

# Preprocessing and Feature Representation

Zeham Management Technologies BootCamp  
by SDAIA

September 3rd, 2024



**SDAIA**  
الهيئة السعودية للبيانات  
والذكاء الاصطناعي  
Saudi Data & AI Authority



# Agenda



Lemmatization and Stemming.



PoS Tagging



Named Entity Recognition (NER).



Bag of words.



TF-IDF.



Word embedding.

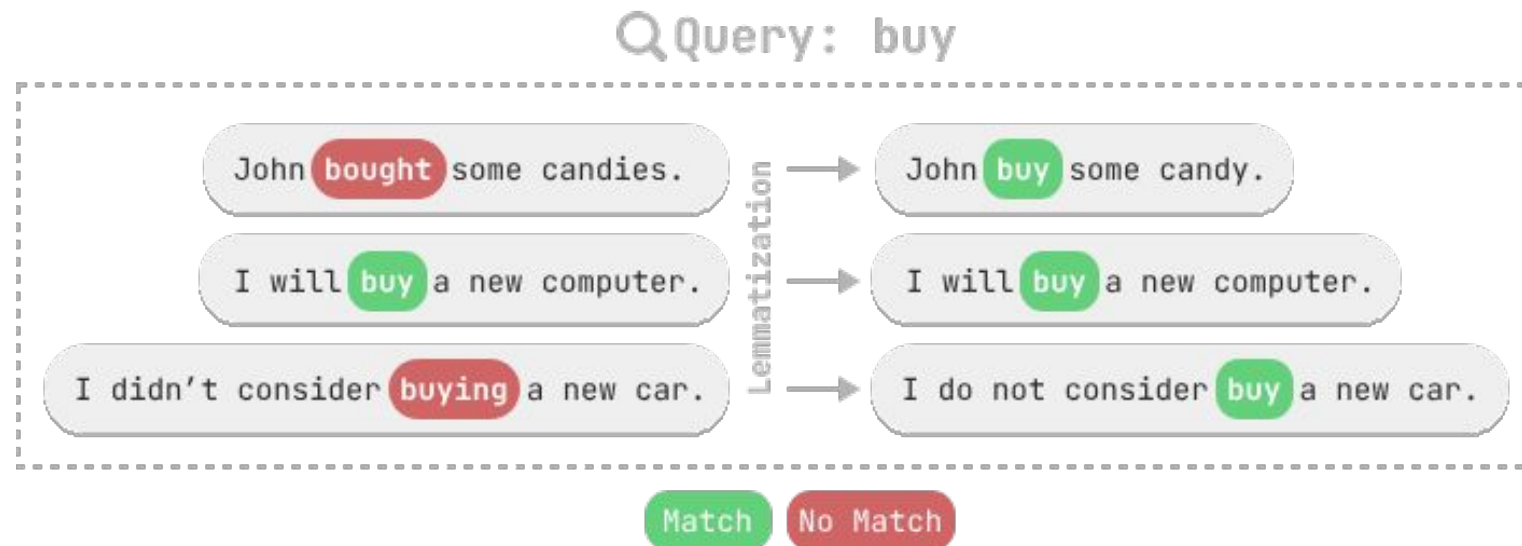


# Lemmatization and Stemming

---

# Text Preprocessing Techniques: Lemmatization

Lemmatization is a text normalization technique used in natural language processing (NLP) to reduce words to their base or dictionary form, known as the lemma. It is used to process words such that different grammatical variants of a word are treated as the same term. It stems the word but makes sure that it does not lose its meaning. Lemmatization has a pre-defined dictionary that stores the context of words and checks the word in the dictionary while diminishing.



# Text Preprocessing Techniques: Lemmatization

```
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
lemmatizer = WordNetLemmatizer()

def lemma_words(text):
    word_tokens = word_tokenize(text)
    lemmas = [lemmatizer.lemmatize(word) for word in word_tokens]
    return lemmas

input_str = "data science uses scientific methods algorithms and many
types of processes"
lemma_words(input_str)
```

Output:

```
['data', 'science', 'us', 'scientific', 'method', 'algorithm',
 'and', 'many', 'type', 'of', 'process']
```



# Text Preprocessing Techniques:

## Stemming

It is also known as the text standardization step where the words are stemmed or diminished to their root/base form. For example, words like 'programmer', 'programming', 'program' will be stemmed to 'program'.

But the disadvantage of stemming is that it stems the words such that its root form loses the meaning, or it is not diminished to a proper English word.

- There are various algorithms that can be used for stemming,
  1. Porter Stemmer algorithm
  2. Snowball Stemmer algorithm
  3. Lovins Stemmer algorithm
  4. ISRISemmer algorithm for Arabic text



# Text Preprocessing Techniques: Stemming

## Arabic Stemmer: ISRIStemmer

```
import nltk
from nltk.stem.isri import ISRIStemmer st =
ISRIStemmer()

# Define a function to perform stemming on the 'text'
column
def stem_words(words):
    return [st.stem(word) for word in words]

stem_words(['اهلا' ، 'مرحبا' ، 'معسكر' ، 'البيانات'])(
```

output:

```
[' اهل ' ، ' رجب ' ، ' عسكر ' ، ' بين ' ]
```



# Text Preprocessing Techniques:

## Stemming

- The difference between Stemming and Lemmatization can be understood with the example provided below:

Original Word	After Stemming	After Lemmatization
goose	goos	goose
geese	gees	goose
running	run	run
went	went	go



# Tutorial

7-Natural Language Processing/2 - Preprocessing and Feature

Representation/LAB/Stemming\_Lemmatization.ipynb

---



## Conclusion

- **Lemmatization:** Reduces words to their base or dictionary form, accounting for context, which improves accuracy in text processing.
- **Stemming:** Cuts words down to their root form, often faster but less accurate than lemmatization.
- **Key Takeaway:** Choose lemmatization for tasks requiring precision and context-awareness; use stemming for faster, less precise applications.

# Part of Speech Tagging (PoS Tagging)

---



## Text Preprocessing Techniques: Part-of-speech tagging

- Part-of-Speech Tagging is the process of assigning parts of speech to each word in a sentence. These parts of speech can include nouns, verbs, adjectives, adverbs, pronouns, conjunctions, prepositions, and more.
- POS tagging helps in understanding the grammatical structure of a sentence, which is essential for various Natural Language Processing (NLP) tasks such as syntactic parsing, word sense disambiguation, and information retrieval.



# Text Preprocessing Techniques: Part-of-speech tagging

## Common Algorithms for POS Tagging:

- **Rule-Based Taggers:** Use a set of hand-written rules.
- **Stochastic Taggers:** Use probabilistic methods such as Hidden Markov Models (HMM).
- **Transformation-Based Taggers (TBL):** Combine rule-based and stochastic approaches.
- **Neural Network Taggers:** Utilize deep learning models, such as Recurrent Neural Networks (RNNs) and Transformers.

# Text Preprocessing Techniques: Part-of-speech tagging

Part-of-speech tagging is used to identify the part of speech for each word in a sentence, such as nouns, verbs, adjectives, adverbs, and more.

```

# Sample text
text = "NLTK is a powerful library for natural language
processing."

# Performing PoS tagging
pos_tags = pos_tag(words)

# Displaying the PoS tagged result in separate lines
for word, pos_tag in pos_tags:
    print(f"{word}: {pos_tag}")
```

Output:

```

PoS Tagging
Result: NLTK:
NNP
is: VBZ
a: DT
powerful: JJ
library: NN
for: IN
natural: JJ
language: NN
processing: NN
```

**Currently, NLTK pos\_tag only supports English and Russian (i.e. lang='eng' or lang='rus')**



# Tutorial

7-Natural Language Processing/2 - Preprocessing and Feature

Representation/LAB/PoS\_Tagging.ipynb

---



## Conclusion

- **PoS Tagging:** Assigns grammatical categories (nouns, verbs, etc.) to words, providing syntactic structure crucial for many NLP tasks.
- **Key Takeaway:** PoS tagging enhances the comprehension of text structure, leading to better model performance in context-dependent tasks.



# Named Entity Recognition (NER)

---



# Text Preprocessing Techniques: Named entity recognition (NER)

Named Entity Recognition (NER) is a sub-task of information extraction that seeks to locate and classify named entities in text into predefined categories such as the names of persons, organizations, locations, dates, and more.

NER helps in structuring unstructured text, enabling more efficient information retrieval and data analysis. It's a critical step in many natural language processing (NLP) applications, such as search engines, recommendation systems, and question-answering systems.



# Text Preprocessing Techniques: Named entity recognition (NER)

## Common Algorithms for NER:

- **Rule-Based Approaches:** Utilize hand-crafted rules and lexicons.
- **Statistical Models:** Employ techniques such as Hidden Markov Models (HMM) and Conditional Random Fields (CRF).
- **Deep Learning Models:** Use architectures like Recurrent Neural Networks (RNN), Long Short-Term Memory networks (LSTM), and Transformers (e.g., BERT).



# Text Preprocessing Techniques: Named entity recognition (NER)

Named Entity Recognition (NER) identifies and classifies named entities in text, such as people, organizations, locations, etc..

```
from nltk import ne_chunk
from nltk.tokenize import
word_tokenize # Sample text
text = "Barack Obama was the 44th President of the United
States." words = word_tokenize(text)

# Performing Named Entity Recognition(NER)
ner_tags = ne_chunk(nltk.pos_tag(words))

# Displaying the NER tagged result in separate
lines for chunk in ner_tags:
    if hasattr(chunk, 'label'):
```

Output:

```
PERSON: Barack
PERSON: Obama GPE:
United States
```



# Tutorial

7-Natural Language Processing/2 - Preprocessing and Feature Representation/LAB/NER.ipynb

---



## Conclusion

- **NER:** Identifies and classifies named entities (e.g., people, organizations, locations) within text.
- **Key Takeaway:** NER adds semantic meaning to text, enabling models to understand and categorize important real-world entities.



# Text Representation

---

# Bag of words

---





# Text Representation Techniques: Bag of words.

Bag of words is a method for text representation in NLP.  
Treats text as a collection (or "bag") of words, disregarding grammar and word order.

## How BoW Works

- **Step 1:** Tokenization - Split text into individual words.
- **Step 2:** Vocabulary Creation - Identify all unique words.
- **Step 3:** Vectorization - Represent each document as a vector of word frequencies.

Example of BoW:

**Document:** "Cats are great pets. Cats are cute."

**Vocabulary:** {"cats", "are", "great", "pets", "cute"}

**Vector Representation:** [2, 2, 1, 1, 1]



# Text Preprocessing Techniques: Bag of words.

```
documents = [
    "Cats are great pets.",
    "Dogs are also great pets.",
    "Cats and dogs are cute."
]

# Step 1: Tokenize the documents
tokenized_docs = [doc.lower().split() for doc in documents]

# Step 2: Create a vocabulary of unique words
vocabulary = sorted(set(word for doc in tokenized_docs for word in doc))

# Step 3: Create the BoW matrix
bow_matrix = []
for doc in tokenized_docs:
    word_count = [doc.count(word) for word in vocabulary]
    bow_matrix.append(word_count)
```

```
[
    [0, 0, 1, 1, 0, 0, 1, 1], # "Cats are great pets."
    [1, 0, 1, 0, 0, 1, 1, 1], # "Dogs are also great pets."
    [0, 1, 1, 1, 1, 1, 0, 0] # "Cats and dogs are cute."
]
```

# Tutorial

7-Natural Language Processing/2 - Preprocessing and Feature  
Representation/LAB/BagOfWords.ipynb

---



## Conclusion

- **Bag of Words (BoW):** Represents text as a collection of word occurrences, ignoring grammar and word order.
- **Key Takeaway:** BoW is a foundational technique that is easy to implement but often requires additional methods to capture contextual information.

# Term Frequency-Inverse Document Frequency (TF-IDF)



# Text Preprocessing Techniques: TF-IDF

A statistical measure to evaluate how important a word is to a document in a collection. Balances the frequency of a term in a document with its inverse frequency across all documents.

## How TF-IDF Works

- **Term Frequency (TF):**

Measures how frequently a term appears in a document.

$$TF = \frac{\text{Number of times the term appears in a document}}{\text{Total numbers of term in the document}}$$

- **Inverse Document Frequency (IDF):**

Measures how important a term is by decreasing the weight of common terms.

$$IDF = \log \left( \frac{\text{Total number of document}}{\text{Numbers of documnets containg the term}} \right)$$

- **TF-IDF Formula:**

$$TF-IDF = TF \times IDF$$





# Text Preprocessing Techniques: TF-IDF

```
documents = [
    "Cats are great pets.",
    "Dogs are also great pets.",
    "Cats and dogs are cute."
]

# Tokenization and Vocabulary Creation
tokenized_docs = [doc.lower().split() for doc in documents]
vocabulary = sorted({word for doc in tokenized_docs for word in doc})

# Compute TF-IDF
tfidf_matrix = []
num_docs = len(documents)

for doc in tokenized_docs:
    tfidf_vector = []
    for word in vocabulary:
        tf = doc.count(word) / len(doc)
        idf = math.log(num_docs / (1 + sum(word in d for d in tokenized_docs))) + 1
        tfidf_vector.append(tf * idf)
    tfidf_matrix.append(tfidf_vector)
```

```
[
    [0.0, 0.0, 0.1781, 0.25, 0.0, 0.0, 0.25, 0.25],
    [0.2811, 0.0, 0.1425, 0.0, 0.0, 0.2, 0.2, 0.2],
    [0.0, 0.2811, 0.1425, 0.2, 0.2811, 0.2, 0.0, 0.0]
]
```

# Tutorial

7-Natural Language Processing/2 - Preprocessing and Feature Representation/LAB/TF-IDF.ipynb

---





## Conclusion

- **TF-IDF:** Weighs terms by their frequency and importance across documents, reducing the impact of common words.
- **Key Takeaway:** TF-IDF is a powerful method for identifying significant terms, balancing term frequency with global relevance.

# Word embedding

---



# Text Representation Techniques: Word Embedding

Dense vector representations of words.

Captures semantic meanings and relationships between words, Unlike Bag of Words or TF-IDF, embeddings consider context and relationships.

## How Word Embeddings Work

- Vector Representation:
  - Words are mapped to continuous vector space.
  - Each word is represented by a vector of real numbers.
- Contextual Relationships:
  - Words with similar meanings have similar vectors.
  - Captures relationships like synonyms, analogies (e.g., "king" - "man" + "woman"  $\approx$  "queen").

# Text Preprocessing Techniques: Word Embedding

```
from gensim.models import Word2Vec

# Sample corpus
sentences = [
    ["cats", "are", "great", "pets"],
    ["dogs", "are", "also", "great", "pets"],
    ["cats", "and", "dogs", "are", "cute"]
]

# Train a Word2Vec model
model = Word2Vec(sentences, vector_size=100, window=5, min_count=1, workers=4)


# Get the vector for a specific word
cat_vector = model.wv['cats']

# Find most similar words
similar_words = model.wv.most_similar('cats')

# Display the results
cat_vector, similar_words
```

Vector for "cats": A 10-dimensional vector representation of the word "cats":


CSS

 Copy code

```
[-0.096, 0.050, -0.088, -0.044, -0.0004, -0.003, -0.077, 0.096, 0.050, 0.092]
```

Most Similar Words to "cats":

CSS

 Copy code

```
[ ('and', 0.6144), ('cute', 0.2495), ('are', -0.2242), ('also', -0.2879),
```

# Tutorial

7-Natural Language Processing/2 - Preprocessing and Feature  
Representation/LAB/Word\_Embeddings.ipynb

---



## Conclusion

- **Word Embedding:** Transforms words into dense vectors that capture semantic meaning and relationships.
- **Key Takeaway:** Word embeddings provide rich, context-aware representations of text, making them essential for modern NLP models.

# Thank you!



**SDAIA**  
الهيئة السعودية للبيانات  
والذكاء الاصطناعي  
Saudi Data & AI Authority