Group 1

Assignment No 1

Problem Statement:

Perform tokenization (Whitespace, Punctuation-based, Treebank, Tweet, MWE) using NLTK library. Use porter stemmer and snowball stemmer for stemming. Use any technique for lemmatization.

Input / Dataset –use any sample sentence

Objective:

To understand the fundamental concepts and techniques of natural language processing (NLP).

CO Relevance: CO1

Contents for Theory:

Introduction to NPL (Natural Language Processing)

Computers speak their own language, the binary language. Thus, they are limited in how they can interact with us humans; expanding their language and understanding our own is crucial to set them free from their boundaries.

NLP is an abbreviation for natural language processing, which encompasses a set of tools, routines, and techniques computers can use to process and understand human communications. Not to be confused with speech recognition, NLP deals with understanding the meaning of words other than interpreting audio signals into those words.

If you think NLP is just a futuristic idea, you may be shocked to know that we are likely to interact with NLP every day when we perform queries in Google when we use translators online when we talk with Google Assistant or Siri. NLP is everywhere, and to implement it in your projects is now very reachable thanks to libraries such as NLTK, which provide a huge abstraction of the complexity.

```
In [1]: pip install nltk
        Defaulting to user installation because normal site-packages is not writeable
        Requirement already satisfied: nltk in /home/dipali/.local/lib/python3.8/site-packages (3.8.1)
        Requirement already satisfied: joblib in /home/dipali/.local/lib/python3.8/site-packages (from nltk) (1.2.0)
        Requirement already satisfied: regex>=2021.8.3 in /home/dipali/.local/lib/python3.8/site-packages (from nltk) (2022.
        10.31)
        Requirement already satisfied: click in /home/dipali/.local/lib/python3.8/site-packages (from nltk) (8.1.3)
        Requirement already satisfied: tqdm in /home/dipali/.local/lib/python3.8/site-packages (from nltk) (4.64.1)

        [notice] A new release of pip is available: 23.0 -> 23.0.1
        [notice] To update, run: python3 -m pip install --upgrade pip
        Note: you may need to restart the kernel to use updated packages.
```

```
In [9]: nltk.download('punkt')
        [nltk_data] Downloading package punkt to /home/dipali/nltk_data...
        [nltk_data]   Unzipping tokenizers/punkt.zip.
Out[9]: True
```

1. Tokenization

Tokenization is the process of breaking text into smaller pieces called tokens. These smaller pieces can be sentences, words, or sub-words.

For example, the sentence “I won” can be tokenized into two word-tokens “I” and “won”.

Sentence Tokenization

In [4]:
```
import nltk
from nltk.tokenize import sent_tokenize
```

In [5]:
```
text=" India is a unique country with diversity. Unity is diversity is the main slogan of the country."
```

In [6]:
```
print(sent_tokenize(text))
```

```
[' India is a unique country with diversity.', 'Unity is diversity is the main slogan of the country.']
```

Word Tokenization

In [7]:
```
import nltk
from nltk.tokenize import word_tokenize
```

In [8]:
```
print(word_tokenize(text))
```

```
['India', 'is', 'a', 'unique', 'country', 'with', 'diversity', '.', 'Unity', 'is', 'diversity', 'is', 'the', 'main',
'slogan', 'of', 'the', 'country', '.']
```

A. Whitespace tokenization

A WhitespaceTokenizer is a tokenizer that splits on and discards only whitespace characters.

In [9]:
```
print(f'Whitespace tokenization = {text.split()}')
```

```
Whitespace tokenization = ['India', 'is', 'a', 'unique', 'country', 'with', 'diversity.', 'Unity', 'is', 'diversity
', 'is', 'the', 'main', 'slogan', 'of', 'the', 'country.']
```

B. Punctuation-based tokenization

Punctuation-based tokenization is slightly more advanced than whitespace-based tokenization since it splits on whitespace and punctuations and also retains the punctuations.

In [10]:
```
from nltk.tokenize import wordpunct_tokenize
```

In [11]:
```
print(f'Punctuation-based tokenization = {wordpunct_tokenize(text)}')
```

```
Punctuation-based tokenization = ['India', 'is', 'a', 'unique', 'country', 'with', 'diversity', '.', 'Unity', 'is',
'diversity', 'is', 'the', 'main', 'slogan', 'of', 'the', 'country', '.']
```

C. Default/TreebankWordTokenizer

The default tokenization method in NLTK involves tokenization using regular expressions as defined in the Penn Treebank (based on English text). It assumes that the text is already split into sentences.

In [12]:
```
from nltk.tokenize import TreebankWordTokenizer
```

In [13]:
```
sentence="What's your name?"
```

In [14]:
```
tokenizer = TreebankWordTokenizer()
print(f'Default/Treebank tokenization = {tokenizer.tokenize(sentence)}')
```

```
Default/Treebank tokenization = ['What', "'s", 'your', 'name', '?']
```

D.TweetTokenizer

When we want to apply tokenization in text data like tweets, the tokenizers mentioned above can't produce practical tokens. Through this issue, NLTK has a rule based tokenizer special for tweets. We can split emojis into different words if we need them for tasks like sentiment analysis.

a. Install emoji library

In [15]:
```
pip install emoji --upgrade
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: emoji in /home/dipali/.local/lib/python3.8/site-packages (2.2.0)

[notice] A new release of pip is available: 23.0 -> 23.0.1
[notice] To update, run: python3 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
```

In [16]:
```
import emoji
```

In [17]:
```
print(emoji.emojize('Hi Everyone! :grinning_face:'))
```

```
Hi Everyone! 😀
```

In [18]:
```
sentence1= emoji.emojize('Hi Everyone! :grinning_face:')
```

In [19]:
```
from nltk.tokenize import TweetTokenizer
```

In [20]:
```
tokenizer = TweetTokenizer()
print(f'Tweet-rules based tokenization = {tokenizer.tokenize(sentence1)}')
```

```
Tweet-rules based tokenization = ['Hi', 'Everyone', '!', '😀']
```

E. MWETokenizer

NLTK's multi-word expression tokenizer (MWETokenizer) provides a function add_mwe() that allows the user to enter multiple word expressions before using the tokenizer on the text. More simply, it can merge multi-word expressions into single tokens.

In [21]:
```
sentence2="Hope, is the only thing stronger than fear! Hunger Games"
```

In [22]:
```
print(word_tokenize(sentence2))
```

```
['Hope', ',', 'is', 'the', 'only', 'thing', 'stronger', 'than', 'fear', '!', 'Hunger', 'Games']
```

In [23]:
```
from nltk.tokenize import MWETokenizer
```

In [24]:
```
tokenizer = MWETokenizer()
tokenizer.add_mwe(('Hunger', 'Games'))
print(f'Multi-word expression (MWE) tokenization = {tokenizer.tokenize(word_tokenize(sentence2))}')
```

```
Multi-word expression (MWE) tokenization = ['Hope', ',', 'is', 'the', 'only', 'thing', 'stronger', 'than', 'fear',
'!', 'Hunger_Games']
```

2. Stemming and Lemmatization

A. Stemming

Stemming is the process of producing morphological variants of a root/base word. Stemming programs are commonly referred to as stemming algorithms or stemmers. Often when searching text for a certain keyword, it helps if the search returns variations of the word. For instance, searching for "boat" might also return "boats" and "boating". Here, "boat" would be the stem for [boat, boater, boating, boats]. Stemming is a somewhat crude method for cataloging related words; it essentially chops off letters from the end until the stem is reached.

i) Porter Stemmer

In [25]:
```
# Import the toolkit and the full Porter Stemmer library
import nltk

from nltk.stem.porter import *
p_stemmer = PorterStemmer()
words = ['run','runner','running','ran','runs','easily','fairly']
for word in words:
    print(word+' --> '+p_stemmer.stem(word))
```

```
run --> run
runner --> runner
running --> run
ran --> ran
runs --> run
easily --> easili
fairly --> fairli
```

ii) Snowball Stemmer

In [26]:
```
from nltk.stem.snowball import SnowballStemmer

# The Snowball Stemmer requires that you pass a language parameter
s_stemmer = SnowballStemmer(language='english')
words = ['run','runner','running','ran','runs','easily','fairly']
for word in words:
    print(word+' --> '+s_stemmer.stem(word))
```

```
run --> run
runner --> runner
running --> run
ran --> ran
runs --> run
easily --> easili
fairly --> fair
```

B. Lemmatization

In contrast to stemming, lemmatization looks beyond word reduction and considers a language's full vocabulary to apply a morphological analysis to words. The lemma of 'was' is 'be' and the lemma of 'mice' is 'mouse'.

Lemmatization is typically seen as much more informative than simple stemming, which is why Spacy has opted to only have Lemmatization available instead of Stemming

Install Spacy

In [27]:
```
!pip3 install spacy
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: spacy in /home/dipali/.local/lib/python3.8/site-packages (3.5.0)
Requirement already satisfied: thinc<8.2.0,>=8.1.0 in /home/dipali/.local/lib/python3.8/site-packages (from spacy)
(8.1.7)
Requirement already satisfied: catalogue<2.1.0,>=2.0.6 in /home/dipali/.local/lib/python3.8/site-packages (from spac
y) (2.0.8)
Requirement already satisfied: tqdm<5.0.0,>=4.38.0 in /home/dipali/.local/lib/python3.8/site-packages (from spacy)
(4.64.1)
Requirement already satisfied: typer<0.8.0,>=0.3.0 in /home/dipali/.local/lib/python3.8/site-packages (from spacy)
(0.7.0)
Requirement already satisfied: setuptools in /home/dipali/.local/lib/python3.8/site-packages (from spacy) (67.0.0)
Requirement already satisfied: preshed<3.1.0,>=3.0.2 in /home/dipali/.local/lib/python3.8/site-packages (from spacy)
```

Download the model for English Langauge

In [28]:
```
!python3 -m spacy download en_core_web_sm
```

```
Defaulting to user installation because normal site-packages is not writeable
Collecting en-core-web-sm==3.5.0
  Downloading https://github.com/explosion/spacy-models/releases/download/en_core_web_sm-3.5.0/en_core_web_sm-3.5.0-
py3-none-any.whl (12.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 12.8/12.8 MB 1.5 MB/s eta 0:00:00m eta 0:00:01[36m0:00:01m
Requirement already satisfied: spacy<3.6.0,>=3.5.0 in /home/dipali/.local/lib/python3.8/site-packages (from en-core-
web-sm==3.5.0) (3.5.0)
Requirement already satisfied: setuptools in /home/dipali/.local/lib/python3.8/site-packages (from spacy<3.6.0,>=3.
5.0->en-core-web-sm==3.5.0) (67.0.0)
Requirement already satisfied: catalogue<2.1.0,>=2.0.6 in /home/dipali/.local/lib/python3.8/site-packages (from spac
y<3.6.0,>=3.5.0->en-core-web-sm==3.5.0) (2.0.8)
Requirement already satisfied: cymem<2.1.0,>=2.0.2 in /home/dipali/.local/lib/python3.8/site-packages (from spacy<3.
6.0,>=3.5.0->en-core-web-sm==3.5.0) (2.0.7)
Requirement already satisfied: numpy>=1.15.0 in /home/dipali/.local/lib/python3.8/site-packages (from spacy<3.6.0,>=
3.5.0->en-core-web-sm==3.5.0) (1.24.1)
Requirement already satisfied: pydantic!=1.8,!=1.8.1,<1.11.0,>=1.7.4 in /home/dipali/.local/lib/python3.8/site-packa
ges (from spacy<3.6.0,>=3.5.0->en-core-web-sm==3.5.0) (1.10.4)
Requirement already satisfied: murmurhash<1.1.0,>=0.28.0 in /home/dipali/.local/lib/python3.8/site-packages (from sp
acy<3.6.0,>=3.5.0->en-core-web-sm==3.5.0) (1.0.9)
Requirement already satisfied: tqdm<5.0.0,>=4.38.0 in /home/dipali/.local/lib/python3.8/site-packages (from spacy<3.
6.0,>=3.5.0->en-core-web-sm==3.5.0) (4.64.1)
```

In [29]:
```
#Perform standard imports:
import spacy
# Load English tokenizer, tagger, parser and NER
nlp = spacy.load('en_core_web_sm')
def show_lemmas(text):
    for token in text:
        print(f'{token.text:{12}} {token.pos_:{6}} {token.lemma:<{22}} {token.lemma_}')
```

In [30]:
```
doc = nlp(u"I am a runner running in a race because I love to run since I ran today.")

show_lemmas(doc)
```

```
I            PRON   4690420944186131903    I
am           AUX    10382539506755952630   be
a            DET    11901859001352538922   a
runner       NOUN   12640964157389618806   runner
running      VERB   12767647472892411841   run
in           ADP    3002984154512732771    in
a            DET    11901859001352538922   a
race         NOUN   8048469955494714898    race
because      SCONJ  16950148841647037698   because
I            PRON   4690420944186131903    I
love         VERB   3702023516439754181    love
to           PART   3791531372978436496    to
run          VERB   12767647472892411841   run
since        SCONJ  10066841407251338481   since
I            PRON   4690420944186131903    I
ran          VERB   12767647472892411841   run
today        NOUN   11042482332948150395   today
.            PUNCT  12646065887601541794   .
```

Conclusion- In this way we have performed tokenization using NLTK. And porter and snowball stemming. Using SpaCy library performed lemmatization.

Viva Questions

1. What is difference between porter and snowball stemmer?
2. What is lemmatization?
3. Differentiate between lemmatization and stemming.
4. What are different python libraries used for lemmatization.
5. Why do we need tokenization?

Date:	
Marks obtained:	
Sign of course coordinator:	
Name of course Coordinator:	

Group 1

Assignment No:2

Title of the Assignment:

Perform bag-of-words approach (count occurrence, normalized count occurrence), TF-IDF on data. Create embeddings using Word2Vec.
Dataset to be used: https://www.kaggle.com/datasets/CooperUnion/cardataset

Objective of the Assignment: To understand the fundamental concepts and techniques of natural language processing (NLP).

CO Relevance: CO1

Theory:

Word Embedding

It is an approach for representing words and documents. Word Embedding or Word Vector is a numeric vector input that represents a word in a lower-dimensional space. It allows words with similar meaning to have a similar representation. They can also approximate meaning. Word Embeddings are the texts converted into numbers and there may be different numerical representations of the same text.

In [1]:
```
pip install pandas
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: pandas in /home/dipali/.local/lib/python3.8/site-packages (1.5.3)
Requirement already satisfied: pytz>=2020.1 in /home/dipali/.local/lib/python3.8/site-packages (from pandas) (2022.
7.1)
Requirement already satisfied: python-dateutil>=2.8.1 in /home/dipali/.local/lib/python3.8/site-packages (from panda
s) (2.8.2)
Requirement already satisfied: numpy>=1.20.3 in /home/dipali/.local/lib/python3.8/site-packages (from pandas) (1.24.
1)
Requirement already satisfied: six>=1.5 in /usr/lib/python3/dist-packages (from python-dateutil>=2.8.1->pandas) (1.1
4.0)

[notice] A new release of pip is available: 23.0 -> 23.0.1
[notice] To update, run: python3 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
```

In [2]:
```
pip install sklearn
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: sklearn in /home/dipali/.local/lib/python3.8/site-packages (0.0.post1)

[notice] A new release of pip is available: 23.0 -> 23.0.1
[notice] To update, run: python3 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
```

In [3]:
```
pip install scikit-learn
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: scikit-learn in /home/dipali/.local/lib/python3.8/site-packages (1.2.1)
Requirement already satisfied: joblib>=1.1.1 in /home/dipali/.local/lib/python3.8/site-packages (from scikit-learn)
(1.2.0)
Requirement already satisfied: scipy>=1.3.2 in /home/dipali/.local/lib/python3.8/site-packages (from scikit-learn)
(1.10.1)
Requirement already satisfied: threadpoolctl>=2.0.0 in /home/dipali/.local/lib/python3.8/site-packages (from scikit-
learn) (3.1.0)
Requirement already satisfied: numpy>=1.17.3 in /home/dipali/.local/lib/python3.8/site-packages (from scikit-learn)
(1.24.1)

[notice] A new release of pip is available: 23.0 -> 23.0.1
[notice] To update, run: python3 -m pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
```

Bag of words(BOW)

Bag of words is a simple and popular technique for feature extraction from text. Bag of word model processes the text to find how many times each word appeared in the sentence. This is also called as vectorization.

Steps for creating BOW

1. Tokenize the text into sentences
2. Tokenize sentences into words
3. Remove punctuation or stop words
4. Convert the words to lower text
5. Create the frequency distribution of words

In the code below, use CountVectorizer, it tokenizes a collection of text documents, builds a vocabulary of known words, and encodes new documents using that vocabulary.

In [11]:
```
#Creating frequency distribution of words using nltk
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize
from sklearn.feature_extraction.text import CountVectorizer
text="""Achievers are not afraid of Challenges, rather they relish them,
        thrive in them, use them. Challenges makes is stronger.
        Challenges makes us uncomfortable. If you get comfortable with uncomfort then you will grow.
        Challenge the challenge. """
#Tokenize the sentences from the text corpus
tokenized_text=sent_tokenize(text)#using CountVectorizer and removing stopwords in english language
cv1= CountVectorizer(lowercase=True,stop_words='english')#fitting the tonized senetnecs to the countvectorizer
text_counts=cv1.fit_transform(tokenized_text)# printing the vocabulary and the frequency distribution pf vocabulary i
print(cv1.vocabulary_)
print(text_counts.toarray())
```

```
{'achievers': 0, 'afraid': 1, 'challenges': 3, 'relish': 7, 'thrive': 9, 'use': 12, 'makes': 6, 'stronger': 8, 'unco
mfortable': 11, 'comfortable': 4, 'uncomfort': 10, 'grow': 5, 'challenge': 2}
[[1 1 0 1 0 0 0 1 0 1 0 0 1]
 [0 0 0 1 0 0 1 0 1 0 0 0 0]
 [0 0 0 1 0 0 1 0 0 0 0 1 0]
 [0 0 0 0 1 1 0 0 0 0 1 0 0]
 [0 0 2 0 0 0 0 0 0 0 0 0 0]]
```

Count Occurrence

Counting word occurrence. The reason behind of using this approach is that keyword or important signal will occur again and again. So if the number of occurrence represent the importance of word. More frequency means more importance.

In [12]:
```
import collections
import pandas as pd
import numpy as np


from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
```

In [13]:
```
doc = "India is my country. India is very beautiful country."

count_vec = CountVectorizer()
count_occurs = count_vec.fit_transform([doc])
count_occur_df = pd.DataFrame((count, word) for word, count in zip(count_occurs.toarray().tolist()[0],
                 count_vec. get_feature_names_out()))
count_occur_df.columns = ['Word', 'Count']
count_occur_df.sort_values('Count', ascending=False)
count_occur_df.head()
```

Out[13]:

	Word	Count
0	beautiful	1
1	country	2
2	india	2
3	is	2
4	my	1

Normalized Count Occurrence

If you think that high frequency may dominate the result and causing model bias. Normalization can be apply to pipeline easily.

In [14]:
```
doc = "India is my country. India is very beautiful country."

norm_count_vec = TfidfVectorizer(use_idf=False, norm='l2')
norm_count_occurs = norm_count_vec.fit_transform([doc])
norm_count_occur_df = pd.DataFrame((count, word) for word, count in zip(
    norm_count_occurs.toarray().tolist()[0], norm_count_vec. get_feature_names_out()))
norm_count_occur_df.columns = ['Word', 'Count']
norm_count_occur_df.sort_values('Count', ascending=False, inplace=True)
norm_count_occur_df.head()
```

Out[14]:

	Word	Count
1	country	0.516398
2	india	0.516398
3	is	0.516398
0	beautiful	0.258199
4	my	0.258199

TF-IDF

TF-IDF take another approach which is believe that high frequency may not able to provide much information gain. In another word, rare words contribute more weights to the model.

Word importance will be increased if the number of occurrence within same document (i.e. training record). On the other hand, it will be decreased if it occurs in corpus (i.e. other training records).

In [15]:
```
doc = "India is my country. India is very beautiful country."
tfidf_vec = TfidfVectorizer()
tfidf_count_occurs = tfidf_vec.fit_transform([doc])
tfidf_count_occur_df = pd.DataFrame((count, word) for word, count in zip(
    tfidf_count_occurs.toarray().tolist()[0], tfidf_vec. get_feature_names_out()))
tfidf_count_occur_df.columns = ['Word', 'Count']
tfidf_count_occur_df.sort_values('Count', ascending=True, inplace=True)
tfidf_count_occur_df.head()
```

Out[15]:

	Word	Count
0	beautiful	0.258199
4	my	0.258199
5	very	0.258199
1	country	0.516398
2	india	0.516398

Introduction to Word2Vec

Word2vec is one of the most popular technique to learn word embeddings using a two-layer neural network. Its input is a text corpus and its output is a set of vectors. Word embedding via word2vec can make natural language computer-readable, then further implementation of mathematical operations on words can be used to detect their similarities. A well-trained set of word vectors will place similar words close to each other in that space. For instance, the words women, men, and human might cluster in one corner, while yellow, red and blue cluster together in another.

There are two main training algorithms for word2vec, one is the continuous bag of words(CBOW), another is called skip-gram. The major difference between these two methods is that CBOW is using context to predict a target word while skip-gram is using a word to predict a target context. Generally, the skip-gram method can have a better performance compared with CBOW method, for it can capture two semantics for a single word. For instance, it will have two vector representations for Apple, one for the company and another for the fruit.

Gensim Python Library Introduction

Gensim is an open source python library for natural language processing and it was developed and is maintained by the Czech natural language processing researcher Radim Řehůřek. Gensim library will enable us to develop word embeddings by training our own word2vec models on a custom corpus either with CBOW of skip-grams algorithms.

In [1]:
```
!pip install --upgrade gensim
```

```
Defaulting to user installation because normal site-packages is not writeable
Collecting gensim
  Downloading gensim-4.3.0-cp38-cp38-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (24.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.1/24.1 MB 586.1 kB/s eta 0:00:00m eta 0:00:01[36m0:00:02
Collecting FuzzyTM>=0.4.0
  Downloading FuzzyTM-2.0.5-py3-none-any.whl (29 kB)
Requirement already satisfied: numpy>=1.18.5 in /home/dipali/.local/lib/python3.8/site-packages (from gensim) (1.24.
1)
Requirement already satisfied: smart-open>=1.8.1 in /home/dipali/.local/lib/python3.8/site-packages (from gensim)
(6.3.0)
Requirement already satisfied: scipy>=1.7.0 in /home/dipali/.local/lib/python3.8/site-packages (from gensim) (1.10.
1)
```

Download the data

Dataset Description

This vehicle dataset includes features such as make, model, year, engine, and other properties of the car. We will use these features to generate the word embeddings for each make model and then compare the similarities between different make model.

In [2]:
```
!wget https://raw.githubusercontent.com/PICT-NLP/BE-NLP-Elective/main/2-Embeddings/data.csv
```

```
--2023-03-01 21:22:40--  https://raw.githubusercontent.com/PICT-NLP/BE-NLP-Elective/main/2-Embeddings/data.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.111.133, 185.199.108.133, 185.199.109.13
3, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.111.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1475504 (1.4M) [text/plain]
Saving to: 'data.csv.1'

data.csv.1          100%[===================>]   1.41M   587KB/s    in 2.5s

2023-03-01 21:22:43 (587 KB/s) - 'data.csv.1' saved [1475504/1475504]
```

Implementation of Word Embedding with Gensim

In [3]:
```
import pandas as pd
```

In [4]:
```
df = pd.read_csv('data.csv')
df.head()
```

Out[4]:

	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors	Market Category	Vehicle Size	Vehicle Style	highway MPG	city mpg	Popularity
0	BMW	1 Series M	2011	premium unleaded (required)	335.0	6.0	MANUAL	rear wheel drive	2.0	Factory Tuner,Luxury,High-Performance	Compact	Coupe	26	19	3916
1	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,Performance	Compact	Convertible	28	19	3916
2	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,High-Performance	Compact	Coupe	28	20	3916
3	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Luxury,Performance	Compact	Coupe	28	18	3916
4	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Luxury	Compact	Convertible	28	18	3916

Data Preprocessing

Since the purpose of this tutorial is to learn how to generate word embeddings using genism library, we will not do the EDA and feature selection for the word2vec model for the sake of simplicity.

Genism word2vec requires that a format of 'list of lists' for training where every document is contained in a list and every list contains lists of tokens of that document. At first, we need to generate a format of 'list of lists' for training the make model word embedding. To be more specific, each make model is contained in a list and every list contains lists of features of that make model.

To achieve this, we need to do the following things

Create a new column for Make Model

In [5]:
```
df['Maker_Model']= df['Make']+ " " + df['Model']
```

Generate a format of ' list of lists' for each Make Model with the following features: Engine Fuel Type, Transmission Type, Driven_Wheels, Market Category, Vehicle Size, Vehicle Style.

In [6]:
```
df1 = df[['Engine Fuel Type','Transmission Type','Driven_Wheels','Market Category','Vehicle Size', 'Vehicle Style', '

df2 = df1.apply(lambda x: ','.join(x.astype(str)), axis=1)

df_clean = pd.DataFrame({'clean': df2})

sent = [row.split(',') for row in df_clean['clean']]
```

In [36]:
```
df_clean
```

Out[36]:

	clean
0	premium unleaded (required),MANUAL,rear wheel ...
1	premium unleaded (required),MANUAL,rear wheel ...
2	premium unleaded (required),MANUAL,rear wheel ...
3	premium unleaded (required),MANUAL,rear wheel ...
4	premium unleaded (required),MANUAL,rear wheel ...
...	...
11909	premium unleaded (required),AUTOMATIC,all whee...
11910	premium unleaded (required),AUTOMATIC,all whee...
11911	premium unleaded (required),AUTOMATIC,all whee...
11912	premium unleaded (recommended),AUTOMATIC,all w...
11913	regular unleaded,AUTOMATIC,front wheel drive,L...

11914 rows × 1 columns

Genism word2vec Model Training

We can train the genism word2vec model with our own custom corpus as following:

In [13]:
```
from gensim.models.word2vec import Word2Vec
```

Let's try to understand the hyperparameters of this model.

1. `vector_size` : The number of dimensions of the embeddings and the default is 100.
2. `window` : The maximum distance between a target word and words around the target word. The default window is 5.
3. `min_count` : The minimum count of words to consider when training the model; words with occurrence less than this count will be ignored. The default for min_count is 5.
4. `workers` : The number of partitions during training and the default workers is 3.
5. `sg` : The training algorithm, either CBOW(0) or skip gram(1). The default training algorithm is CBOW.

After training the word2vec model, we can obtain the word embedding directly from the training model as following.

In [29]:
```
model = Word2Vec(sent, min_count=1,vector_size= 50,workers=3, window =3, sg= 1)
```

Save the model

In [30]:
```
model.save("word2vec.model")
```

Load the model

In [31]:
```
model = Word2Vec.load("word2vec.model")
```

After training the word2vec model, we can obtain the word embedding directly from the training model as following.

In [32]:
```
model.wv['Toyota Camry']
```

Out[32]:
```
array([ 0.01411632,  0.14784585,  0.01885239, -0.10753247, -0.07146065,
       -0.22338827,  0.01374025,  0.30203253, -0.09214514, -0.08290682,
        0.04862676,  0.03026489,  0.11046173, -0.02939197, -0.03781607,
        0.17612398,  0.15454124,  0.29968512, -0.13931143, -0.29023492,
       -0.05522037, -0.0564889 ,  0.25777268,  0.07147302,  0.17070875,
        0.00251983, -0.03870121,  0.393018  , -0.04162066, -0.00246239,
        0.01573551,  0.02139783,  0.03586031,  0.00898253,  0.085445  ,
       -0.11928873,  0.1799241 , -0.02834527,  0.04921703,  0.08003329,
        0.10614782, -0.03156348, -0.22044587,  0.09316084,  0.37852806,
        0.0510865 , -0.0260468 , -0.15805483,  0.00056193,  0.01605724],
      dtype=float32)
```

In [33]:
```
sims = model.wv.most_similar('Toyota Camry', topn=10)
sims
```

Out[33]:
```
[('Chevrolet Malibu', 0.9873985648155212),
 ('Nissan Sentra', 0.9869186878204346),
 ('Mazda 6', 0.9854127764701843),
 ('Buick Verano', 0.9837399125099182),
 ('Toyota Avalon', 0.9836648106575012),
 ('Nissan Altima', 0.9834702610969543),
 ('Pontiac Grand Am', 0.9833094477653503),
 ('Chevrolet Cruze', 0.9826680421829224),
 ('Suzuki Verona', 0.9806841611862183),
 ('Suzuki Kizashi', 0.9794840812683105)]
```

Calculate similarity between two words

```
In [34]: model.wv.similarity('Toyota Camry','Mazda 6')
Out[34]: 0.98541266

In [35]: model.wv.similarity('Dodge Dart','Mazda 6')
Out[35]: 0.97065187
```

Conclusion: In this way we have implement Bag-of-Word (BOW), Tf-Idf appoarch by using python library. And word2vec model implemented successfully by using genism library.

Viva Questions:

1. What is Tf-Idf?
2. Differentiate between continuous-bags-of-words and skip-gram.
3. What is mean by Word Embedding? And what are techniques of word embedding?
4. Why word embedding is required?
5. Which library used in word embedding?

.

Date:	
Marks obtained:	
Sign of course coordinator:	
Name of course Coordinator:	