Doc 2	TF in Doc 3
I	$\frac{1}{4} = 0.25$	$\frac{0}{4} = 0.00$	$\frac{1}{4} = 0.25$
love	$\frac{1}{4} = 0.25$	$\frac{0}{4} = 0.00$	$\frac{0}{4} = 0.00$
machine	$\frac{1}{4} = 0.25$	$\frac{1}{4} = 0.25$	$\frac{0}{4} = 0.00$
learning	$\frac{1}{4} = 0.25$	$\frac{1}{4} = 0.25$	$\frac{1}{4} = 0.25$
is	$\frac{0}{4} = 0.00$	$\frac{1}{4} = 0.25$	$\frac{1}{4} = 0.25$
fun	$\frac{0}{4} = 0.00$	$\frac{1}{4} = 0.25$	$\frac{0}{4} = 0.00$
AI	$\frac{0}{4} = 0.00$	$\frac{0}{4} = 0.00$	$\frac{1}{4} = 0.25$
exciting	$\frac{0}{4} = 0.00$	$\frac{0}{4} = 0.00$	$\frac{1}{4} = 0.25$

Word	Number of Documents Containing Word	IDF
I	2	$\log\left(\frac{3}{2}\right) = 0.18$
love	1	$\log\left(\frac{3}{1}\right) = 1.10$
machine	2	$\log\left(\frac{3}{2}\right) = 0.18$
learning	3	$\log\left(\frac{3}{3}\right) = 0.00$
is	2	$\log\left(\frac{3}{2}\right) = 0.18$
fun	1	$\log\left(\frac{3}{1}\right) = 1.10$
AI	1	$\log\left(\frac{3}{1}\right) = 1.10$
exciting	1	$\log\left(\frac{3}{1}\right) = 1.10$

Applications of TF-IDF

- **Text classification**
- **Document retrieval**
- **Keyword extraction**

Pros of TF-IDF

- Simple to Implement
- Effective at Capturing Word Importance
- Handles Common Words
- Sparse Representation

Cons of TF-IDF

- Ignores Word Order
- Fails to Capture Semantics
- Hard to Handle Polysemy
- Sensitive to Document Length
- Vocabulary Size Grows Quickly

N-Grams

- **N-grams** are contiguous sequences of n words in a document. For example, bigrams consist of two-word sequences, and trigrams consist of three-word sequences. N-grams help capture the context and order of words, which is useful for tasks like language modeling or sentiment analysis.
- **How it Works:** Create word pairs (bigrams) or triplets (trigrams) from the text. For example, in the sentence "I love NLP," the bigrams are: ["I love", "love NLP"].

N-Grams

- **N-grams** are contiguous sequences of n words in a document. For example, bigrams consist of two-word sequences, and trigrams consist of three-word sequences. N-grams help capture the context and order of words, which is useful for tasks like language modeling or sentiment analysis.
- **How it Works:** Create word pairs (bigrams) or triplets (trigrams) from the text. For example, in the sentence "I love NLP," the bigrams are: ["I love", "love NLP"].

Example of N-grams

- Sentence: "I love machine learning“
- Unigram (n=1)
- Unigrams=[I],[love],[machine],[learning]
- Bigram (n=2)
- Bigrams=[Ilove],[lovemachine],[machinelearning]
- Trigram (n=3)
- Trigrams=[Ilovemachine],[lovemachinelearning]

Another Example of N-grams

- Sentence: "The cat sat on the mat.“

Homework

- "I love machine learning“
- "I love natural language processing“
- "Machine learning is fun“
- "Natural language processing is exciting“
- **Create Bigrams**
- **Let's say we want to predict the next word after the phrase "I love". What will be the next predicted word?**

Applications of N-grams

- Text Generation
- Text Classification
- Speech Recognition

Pros and Cons

Pros	Cons
Captures Word Context	High Dimensionality
Improves Feature Extraction	Sparsity
Phrase Detection	Data Hunger
Improved Text Classification	Memory and Computational Cost
Simple to Implement	Ignores Long-Range Dependencies

Word Embeddings (Word2Vec, GloVe)

- **Word embeddings** are dense vector representations of words that capture their semantic meaning by placing similar words closer together in a vector space. Popular methods include Word2Vec, and GloVe.
- **How it Works:** Word embeddings are trained on large text corpora and learn relationships between words based on context. For example, "king" and "queen" might be closer to each other in the vector space because they share semantic meaning.
- **Applications:** Useful for sentiment analysis, text classification, and more.

Word2Vec

- **Word2Vec** is a popular neural network-based model introduced by researchers at Google in 2013 that transforms words into high-dimensional vectors of real numbers. These vectors, known as **word embeddings**, capture semantic relationships between words in a continuous vector space. The core idea of Word2Vec is to place semantically similar words close together in the vector space, which helps in many natural language processing (NLP) tasks.
- Word2Vec can be trained using two different approaches:
  - Continuous Bag of Words (CBOW): Predicts a target word based on its context (i.e., surrounding words).
  - Skip-Gram: Predicts the surrounding context words based on a target word.

Continuous Bag of Words (CBOW)

- Objective: Given a set of context words, CBOW predicts the target (center) word.
- Input: The context words around a target word.
- Output: The target word.
- **Example:**
  - The cat sits on the mat
  - Our task is to predict the word "sits" using its surrounding words (context) within a window of 2 words. In CBOW, the context words are used to predict the target word.
  - First tokenize the word
  - Define Context and Target Words: We define a context window size of 2, meaning we will consider 2 words on either side of the target word for context. Our goal is to predict the word "sits" using its context words.
  - Vocabulary
  - One-Hot Encoding: The size of our vocabulary is 6. Each word will be represented by a 6-dimensional one-hot vector.

CBOW Architecture

- **Input Layer:** Taking the average of the one-hot encoded context words.
- **Hidden Layer:** Mapping the input through a hidden layer of neurons (embedding layer) to generate word vectors.
- **Output Layer:** Predicting the target word as a probability distribution over all words in the vocabulary.

- **Input:** The context words ["The", "cat", "on", "the"] are encoded as one-hot vectors.
- **Hidden Layer:** The hidden layer learns word embeddings (dense representations) of the context words.
- **Output Layer:** The output layer predicts the word "sits" as the target.

CBOW Architecture

- Average Context Vectors (Input Layer)

$$\text{Average vector} = \frac{[1,0,0,0,0,0] + [0,1,0,0,0,0] + [0,0,0,1,0,0] + [0,0,0,0,1,0]}{4}$$

$$= \frac{[1,1,0,1,1,0]}{4} = [0.25, 0.25, 0, 0.25, 0.25, 0]$$

This vector represents the input to the hidden layer.

Word	One-Hot Vector
The	[1, 0, 0, 0, 0, 0]
cat	[0, 1, 0, 0, 0, 0]
sits	[0, 0, 1, 0, 0, 0]
on	[0, 0, 0, 1, 0, 0]
the	[0, 0, 0, 0, 1, 0]
mat	[0, 0, 0, 0, 0, 1]

CBOW Architecture

- Hidden Layer (Word Embeddings)
- In the hidden layer, the model learns to project the input (average context vector) into a lower-dimensional space, resulting in word embeddings. Let's assume that we want the embedding size to be 3, meaning each word will be represented by a 3-dimensional vector.
- The hidden layer has a weight matrix W of size
- [6 (input dimensions) x 3 (embedding dimensions)]
- that transforms the 6-dimensional input into a 3-dimensional output.
- Suppose the weight matrix W looks like this:

$$W = \begin{bmatrix} 0.2 & 0.1 & 0.3 \\ 0.4 & 0.2 & 0.5 \\ 0.1 & 0.3 & 0.2 \\ 0.5 & 0.1 & 0.4 \\ 0.3 & 0.6 & 0.3 \\ 0.2 & 0.5 & 0.1 \end{bmatrix}$$

CBOW Architecture

- Hidden Layer (Word Embeddings)
- Embedding Vector=[0.25,0.25,0,0.25,0.25,0]×W:

Performing the matrix multiplication:

$$\text{Embedding Vector} = 0.25 \times [0.2, 0.1, 0.3] + 0.25 \times [0.4, 0.2, 0.5] + 0.25 \times [0.5, 0.1, 0.4] + 0.25 \times [0.3, 0.6, 0.3]$$

$$= [0.05 + 0.1 + 0.125 + 0.075, 0.025 + 0.05 + 0.025 + 0.15, 0.075 + 0.125 + 0.1 + 0.075]$$

$$= [0.35, 0.25, 0.375]$$

This 3-dimensional vector `[0.35, 0.25, 0.375]` is the **word embedding** that represents the context.

This is the **embedding vector** that represents the context for the word **"sits"**.

CBOW Architecture

- Cosine Similarity:
- Now, let's calculate the cosine similarity between two words based on their embeddings. Cosine similarity measures the cosine of the angle between two vectors and is commonly used to measure how similar two word vectors are. The formula is:

$$\text{Cosine Similarity} = \frac{\vec{A} \cdot \vec{B}}{||\vec{A}|| \times ||\vec{B}||}$$

- $\vec{A}$ and $\vec{B}$ are two word vectors.
- $\cdot$ denotes the dot product.
- $||\vec{A}||$ and $||\vec{B}||$ are the magnitudes (norms) of the vectors.

CBOW Architecture

- Let’s assume we want to find the similarity between the word "cat" and the word "mat". From the weight matrix W, we already know their embeddings:
- Embedding of "cat": [0.4, 0.2, 0.5]
- Embedding of "mat": [0.2, 0.5, 0.1]

1. **Dot Product:**

$$\vec{A} \cdot \vec{B} = (0.4 \times 0.2) + (0.2 \times 0.5) + (0.5 \times 0.1)$$

$$= 0.08 + 0.1 + 0.05 = 0.23$$

2. **Magnitude of "cat" vector:**

$$||\vec{A}|| = \sqrt{(0.4^2 + 0.2^2 + 0.5^2)} = \sqrt{(0.16 + 0.04 + 0.25)} = \sqrt{0.45} = 0.67$$

3. **Magnitude of "mat" vector:**

$$||\vec{B}|| = \sqrt{(0.2^2 + 0.5^2 + 0.1^2)} = \sqrt{(0.04 + 0.25 + 0.01)} = \sqrt{0.3} = 0.55$$

4. **Cosine Similarity:**

$$\text{Cosine Similarity} = \frac{0.23}{0.67 \times 0.55} = \frac{0.23}{0.3685} \approx 0.624$$

The cosine similarity between the embeddings of **"cat"** and **"mat"** is **0.624**, indicating moderate similarity between these words in this vector space.

Home work

- Calculate the cosine similary of all words?

	The	cat	sits	on	the	mat
The	1					
Cat		1				
Its			1			
On				1		
the					1	
mat						1