NLP

1

Regex

*	+	?	{n,m}
at least 0 times	at least 1 times	at most 1 times	between **n** and **m** times
abc* "ab" followed by 0 or more "c"	abc+ "ab" followed by 1 or more "c"	abc? "ab" followed by 0 or 1 "c"	abc{3,} "ab" followed by at least 3 "c"
^	**$**	**\w**	**\s**
start with	end with	word characters	space
^www start with www	$be end with be	ab\\w match abd	ab\\sc match ab c but not abc

NLP tasks

Named Entity Recognition (NER):
Description: Identifying and classifying proper nouns into predefined categories like names of people, organizations, locations, dates.
Use Case: Information retrieval, question answering, content classification.

Text Classification:
Description: Categorizing text into predefined classes or categories.
Use Case:Spam detection, topic labeling, sentiment categorization.

Text Similarity:
Description:how similar two pieces of text are to each other.
Use Case:Search Engines, Recommendation Systems, job matching.

Information Extraction:
Description:Automatically extracting structured information from unstructured text.
Use Case:Extract patient information from doctors' notes, extract candidate information from resumes.

Question Answering:
Description:Building systems that can answer questions posed in natural language
Use Case:Virtual assistants, customer support bots, educational tools.

Machine Translation:
Description:Automatically translating text from one language to another.
Use Case:Language translation services like Google Translate, multilingual communication tools.

Language Modeling:
Description:Predicting the next word in a sequence or assessing the probability of a given sequence of words..
Use Case:Autocomplete features, speech recognition, text generation.

Text Summarization:
Description:reating a concise and coherent summary of a longer text document.
Use Case:document summarization for quick information retrieval.

Preprocessing

A:Removing Punctuation and Special Characters
Remove punctuation marks (e.g., ".", ",", "!") and special characters (e.g., "@", "#", "$") that do not contribute to the meaning of the text for many NLP tasks.

B:Tokenization
breaking down a piece of text into smaller units called tokens,There are two main types of tokenization, **Sentence Tokenization** (or sentence segmentation) and **Word Tokenization** (or lexical analysis)

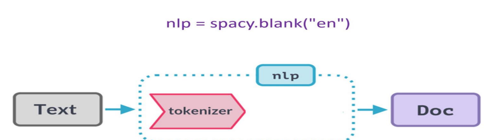
C:Removing Stop Words
Description: Stop words are common words like "the", "is", "in", "and" that do not carry significant meaning. They are often removed to reduce the noise in the text.

D:Stemming and Lemmatization
Stemming: Reduces words to their root by chopping off prefixes or suffixes.(Example: "Caring" → "Car")
Lemmatization: Converts words to their dictionary form, considering grammar and meaning.(Example: "Caring" → "Care")

Tokenization(spacy)

```
import spacy
nlp = spacy.blank("en")
doc = nlp("Dr. Strange loves pav bhaji of mumbai as it costs only 2$ per plate.")
for token in doc:
    print(token)
```

Creating blank language object gives a tokenizer and an empty pipeline



Token att

```
doc = nlp("Tony gave two $ to Peter.")
token0 = doc[0] #Tony
token0.is_alpha # True
token0.like_num # False
dir(token0) # see other attribute
```

Customizing tokenizer

```
from spacy.symbols import ORTH
nlp = spacy.blank("en")
doc = nlp("gimme double cheese extra large healthy pizza")
tokens = [token.text for token in doc]
tokens # ['gimme',  'large', 'pizza']
nlp.tokenizer.add_special_case("gimme", [{ORTH: "gim"}, {ORTH: "me"},])
doc = nlp("gimme double cheese extra large healthy pizza")
tokens = [token.text for token in doc]
```

Sentence Segmentation

```
for sent in doc.sents: #if error add nlp.add_pipe('sentencizer')
    print(sent.text)
```

Stemming in NLTK

```
from nltk.stem import PorterStemmer
stemmer = PorterStemmer()
words = ["eating", "eats", "eat", "ate", "adjustable",  "ability"]
for word in words:
   print(word, "|", stemmer.stem(word))#eating | eat
eats | eat / eat | eat / ate | ate / adjustable | adjust / ability | abil
```

use simple rules

Lemmatization in Spacy

```
nlp = spacy.load("en_core_web_sm")
doc = nlp("Mando talked for 3 hours although talking isn't his thing")
for token in doc:
   print(token, " | ", token.lemma_) # eating  |  eat / eats  |  eat / eat  |  eat / ate  |  eat /adjustable  |  adjustable / ability  |  ability
```

Customizing lemmatizer

```
ar = nlp.get_pipe('attribute_ruler')
ar.add([[{"TEXT":"Bro"}],[{"TEXT":"Brah"}]],{"LEMMA":"Brother"})
```

Part Of Speach in Spacy

POS tagging involves assigning grammatical categories to each token (word) in a text.like:

VERB (Verb): Describes an action, state, or occurrence.
Examples: "run", "is", "jumping"
ADJ (Adjective): Describes or modifies a noun.
Examples: "beautiful", "large", "red"
ADV (Adverb): Modifies verbs, adjectives, or other adverbs, often indicating manner, place, time, or degree.
Examples: "quickly", "very", "well"
AUX (Auxiliary Verb): Helps the main verb to express tense, aspect, mood, or voice.
Examples: "is", "have", "will"
NOUN (Noun): Names a person, place, thing, or idea.
Examples: "dog", "city", "happiness"
PRON (Pronoun): Replaces a noun or noun phrase.
Examples: "he", "they", "which"
PROPN (Proper Noun): Specific names of people, places, organizations, etc.

```
doc = nlp("He quits the job")
print(doc[1].text, "|", doc[1].tag_, "|", spacy.explain(doc[1].tag_))
#quits | VBZ | verb, 3rd person singular present
```

count each POS

```
count = doc.count_by(spacy.attrs.POS)
for k,v in count.items():
   print(doc.vocab[k].text, "|",v) # PROPN | 13
NOUN | 48
VERB | 23
```

NER in Spacy

```
doc = nlp("Tesla Inc is going to acquire twitter for $45 billion")
for ent in doc.ents:
   print(ent.text, " | ", ent.label_, " | ", spacy.explain(ent.label_))
#Tesla Inc  |  ORG  |  Companies, agencies, institutions, etc.
#$45 billion  |  MONEY  |  Monetary values, including unit
```

for looking out put better

```
from spacy import displacy
displacy.render(doc, style="ent")
```

Setting custom entities

```
from spacy.tokens import Span
s1 = Span(doc, 0, 1, label="ORG")
s2 = Span(doc, 5, 6, label="ORG")
doc.set_ents([s1, s2], default="unmodified")
```

Text Representation

Text representation is a fundamental concept in (NLP) that involves converting textual data into a format that machine learning models can understand.

Traditional Text Representation Methods

1,**one hot encoding:**Each word is assigned a unique binary vector where only one element is '1' (indicating the presence of that word).

	I	LOVE	LEARNING
I	1	0	0
LEARNING	0	0	1

Disadvantages: 1.Storing large one-hot vectors requires significant memory.
2.most of their elements are zeros.
3.One-hot encoding treats each word as an independent entity without capturing any semantic or syntactic relationships between words.
4.relies on a fixed vocabulary

2,Bag of Words (BoW): Compile a list of all unique words (vocabulary) from the entire corpus (collection of documents/texts).
For each document, create a vector where each element represents the frequency of a corresponding word from the vocabulary in that document.

	I	LOVE	LEARNING
I love learning ,just learning	1	1	2
I eat pizza	1	0	0

Bag of Words, aggregates the presence and count of words within a larger text unit.but,
One-Hot Encoding ,represents single words (or tokens) independently without considering their frequency or context within a document.both have same problems.

```python
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import CountVectorizer
clf = Pipeline([
    ('vectorizer', CountVectorizer()),
    ('nb', MultinomialNB())
])
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print(classification_report(y_test, y_pred))
```

3,Bag of n-grams: is an extension of the Bag of Words (BoW) model, a fundamental technique in (NLP) for text representation. While BoW focuses on individual words, Bag of n-grams considers contiguous sequences of words (or tokens), capturing more contextual information from the text. This approach enhances the model's ability to understand phrases and local word dependencies.

	I LOVE	LOVE PIZZA	LOVE DOG
i love pizza	1	1	0
i love dog	1	0	1

```python
from sklearn.pipeline import Pipeline
clf = Pipeline([
    ('vectorizer_1_2_gram', CountVectorizer(ngram_range = (1, 2))), #using the one token and two tokens
    ('Multi NB', MultinomialNB())
])
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print(classification_report(y_test, y_pred))
```

5,**Term Frequency-Inverse Document Frequency (TF-IDF)**:Helps in identifying words that are most relevant to a particular document.Filters out common but less informative words (like stop words), ,focusing on words that carry more meaning,Assigns weights to words based on their importance.

(TF) measures how frequently a term (word) appears in **a document**.

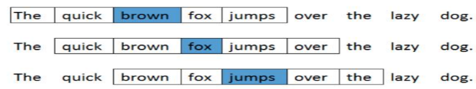
(IDF) terms appearing in many documents are less discriminative and thus less important.

```python
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
clf = Pipeline([
 ('vectorizer_tfidf',TfidfVectorizer()),
    ('KNN', KNeighborsClassifier())
])
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print(classification_report(y_test, y_pred))
```

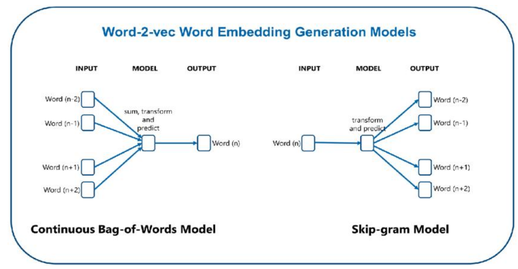
Word Embeddings Text Representation Methods

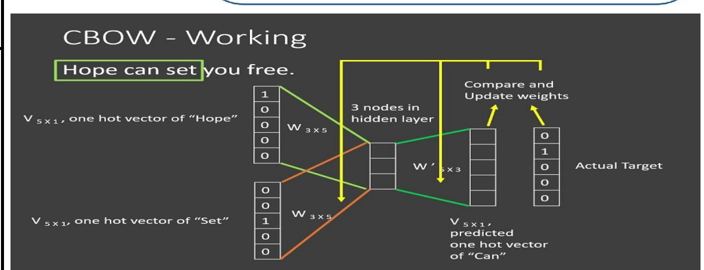
Word embeddings are dense, low-dimensional vector representations of words that capture semantic and syntactic relationships based on context. Unlike sparse representations like BoW and TF-IDF, embeddings encode words in continuous vector where similar words have similar vectors.

1,Word2Vec: Word2Vec primarily consists of two model architectures:
1.1:Continuous Bag of Words (CBOW)
1.2:Skip-gram
Both models use neural networks to learn word embeddings.
In the following image, the word in the blue box is called the target word and the words in the white boxes are called context words in a window of size 5 .



CBOW: The model is fed by the context , and **predicts the target** word. The result of the hidden layer is the new representation of the word ($h_1, \ldots, h_N$).
Skip Gram: The model is fed with the target word , and **predict**s words from the **context**. The result of the hidden layer is the new representation of the word ($h_1, \ldots, h_N$).






```python
import gensim
review_text = df.reviewText.apply(gensim.utils.simple_preprocess)
model = gensim.models.Word2Vec(
    window=10,
    min_count=2,
    workers=4,
)
model.build_vocab(review_text) #Build Vocabulary
model.train(review_text, total_examples=model.corpus_count,
epochs=model.epochs)
model.wv.most_similar("bad")#[('terrible', 0.6617082357406616),
 ('horrible', 0.6136840581893921),...]
model.wv.similarity(w1="cheap", w2="inexpensive") #0.734
```


Disadvantages: 1.global information not preserved (solve by Glove).

```python
import fasttext
model_en = fasttext.load_model('path.bin')
model_en.get_nearest_neighbors('good')
```

2,GloVe: Creates Count Word Co-occurrences,looks at a large collection of text and counts how often each word appears next to other words.
imagine a table where:Rows are words and Columns are words.Each cell shows how often the row word appears near the column word.This table helps GloVe understand the relationships between all pairs of words.

X_{ij} tabulate the number of times word j occurs in the context of word i.
$X_i = \sum_k X_{ik}$
$P_{ij} = P(j|i) = X_{ij}/X_i$

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Very small or large:
solid is related to ice but not steam, or
gas is related to steam but not ice

close to 1:
water is highly related to ice and steam, or
fashion is not related to ice or steam.

$w \in \mathbb{R}^d$ are word vectors   probe word

$$F(w_i, w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

co-relations between the word wi and wj co-occurrence probabilities for the word wj and wk

```
import gensim.downloader as api
# Load the pre-trained GloVe model on wiki-gigaword-100
model = api.load ("glove-wiki-gigaword-100")
# Find the vector for a word
vector_king = model['king']
# Find similar words
similar_to_king = model.most_similar('king')
for word, similarity in similar_to_king:
    print(f"{word}: {similarity:.4f}") # queen: 0.7117  monarch:
```

in spacy

```
import spacy
nlp = spacy.load("en_core_web_lg")
#This will take some time(nearly 15 minutes)
df['vector'] = df['Text'].apply(lambda text: nlp(text).vector)
```

2,FastText: It extends the popular Word2Vec model by incorporating **subword** information.Instead of representing each word as a single unit, FastText breaks words into smaller parts called **n-grams** (e.g., for the word "playing," with n=3, the n-grams are "pla," "lay," "ayi," "yin," "ing").it can handl Rare and OOV Words:By using subword information, FastText can generate vectors for words it hasn't seen during training by combining the vectors of their n-grams.Example: If "happiness" wasn't in the training data, FastText can still create its vector using the n-grams like "hap," "app," "ppi," "pin," "ine," "nes."
plus it can generate meaningful vectors even for misspelled words based on their n-grams.

```
model_en.get_nearest_neighbors("good")
model_en.get_analogies("berlin","germany","france") #paris
# Custom train word embeddings
custom_model = fasttext.train_unsupervised("ourtxt.txt")
custom_model.get_word_vector("dost")
#https://fasttext.cc/docs/en/unsupervised-tutorial.html for details on parameters in train_unsupervised function.
```

Transformer based Text Representation Methods