UNIVERSITY INSTITUTE OF ENGINEERING AND TECHNOLOGY PANJAB UNIVERSITY, SECTOR 25, CHANDIGARH

LAB FILE

CSE
DEPARTMENT
BATCH: 2023-27

SUBMITTED BY:
Tushya Gupta
UE233106
CSE Section 2
Group 6

SUPERVISED BY:
Amritpal Ma'am

Index

Lab 1

Q1. To implement basic text processing operations like Tokenization, Normalization, Stemming, Lemmatization, Stop words removal, Sentence Segmentation etc. on text document.

Text processing in NLP – Text processing in NLP is the process of converting the corpus into a format that can be understood by a computer. It involves the following steps –

1. Tokenization – Tokenization is the process of breaking down the corpus into smaller chunks from paragraphs to sentences, then from sentences to words which are called tokens.
2. Case normalization – This is the process of converting the whole corpus into lower to reduce the number of unique cases due to capitalization of words like 'the' and 'The'.
3. Stemming – Stemming is the process of finding out the root/stem word from a given word by removing all the affixes out of them. This can create a root word which does or does not hold a meaning the in the dictionary.
4. Lemmatization – This is the step to fix the issue in the stemming where root words don't hold a dictionary meaning. In Lemmatization we find out the root words which do hold a dictionary meaning.
5. Stop word removal – This is the removal of words that are not required for use in the NLP process and will slow it down like articles and punctuation.
6. POS tagging – This is tagging the words according to the grammar rules of the language like nouns, adverbs, adjectives, etc.
7. Noun chunk phrases – These are phrases that hold a value when the words are kept together like in 'artificial intelligence' they together mean AI but have a different meaning separately.
8. Dependency parsing – Finding out the relation between words and creating a dependency tree from them which shows how the words are dependent on one another
9. Named entity recognition – Named entities are real word entities which land in a pre-defined category like a person or an organization. This step involves finding out such entities

Text Preprocessing

August 26, 2025

1 To implement basic text processing operations like Tokenization, Normalization, Stemming, Lemmatization, Stop words removal, Sentence Segmentation etc. on text document

[10]:
```
import nltk
import spacy
from nltk.stem import PorterStemmer
from rich import print

nlp = spacy.load("en_core_web_sm")
nltk.download("punkt")
stemmer = PorterStemmer()
```

```
[nltk_data] Downloading package punkt to /Users/tushya/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
```

1.1 Tokenization

Tokenization is the breaking down of large input into smaller and smaller chunks called tokens. These tokens usually represent the words in a sentence. In Tokenization we first break down the Paragrapgh -> Sentences and the Sentences -> Words where these words represent a token

[11]:
```
text = "The rapid advancement of technology has transformed the way people␣
 ↪communicate, work, and learn. Social media platforms allow individuals to␣
 ↪share ideas instantly across the globe, but they also raise concerns about␣
 ↪privacy and misinformation. At the same time, businesses rely on data-driven␣
 ↪decision-making to improve efficiency and customer satisfaction. Education␣
 ↪has also shifted significantly, with online learning tools providing access␣
 ↪to knowledge for millions of students worldwide. Despite these benefits,␣
 ↪challenges such as digital inequality and cybersecurity threats continue to␣
 ↪grow, making it important to balance innovation with responsibility."
doc = nlp(text)
tokens = [token.text for token in doc]
print(tokens[:10])
```

```
['The', 'rapid', 'advancement', 'of', 'technology', 'has', 'transformed', 'the',␣
 ↪'way', 'people']
```

1.2 Case Normalization

Transforming the case of the document into lowercase for easier processing and lowered complexity

[12]:
```
doc = nlp(text.lower())
```

1.3 Stemming

Stemming is the process to procuring the stem/root word from a given token by removing all of its affixs. The stem produced might or might not be a word in the dictionary

[13]:
```
stems = [(token, stemmer.stem(token)) for token in tokens]
print(stems[:10])
```

```
[
    ('The', 'the'),
    ('rapid', 'rapid'),
    ('advancement', 'advanc'),
    ('of', 'of'),
    ('technology', 'technolog'),
    ('has', 'ha'),
    ('transformed', 'transform'),
    ('the', 'the'),
    ('way', 'way'),
    ('people', 'peopl')
]
```

1.4 Lemmatization

Lemmatization is the process of converting the stem words that do not have a meaning in the dictionary to acctual root words that hold a meaning in the dictionary

[14]:
```
lemmas = [(token, token.lemma_) for token in doc]
print(lemmas[:10])
```

```
[
    (the, 'the'),
    (rapid, 'rapid'),
    (advancement, 'advancement'),
    (of, 'of'),
    (technology, 'technology'),
    (has, 'have'),
    (transformed, 'transform'),
    (the, 'the'),
    (way, 'way'),
    (people, 'people')
]
```

1.5 Stop word removal

A lot of our input consists of stop words or words that dont hold much value like articles and punctuations. To increase our processing speed we remove these useless tokens and try to reduce the size of information our model has to process

[15]:
```
filtered_tokens = [
    token
    for token in doc
    if token.is_alpha and not token.is_stop and not token.is_punct
]
print(filtered_tokens[:10])
```

```
[rapid, advancement, technology, transformed, way, people, communicate, work,␣
↪learn, social]
```

1.6 POS Tagging

POS or Parts of Speech Tagging is to tag each of the token according to the type they are for example noun, adverb, adjective etc according to the rules of grammar

[16]:
```
pos_tags = [(token.text, token.pos_, token.tag_) for token in filtered_tokens]
print(pos_tags[:10])
```

```
[
    ('rapid', 'ADJ', 'JJ'),
    ('advancement', 'NOUN', 'NN'),
    ('technology', 'NOUN', 'NN'),
    ('transformed', 'VERB', 'VBN'),
    ('way', 'NOUN', 'NN'),
    ('people', 'NOUN', 'NNS'),
    ('communicate', 'VERB', 'VBP'),
    ('work', 'NOUN', 'NN'),
    ('learn', 'VERB', 'VB'),
    ('social', 'ADJ', 'JJ')
]
```

1.7 Noun Chunk Phrases

Noun chunks are a series of words that make up a phrase that are extracted from the text. For eg 'social media platforms', etc

[17]:
```
noun_chunks = [chunk for chunk in doc.noun_chunks]
print(noun_chunks[:10])
```

```
[
    the rapid advancement,
    technology,
    the way,
    people,
    social media platforms,
    individuals,
    ideas,
    the globe,
    they,
    concerns
]
```

1.8 Dependency parsing

This is the process of analyzing the grammatical structure of the text, i.e. how the words are related to each other in the text

[18]:
```
deps = [(token.text, token.dep_, token.head.text) for token in filtered_tokens]
print(deps[:10])
```

```
[
    ('rapid', 'amod', 'advancement'),
    ('advancement', 'nsubj', 'transformed'),
    ('technology', 'pobj', 'of'),
    ('transformed', 'ROOT', 'transformed'),
    ('way', 'dobj', 'transformed'),
    ('people', 'nsubj', 'communicate'),
    ('communicate', 'relcl', 'way'),
    ('work', 'conj', 'communicate'),
    ('learn', 'conj', 'work'),
    ('social', 'amod', 'media')
]
```

1.9 Named Entities Recognition

Named entities are real word entities which land in a pre-defined catagory like a person or a organization

[19]:
```
entities = [(ent.text, ent.label_) for ent in doc.ents]
print(entities[:10])
```

```
[('millions', 'CARDINAL')]
```

Lab 2

Q2. To implement N-gram Language Model.

N-gram Model - The N-gram model is a probabilistic model to find out the probability of a word occurring after n-1 given words. This only takes into account a limited amount of context and is this computationally viable.
N-gram models are mainly of 3 types –

1. Unigram – Here we find out the probability of a word without information of any surrounding words. Each word is taken to be independent of one another and the probability is based solely on its frequency in the corpous
$$P(w_i) = \frac{count(w_i)}{N}$$
Where N is the total number of words in the corpus
2. Bigram – Here we find out the probability of a word occurring right after a given word and thus can capture some dependencies like 'artificial' → 'intelligence'. It is calculated as follows
$$P(w_i|w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})}$$
Where $count(w_{i-1}, w_i)$ means the number of times w_i occurs after w_{i-1}.
3. Trigram – This is similar to the bigram model but we take into account 2 words before the one we are trying to calculate the probability for and thus can capture more complex dependencies than a bigram model.
$$P(w_i|w_{i-2}, w_{i-1}) = \frac{count(w_{i-2}, w_{i-1}, w_i)}{count(w_{i-2}, w_{i-1})}$$

Limitation of N-Gram Modeling –

- Data sparsity – Some words or sequence of words may not appear in the corpus
  - Solution → Smoothing techniques
- Higher Gram models require way more compute and also significantly increase the issue of data sparsity.

N-Gram Modeling

August 26, 2025

1 To implement N-gram Language Model.

The N-gram model is a probabilistic model to find out the probability of a word occouring after n-1 given words

[1]:
```
import nltk
from nltk import trigrams
from nltk.corpus import reuters
from collections import defaultdict
import random

nltk.download("punkt")
nltk.download("punkt_tab")
nltk.download("reuters")
```

```
[nltk_data] Downloading package punkt to /Users/tushya/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package punkt_tab to
[nltk_data]     /Users/tushya/nltk_data…
[nltk_data]   Package punkt_tab is already up-to-date!
[nltk_data] Downloading package reuters to /Users/tushya/nltk_data…
[nltk_data]   Package reuters is already up-to-date!
```

[1]:
```
True
```

Taking corpous from nltk library itself

[2]:
```
text = nltk.word_tokenize(" ".join(reuters.words()))
tri = list(trigrams(text))
```

We create a nested dict for our trigram model where (w1,w2) gives use a dict of next words and their possible probabilities

[3]:
```
model = defaultdict(lambda: defaultdict(lambda: 0))
# Getting the raw count
for w1, w2, w3 in tri:
    model[(w1, w2)][w3] += 1

# Converting the count into probability
```

```
for bi in model:
    c = float(sum(model[bi].values()))
    for w3 in model[bi]:
        model[bi][w3] /= c
```

Getting the next most probable word from the model. If we dont have a seen pattern for (w1,w2) we return a random word

```
[4]: def get_next_word(w1, w2):
         next_word_probs = model[(w1, w2)]
         if next_word_probs:
             return max(next_word_probs, key=next_word_probs.get)
         else:
             return random.choice(text)
```

Creating a sentence of length 12 by choosing 2 random words at the start and then creating a sentence out of them

```
[ ]: w1 = random.choice(text)
     w2 = random.choice(text)
     print(f"{w1} {w2} ", end="")
     for i in range(10):
         w3 = get_next_word(w1, w2)
         w1 = w2
         w2 = w3
         print(f"{w3} ", end="")
```

```
, place any restrictions on the sale of its common stock .
```

Lab 3

Q3. Write a program to extract features – TF, TF-IDF score from text.

TF-IDF – Term Frequency – Inverse Document Frequency is a statistical model used in NLP to evaluate how important a word is to a document.
It helps to lower the importance of frequently used words like I, am, the, etc. from the document and give more value to words which occur rarely throughout the document. This is done to get the vector representation of a document where the important words get a higher value

It has 2 main components

- **Term Frequency** – This is the ratio of the number of times the word occurs in the sentence as to the total number of words in the sentence.
$$TF(t,d) = \frac{Number\ of\ time\ t\ occurs\ in\ document\ d}{Number\ of\ words\ in\ document\ d}$$

- **Inverse Document Frequency** – This is the log of the ratio of the total number of documents and the documents in which the term t occurs
$$IDF(t,D) = \frac{Total\ number\ of\ documents\ in\ D}{Number\ of\ documents\ containing\ the\ word\ t}$$

TF-IDF Score – By multiplying the TF with IDF we get the TF-IDF score of documents. We get a TF score for each word from the vocab in each document whereas we get the IDF score for each of the words in the vocab. To get the TD-IDF score we multiply the values of the TF table with the corresponding value in the IDF.
$$TF - IDF = TF * IDF$$

TF_IDF-py

September 2, 2025

1 Write a program to extract features – TF, TF-IDF score from text

TF-IDF (Term Frequency–Inverse Document Frequency) is a statistical method used in natural language processing and information retrieval to evaluate how important a word is to a document in relation to a larger collection of documents

[1]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
```

[2]:
```
documents = [
    "i am henry.",
    "i like college.",
    "do henry like college?",
    "i am do i like college?",
    "i do like henry.",
    "do i like henry?",
]
```

[3]:
```
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform(documents)
feature_names = vectorizer.get_feature_names_out()
tfidf_values = {}

print("=== TF-IDF Matrix ===")
print(f"{'':4}", end="")
for i, feature in enumerate(feature_names):
    if i == len(feature_names) - 1:
        print(f"{feature:10}", end="")
    else:
        print(f"{feature:11}", end="")
print()

for doc_index in range(len(documents)):
    print(f"D{str(doc_index + 1)} {tfidf_matrix[doc_index, :].toarray()[0]}")
print()

for doc_index, doc in enumerate(documents):
    feature_index = tfidf_matrix[doc_index, :].nonzero()[1]
```

1

```
    tfidf_doc_values = zip(
        feature_index, [tfidf_matrix[doc_index, x] for x in feature_index]
    )
    tfidf_values[doc_index] = {feature_names[i]: value for i, value in␣
↪tfidf_doc_values}

for doc_index, values in tfidf_values.items():
    print(f"Document {doc_index + 1}:")
    for word, tfidf_value in values.items():
        print(f"{word}: {tfidf_value}")
    print("\n")
```

```
=== TF-IDF Matrix ===
    am         college    do         henry      like
D1  [0.81019752 0.         0.         0.58615696 0.        ]
D2  [0.         0.80383327 0.         0.         0.59485466]
D3  [0.         0.57579095 0.4934091  0.4934091  0.42609823]
D4  [0.61703105 0.52094001 0.44640601 0.         0.38550729]
D5  [0.         0.         0.60348696 0.60348696 0.52115926]
D6  [0.         0.         0.60348696 0.60348696 0.52115926]

Document 1:
am: 0.8101975203608325
henry: 0.5861569567966913


Document 2:
like: 0.5948546604855911
college: 0.8038332743166161


Document 3:
henry: 0.49340910033240876
like: 0.4260982255952437
college: 0.5757909530054383
do: 0.49340910033240876


Document 4:
am: 0.6170310457438672
like: 0.3855072948390351
college: 0.5209400071446677
do: 0.4464060070946989


Document 5:
henry: 0.6034869604104566
like: 0.5211592628257661
```

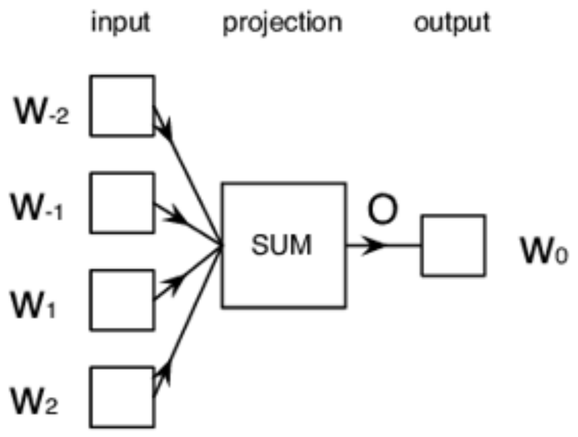
2

```
do: 0.6034869604104566


Document 6:
henry: 0.6034869604104566
like: 0.5211592628257661
do: 0.6034869604104566
```

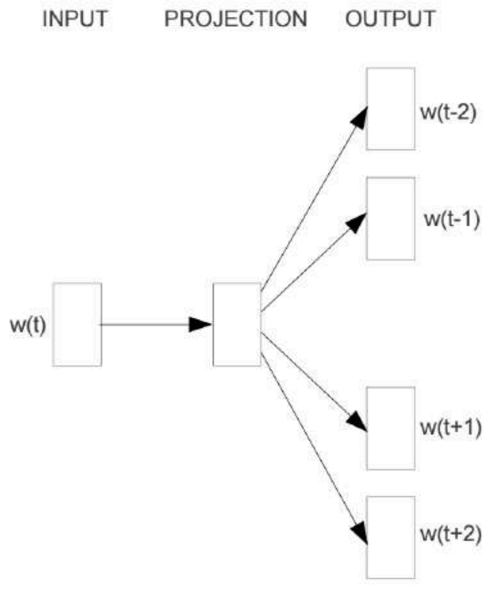
Lab 4

Q4. To implement word embeddings using Word2Vec, GloVe, FastText and BERT.

Word2Vec
It is static word embedding technique that uses shallow neural networks to get the word embeddings. We use the weights between the input and the hidden layer as the word embeddings.
It has types of implementations –
Continuous Bag of Words(CBOW)
CBOW uses the words around the target word to predict it.



Skip-Gram
Skip-gram uses a target word and then predict its surrounding words.



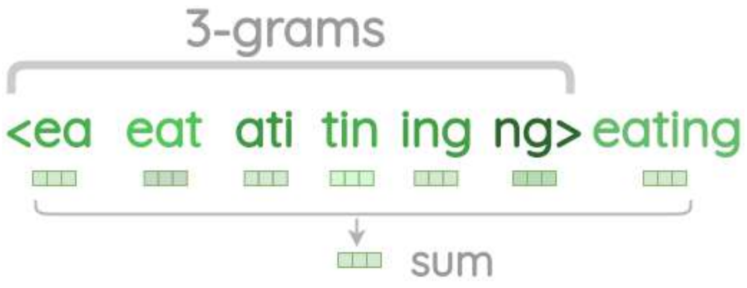
Skip-gram

GloVe(Global Vectors for Word Representation)
It is static word embedding technique that uses co-occurrence matrix to get the word embeddings. It works on the principle that words that occur together more often have a similar meaning.

So in this technique the words that occur together get placed closer in the vector space whereas the words that don't occur together are put far away from each other in the vector space

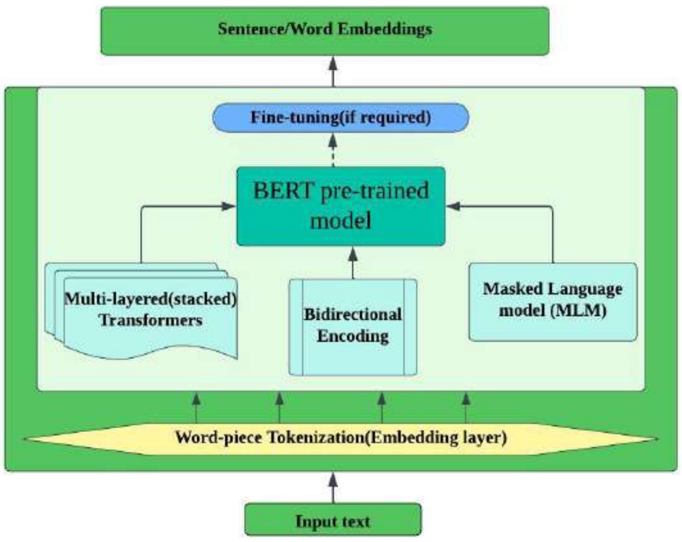
$$X = \begin{array}{c|cccccccc} & I & like & enjoy & deep & learning & NLP & flying & . \\ \hline I & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ like & 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ enjoy & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ deep & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ learning & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ NLP & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ flying & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ . & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{array}$$

FastText
It is static word embedding technique that breaks down words into further smaller n grams for which then we create word embeddings.
This allows the FastText model to predict out of vocab words too if they have parts in them that was seen before.
The embeddings of a word are the sum of all the n grams broken out from it.



BERT(Bidirectional encoder representations from transformers)
It is a word embedding technique that uses the transformer architecture.
It is a truly bidirectional model and takes in context from both the left and right of the word to embed to get the correct meaning of the word and solve the ambiguity.
It uses 2 pretraining strategies to make it bidirectional

1. Masked Language Model (MLM) - Where it hides 15% of the words randomly
2. Next Sentence Prediction (NSP) - Where it predicts the probability of a sentence following another one


Sentence/Word Embeddings
Fine-tuning(if required)
BERT pre-trained model
Multi-layered(stacked) Transformers
Bidirectional Encoding
Masked Language model (MLM)
Word-piece Tokenization(Embedding layer)
Input text

cbow

September 30, 2025

1 CBOW

Continous Bag Of Words(CBOW) is a word embedding technique under the Word2Vec method
Here we use a shallow neural netword to create our word embeddings
We feed in the surrounding words to the NN to predict the target word.

[1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
```

[ ]:
```
class CBOW(nn.Module):
    def __init__(self, vocab_size, embedding_size):
        super(CBOW, self).__init__()

        self.embeddings = nn.Embedding(vocab_size, embedding_size)
        self.linear = nn.Linear(embedding_size, vocab_size)

    def forward(self, context):
        context_embedding = self.embeddings(context.unsqueeze(0)).sum(dim=1)
        res = self.linear(context_embedding)
        return res
```

[14]:
```
window_size = 2
doc = [
    "i am henry",
    "i like college",
    "do henry like college",
    "i am do i like college",
    "i do like henry",
    "do i like henry",
]
raw_text = " ".join(doc)
tokens = raw_text.split()
vocab = set(tokens)
vocab_size = len(vocab)
```

[ ]:
```
data = []
word_index = {word: i for i, word in enumerate(vocab)}
```

```
for i in range(2, len(tokens) - 2):
    context = [
        word_index[word]
        for word in tokens[i - window_size : i] + tokens[i + 1 : i +␣
 ↪window_size + 1]
    ]
    target_word = word_index[tokens[i]]
    data.append((torch.tensor(context), torch.tensor(target_word)))

print(data)
```

```
[(tensor([1, 0, 1, 3]), tensor(2)), (tensor([0, 2, 3, 5]), tensor(1)),
(tensor([2, 1, 5, 4]), tensor(3)), (tensor([1, 3, 4, 2]), tensor(5)),
(tensor([3, 5, 2, 3]), tensor(4)), (tensor([5, 4, 3, 5]), tensor(2)),
(tensor([4, 2, 5, 1]), tensor(3)), (tensor([2, 3, 1, 0]), tensor(5)),
(tensor([3, 5, 0, 4]), tensor(1)), (tensor([5, 1, 4, 1]), tensor(0)),
(tensor([1, 0, 1, 3]), tensor(4)), (tensor([0, 4, 3, 5]), tensor(1)),
(tensor([4, 1, 5, 1]), tensor(3)), (tensor([1, 3, 1, 4]), tensor(5)),
(tensor([3, 5, 4, 3]), tensor(1)), (tensor([5, 1, 3, 2]), tensor(4)),
(tensor([1, 4, 2, 4]), tensor(3)), (tensor([4, 3, 4, 1]), tensor(2)),
(tensor([3, 2, 1, 3]), tensor(4)), (tensor([2, 4, 3, 2]), tensor(1))]
```

[16]:
```
embed_size = 10
learning_rate = 0.01
epochs = 1000

model = CBOW(vocab_size, embed_size)
lossfn = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

[ ]:
```
for epoch in range(epochs):
    total_loss = 0
    for context, target in data:
        optimizer.zero_grad()
        output = model(context)
        loss = lossfn(output, target.unsqueeze(0))
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    if epoch % 50 == 0:
        print(epoch, total_loss)
```

```
0 43.41829997301102
50 10.840636879205704
100 7.696862827986479
150 6.366553973406553
200 5.675506556406617
```

```
250 5.270045618060976
300 5.008826375938952
350 4.827488952549174
400 4.693799571483396
450 4.590332630323246
500 4.507075492525473
550 4.437947267317213
600 4.379104012215976
650 4.327999471192015
700 4.28289133211365
750 4.242555095930584
800 4.20609219041944
850 4.1728397329716245
900 4.142292355696554
950 4.11404877804307
```

[18]:
```
word_to_lookup = "henry"
wi = word_index[word_to_lookup]
embedding = model.embeddings(torch.tensor([wi]))
print(f"Embedding for '{word_to_lookup}': {embedding.detach().numpy()}")
```

```
Embedding for 'henry': [[-0.8072041   0.8175361  -0.68911165 -0.32924142
-0.23396243  0.68101513
  -2.9420059  -0.21836732  0.3715383  -0.33744484]]
```

3

skip_gram

September 30, 2025

1 Skip-Gram

This is a word embedding technique under the Word2Vec method
Here we use a shallow neural netword to create our word embeddings
We feed in a single word to the NN to predict the surrouding words.

[33]:
```
import torch
import torch.nn as nn
import torch.optim as optim
```

[34]:
```
class SkipGram(nn.Module):
    def __init__(self, vocab_size, window_size, embedding_size):
        super(SkipGram, self).__init__()

        self.embeddings = nn.Embedding(vocab_size, embedding_size)
        self.linear = nn.Linear(embedding_size, vocab_size)

    def forward(self, target):
        target_embedding = self.embeddings(target)
        res = self.linear(target_embedding)
        return res
```

[35]:
```
window_size = 2
doc = [
    "i am henry",
    "i like college",
    "do henry like college",
    "i am do i like college",
    "i do like henry",
    "do i like henry",
]
raw_text = " ".join(doc)
tokens = raw_text.split()
vocab = set(tokens)
vocab_size = len(vocab)
```

[36]:
```
data = []
word_index = {word: i for i, word in enumerate(vocab)}
```

```
# onehot = {word: [0 for _ in vocab] for word in vocab}
# for i, word in enumerate(vocab):
#     onehot[word][i] = 1

for i in range(2, len(tokens) - 2):
    context = word_index[tokens[i]]
    for j in range(-window_size, window_size + 1):
        if j == 0:
            continue
        data.append((context, word_index[tokens[i + j]]))
print(data)
```

```
[(1, 5), (1, 4), (1, 5), (1, 2), (5, 4), (5, 1), (5, 2), (5, 3), (2, 1), (2, 5),
(2, 3), (2, 0), (3, 5), (3, 2), (3, 0), (3, 1), (0, 2), (0, 3), (0, 1), (0, 2),
(1, 3), (1, 0), (1, 2), (1, 3), (2, 0), (2, 1), (2, 3), (2, 5), (3, 1), (3, 2),
(3, 5), (3, 4), (5, 2), (5, 3), (5, 4), (5, 0), (4, 3), (4, 5), (4, 0), (4, 5),
(0, 5), (0, 4), (0, 5), (0, 2), (5, 4), (5, 0), (5, 2), (5, 3), (2, 0), (2, 5),
(2, 3), (2, 5), (3, 5), (3, 2), (3, 5), (3, 0), (5, 2), (5, 3), (5, 0), (5, 2),
(0, 3), (0, 5), (0, 2), (0, 1), (2, 5), (2, 0), (2, 1), (2, 0), (1, 0), (1, 2),
(1, 0), (1, 5), (0, 2), (0, 1), (0, 5), (0, 2), (5, 1), (5, 0), (5, 2), (5, 1)]
```

[37]:
```
embed_size = 10
learning_rate = 0.01
epochs = 1000

model = SkipGram(vocab_size, window_size, embed_size)
lossfn = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

[38]:
```
for epoch in range(epochs):
    total_loss = 0
    for context, target in data:
        optimizer.zero_grad()
        output = model(torch.tensor([context]))
        loss = lossfn(output, torch.tensor([target]))
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    if epoch % 50 == 0:
        print(epoch, total_loss / len(data))
```

```
0 1.8511932469904422
50 1.545641028881073
100 1.5361952051520347
150 1.5330601558089256
200 1.5311076186597348
250 1.529517511278391
300 1.5280845649540424
```

2

```
350 1.5267456322908401
400 1.5254790008068084
450 1.524276676028967
500 1.5231351152062416
550 1.5220524609088897
600 1.5210270047187806
650 1.5200570434331895
700 1.5191406615078449
750 1.518275821954012
800 1.5174602322280406
850 1.5166916601359843
900 1.515967819094658
950 1.5152863003313541
```

[39]:
```python
word_to_lookup = "henry"
wi = word_index[word_to_lookup]
embedding = model.embeddings(torch.tensor([wi]))
print(f"Embedding for '{word_to_lookup}': {embedding.detach().numpy()}")
```

```
Embedding for 'henry': [[-1.7774011   1.436296   -0.60040134  1.001669
0.47328883  0.277007
  -1.6477649   1.3790239   0.685      -0.6399059 ]]
```

fasttext

September 30, 2025

1 FastText

FastText is a word embedding technique that breaks down words into n grams and creates the word embeddings for those smaller units
This helps FastText to also find out ebmedding for previously unseen words by using the smaller parts of the word

[1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
```

[2]:
```
class SkipGram(nn.Module):
    def __init__(self, vocab_size, window_size, embedding_size):
        super(SkipGram, self).__init__()

        self.embeddings = nn.Embedding(vocab_size, embedding_size)
        self.linear = nn.Linear(embedding_size, vocab_size)

    def forward(self, target):
        target_embedding = self.embeddings(target)
        res = self.linear(target_embedding)
        return res
```

[3]:
```
window_size = 3
doc = [
    "<i am henry>",
    "<i like college>",
    "<do henry like college>",
    "<i am do i like college>",
    "<i do like henry>",
    "<do i like henry>",
]
raw_text = " ".join(doc)
tokens = raw_text.split(" ")
```

[4]:
```
def get_new_tokens(tok):
    char_tokens = []
    for token in tok:
        if len(token) < window_size:
            char_tokens.append(token)
        for i in range(0, len(token) - window_size + 1):
            char_tokens.append(token[i : i + window_size])
    return char_tokens
```

[5]:
```
char_tokens = get_new_tokens(tokens)
print(char_tokens[:10])
vocab = set(char_tokens)
vocab_size = len(vocab)
```

```
['<i', 'am', 'hen', 'enr', 'nry', 'ry>', '<i', 'lik', 'ike', 'col']
```

[6]:
```
data = []
word_index = {word: i for i, word in enumerate(vocab)}

for i in range(window_size, len(char_tokens) - window_size):
    context = word_index[char_tokens[i]]
    for j in range(-window_size, window_size + 1):
        if j == 0:
            continue
        data.append((context, word_index[char_tokens[i + j]]))
print(data[:10])
```

```
[(3, 16), (3, 0), (3, 1), (3, 13), (3, 14), (3, 16), (13, 0), (13, 1), (13, 3),
(13, 14)]
```

[7]:
```
embed_size = 10
learning_rate = 0.01
epochs = 1000

model = SkipGram(vocab_size, window_size, embed_size)
lossfn = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

[8]:
```
for epoch in range(epochs):
    total_loss = 0
    for context, target in data:
        optimizer.zero_grad()
        output = model(torch.tensor([context]))
        loss = lossfn(output, torch.tensor([target]))
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    if epoch % 50 == 0:
        print(epoch, total_loss / len(data))
```

```
0 2.929987609791918
50 2.3488601352892764
```

```
100 2.2823653622549407
150 2.261180039165782
200 2.253301527629904
250 2.249035820263584
300 2.2461052427486496
350 2.2436619603714973
400 2.241289444521171
450 2.2389877912949543
500 2.2370689385602263
550 2.235602776614987
600 2.234455091207206
650 2.2335173297090596
700 2.2327251633008323
750 2.2320393364445694
800 2.2314340431673996
850 2.230892019206975
900 2.230401324982546
950 2.2299533826964244
```

[9]:
```
word_to_lookup = "henbenry"
lookup = get_new_tokens([word_to_lookup])
res = []
for lu in lookup:
    if lu in word_index.keys():
        wi = word_index[lu]
        embedding = model.embeddings(torch.tensor([wi]))
        res.append(embedding.detach().numpy()[0])
        print(f"Embedding for '{lu}': {embedding.detach().numpy()}")

print(f"Embedding for {word_to_lookup}: {sum(res) / len(res)}")
```

```
Embedding for 'hen': [[-0.8204228   0.10082058 -0.7994618  -0.63440734
-0.32108185  0.3642752
   0.25619638  0.5068886   2.0844624  -0.01407014]]
Embedding for 'enr': [[-0.424241   -0.6773601  -0.7903334   0.26817885
0.25904417  1.2208512
  -0.5175885   1.9262754  -0.9821287  -0.88805467]]
Embedding for 'nry': [[ 0.29178938 -2.0117695  -0.89259404 -1.5418677
-0.04163222 -0.38947254
  -0.5865991  -0.7319726  -1.0225619  -0.28157222]]
Embedding for henbenry: [-0.3176248  -0.86276966 -0.8274631  -0.63603204
-0.03455663  0.39855132
 -0.28266373  0.56706387  0.02659062 -0.39456567]
```

glove

September 30, 2025

1 GloVe

Global Vectors for Word Representation(GloVe) is a word embedding technique that generates vector representations of words based on co-occurrence matrix.
Words that occur together in the cooccurence matrix are closer in the vector space and words that dont occour together get pushed away from each other

[1]:
```
from gensim.downloader import load

glove_model = load("glove-wiki-gigaword-50")
embeddings = glove_model.get_vector("henry")
print(len(embeddings))
print(embeddings)
```

```
50
[ 0.27137   0.61347  -0.52498  -0.7617    0.37252   0.21401  -1.0817
  0.16501  -0.45105  -0.77013  -0.55703   1.7441   -0.70152  -1.6522
  0.047489 -0.16412   0.10451  -0.86788  -0.71332  -0.12108   0.12376
 -0.14268  -0.18158  -0.83291   0.21332  -1.6042    0.22437  -1.0787
 -0.53084   0.44731   1.207    -0.94696  -0.082957 -0.39328   0.70247
 -0.22002  -0.040118  0.79309   0.19654  -0.25612   0.82113   0.8323
 -0.63336  -0.32116   0.21277   0.47812  -0.41281  -1.7192   -0.26018
  0.57005 ]
```

bert

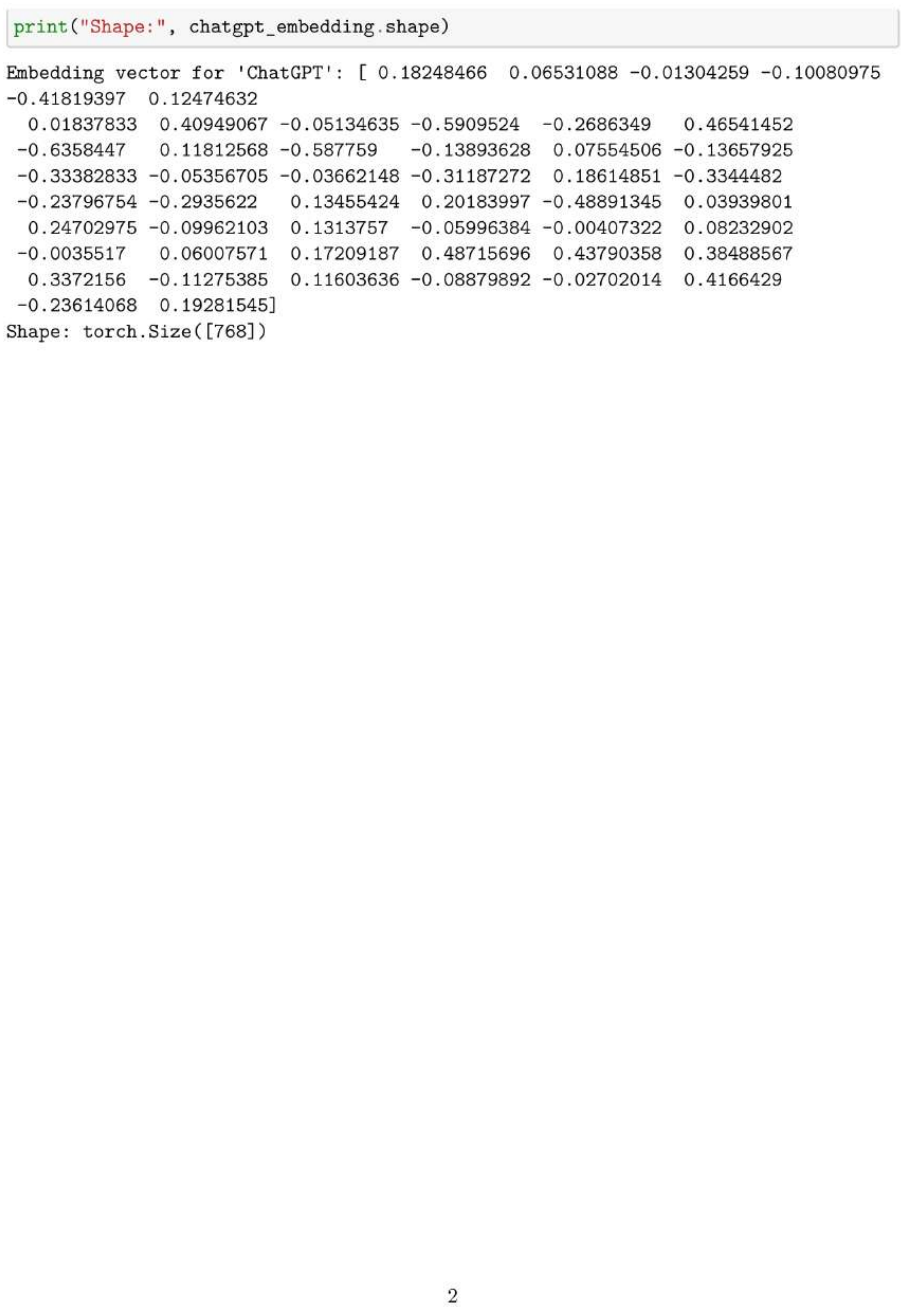
September 30, 2025

1 BERT

Bidirectional encoder representations from transformers(BERT) is a word embedding technique that uses the transformer architecture.
It is a truely bidirectional model and takes in context from both the left and right of the word to embed to get the correct meaning of the word and solve the ambiguity.
It uses 2 pretraining stratergies to make it bidirectional 1. Masked Language Model (MLM) - Where it hides 15% of the words randomly 2. Next Sentence Prediction (NSP) - Where it predicts the probabilty of a sentence following another one

[ ]:
```
from transformers import BertTokenizer, BertModel
import torch

# Load tokenizer and model
tokenizer = BertTokenizer.from_pretrained("bert-base-cased")
model = BertModel.from_pretrained("bert-base-cased")

text = "ChatGPT is a language model developed by OpenAI, based on the GPT␣
 ↪(Generative Pre-trained Transformer) architecture."

# Tokenize and encode the text
encoding = tokenizer(text, return_tensors="pt")
input_ids = encoding["input_ids"]
attention_mask = encoding["attention_mask"]

tokens = tokenizer.convert_ids_to_tokens(input_ids[0])
print("Tokens:", tokens)

# Get embeddings from BERT
with torch.no_grad():
    outputs = model(input_ids, attention_mask=attention_mask)
embeddings = outputs.last_hidden_state  # shape: (1, seq_len, hidden_size)
```

[4]:
```
word_tokens = ["Cha", "##t", "##GP", "##T"]
word_indices = [i for i, t in enumerate(tokens) if t in word_tokens]

chatgpt_embedding = embeddings[0, word_indices, :].mean(dim=0)
print("Embedding vector for 'ChatGPT':", chatgpt_embedding.numpy()[:50])
```



```python
print("Shape:", chatgpt_embedding.shape)
```

```
Embedding vector for 'ChatGPT': [ 0.18248466  0.06531088 -0.01304259 -0.10080975
-0.41819397  0.12474632
  0.01837833  0.40949067 -0.05134635 -0.5909524  -0.2686349   0.46541452
 -0.6358447   0.11812568 -0.587759   -0.13893628  0.07554506 -0.13657925
 -0.33382833 -0.05356705 -0.03662148 -0.31187272  0.18614851 -0.3344482
 -0.23796754 -0.2935622   0.13455424  0.20183997 -0.48891345  0.03939801
  0.24702975 -0.09962103  0.1313757  -0.05996384 -0.00407322  0.08232902
 -0.0035517   0.06007571  0.17209187  0.48715696  0.43790358  0.38488567
  0.3372156  -0.11275385  0.11603636 -0.08879892 -0.02702014  0.4166429
 -0.23614068  0.19281545]
Shape: torch.Size([768])
```

2

Lab 5

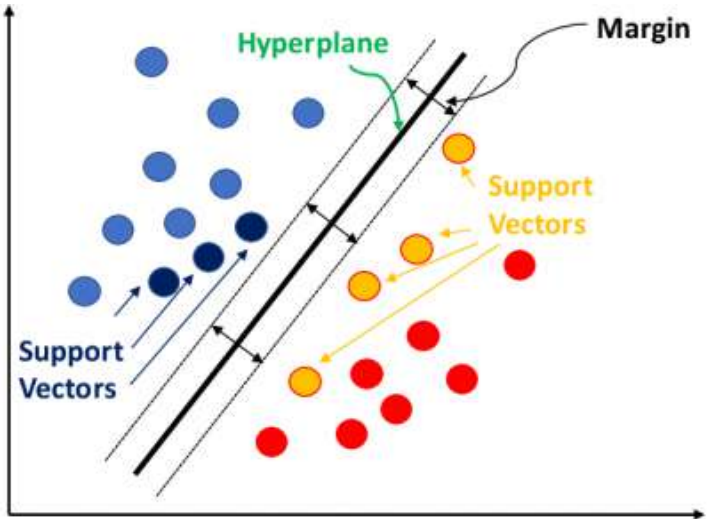
Q5. To implement the text classification using Naïve Bayes' and SVM Classifier.

Naïve bayes –
Multinomial Naive Bayes is a probabilistic classification algorithm based on Bayes' theorem, commonly used for discrete features such as word counts in text classification.
It assumes that -

1. Features are conditionally independent given the class
2. That feature frequencies follow a multinomial distribution

The model calculates the probability of a class by combining prior probabilities with the likelihood of observing the given feature counts within that class. It performs well for tasks like spam detection and sentiment analysis, where the relative frequency of words is informative.

Support vector machine/classifier(SVC) –
Support Vector Classifier is a supervised learning algorithm that aims to find the best linear boundary separating different classes in a dataset.
It works by maximizing the margin—the distance between the decision boundary and the nearest data points from each class, known as support vectors.
Linear SVC is effective for high-dimensional data such as text or image features and is robust to overfitting when properly regularized . It performs well when classes are linearly separable or nearly so.



NLP-classification_practical

October 8, 2025

1 Classification

Classification models help us to segregate data into seprate classes to find out the for example if a email is spam or not

```
[25]: from sklearn.model_selection import train_test_split
      from sklearn.naive_bayes import MultinomialNB as MNB
      from sklearn.svm import LinearSVC
      from sklearn.metrics import (
          confusion_matrix,
          accuracy_score,
          precision_score,
          recall_score,
          f1_score,
      )
      import pandas as pd
      from sklearn.feature_extraction.text import (
          TfidfVectorizer,
          CountVectorizer,
          HashingVectorizer,
      )
      from sklearn.preprocessing import LabelEncoder
      from datetime import datetime
```

```
[26]: dataset = pd.read_csv("./IMDB Dataset.csv")
      X = dataset["review"].values
      y = dataset["sentiment"].values
```

```
[27]: label_encoder = LabelEncoder()
      y = label_encoder.fit_transform(y)
      print(y[:5])
```

```
[1 1 1 0 1]
```

```
[55]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
      print(f"X-train {len(X_train)}")
      print(f"X-test {len(X_test)}")
```

1

```
X-train 40000
X-test 10000
```

[56]:
```
vectorizer = TfidfVectorizer(stop_words="english", ngram_range=(1, 3), min_df=2)
# vectorizer = HashingVectorizer()
# vectorizer = CountVectorizer()
X_train = vectorizer.fit_transform(X_train)
X_test = vectorizer.transform(X_test)
```

[57]:
```
def eval_res(true, pred):
    cm = confusion_matrix(true, pred)
    print(cm)
    print(f"Accuracy = {accuracy_score(true, pred)}")
    print(f"Precision = {precision_score(true, pred)}")
    print(f"Recall = {recall_score(true, pred)}")
    print(f"F1 = {f1_score(true, pred)}")
```

1.1 Multinomial Naive Bayes

It deal with the frequency of words for our predictions.
It assumes that the features are independent to one another

[58]:
```
start = datetime.now()
model = MNB()
model.fit(X_train, y_train)
pred = model.predict(X_test)
print(f"Total time = {datetime.now() - start}")
eval_res(y_test, pred)
```

```
Total time = 0:00:00.076304
[[4431  596]
 [ 497 4476]]
Accuracy = 0.8907
Precision = 0.8824921135646687
Recall = 0.9000603257590991
F1 = 0.8911896465903435
```

1.2 Support Vector Machine/Classifier(SVC)

SVC use the concept of support vectors to position the seprator between classes. It also has a margin - the distance from the boundary to the closent point to maximise class seperation

[59]:
```
start = datetime.now()
model = LinearSVC(C=0.5)
model.fit(X_train, y_train)
pred = model.predict(X_test)
print(f"Total time = {datetime.now() - start}")
eval_res(y_test, pred)
```

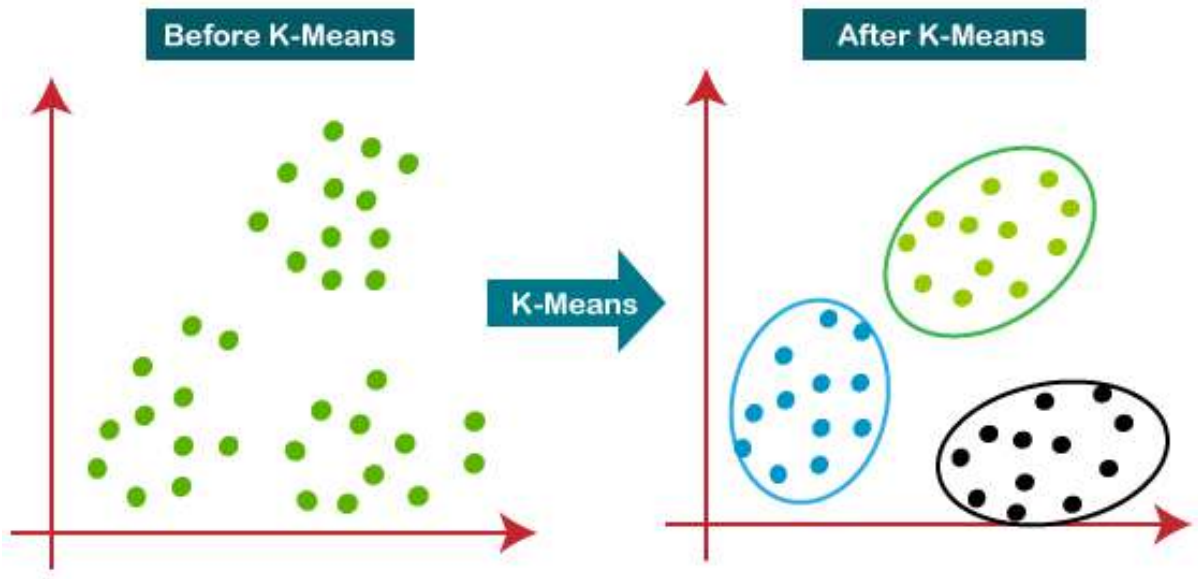
```
Total time = 0:00:00.477187
[[4507  520]
 [ 388 4585]]
Accuracy = 0.9092
Precision = 0.8981390793339863
Recall = 0.9219786848984516
F1 = 0.9099027584838262
```

Lab 6

Q6. To implement K-Means Clustering Algorithm on text.

K-Means Clustering is an unsupervised machine learning algorithm that helps group data points into clusters based on their inherent similarity. Unlike supervised learning, where we train models using labeled data, K-Means is used when we have data that is not labeled, and the goal is to uncover hidden patterns or structures.

1. **Initialization:** We begin by randomly selecting k cluster centroids.
2. **Assignment Step:** Each data point is assigned to the nearest centroid, forming clusters.
3. **Update Step:** After the assignment, we recalculate the centroid of each cluster by averaging the points within it.
4. **Repeat:** This process repeats until the centroids no longer change or the maximum number of iterations is reached.



NLP-K_means

October 22, 2025

1 K-means Clustering

K-means clustering is an unsupervised machine learning algorithm that groups data points into clusters based on their similarity, using the distance to cluster centroid
It aims to minimize the variance within each cluster by assigning points to the nearest centroid and updating the centroids iteratively until convergence

```
[ ]: from sklearn.cluster import KMeans
     import pandas as pd
     from sklearn.feature_extraction.text import TfidfVectorizer
     import matplotlib.pyplot as plt
     from sklearn.decomposition import PCA
```

```
[13]: # dataset = pd.read_csv("../classification/spam.csv", encoding="latin1")
      dataset = pd.read_csv("../classification/IMDB Dataset.csv")
      X_raw = dataset["review"].values
```

```
[14]: vectorizer = TfidfVectorizer()
      X = vectorizer.fit_transform(X_raw)
```

```
[15]: model = KMeans(n_clusters=2)
      pred = model.fit_predict(X)
```

```
[16]: dataset["pred"] = pred
```

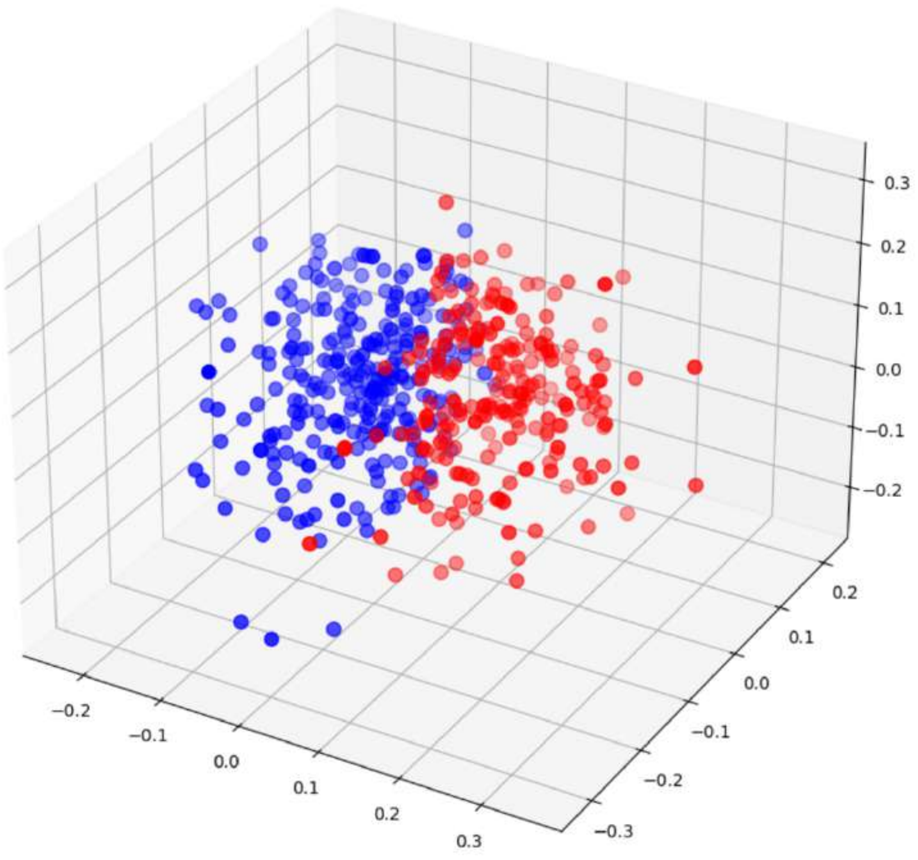
```
[44]: dataset.head()
```

```
[44]:                                               review sentiment  pred
      0  One of the other reviewers has mentioned that ...  positive     0
      1  A wonderful little production. <br /><br />The...  positive     0
      2  I thought this was a wonderful way to spend ti...  positive     0
      3  Basically there's a family where a little boy ...  negative     0
      4  Petter Mattei's "Love in the Time of Money" is...  positive     0
```

```
[ ]: def display_pca_scatterplot(vectorizer, dataset):
         fig = plt.figure(figsize=(10, 10))
         ax = fig.add_subplot(projection="3d")
         pca = PCA()
```

```
    pca.fit(vectorizer.transform(dataset["review"].values))
    for i in range(2):
        dataset_0 = dataset.loc[dataset["pred"] == i]
        words = dataset_0["review"].values
        word_vectors = vectorizer.transform(words)

        twodim = pca.transform(word_vectors)[:, :3]

        ax.scatter(
            twodim[:, 0], twodim[:, 1], twodim[:, 2], c="r" if i == 0 else "b",␣
↪s=64
        )
```

[54]:
```
display_pca_scatterplot(vectorizer, dataset[:500])
```



2

Lab 7

Q7. To Implement PoS Tagging on text.

POS tagging, or part-of-speech tagging, is the process of assigning grammatical categories like nouns, verbs, and adjectives to each word in a sentence. This helps machines understand the structure and meaning of sentences by identifying the roles of words and their relationships.

Benefits of POS Tagging

- Helps determine grammatical structure and interpretation of sentences.
- Enables many NLP tasks such as parsing, NER, machine translation, and text-to-speech.
- Improves accuracy of downstream models by providing syntactic context.
- Useful for information extraction, semantic understanding, and keyword analysis.
- Supports linguistic analysis and corpus research.

Challenges of POS Tagging

- Same word can have multiple meanings depending on context (e.g., "book" as noun vs verb).
- Ambiguity and complex grammar reduce accuracy in real-world text.
- Performance varies across languages and informal text (tweets, slang, code-mixing).
- Requires large annotated datasets for high accuracy.
- Domain adaptation is difficult—models trained on news may fail on medical or legal text.
- OOV (Out-of-Vocabulary) and rare words are hard to tag correctly.

NLP-pos_tagging-code

December 1, 2025

1 POS Tagging

POS tagging, or part-of-speech tagging, is the process of assigning grammatical categories like nouns, verbs, and adjectives to each word in a sentence. This helps machines understand the structure and meaning of sentences by identifying the roles of words and their relationships.

1.1 ML Based POS tagging

[1]:
```
from nltk.tag.perceptron import PerceptronTagger
import nltk
from nltk import pos_tag
from datetime import datetime
```

[2]:
```
nltk.download("averaged_perceptron_tagger_eng")
start = datetime.now()
tagger = PerceptronTagger()
print(f"Time taken = {datetime.now() - start}")
```

```
Time taken = 0:00:00.086204
```

```
[nltk_data] Downloading package averaged_perceptron_tagger_eng to
[nltk_data]     /Users/tushya/nltk_data…
[nltk_data]   Package averaged_perceptron_tagger_eng is already up-to-
[nltk_data]       date!
```

[3]:
```
sent = "This is an example sentence"
start = datetime.now()
tags = tagger.tag(sent.split())
print(f"Time taken = {datetime.now() - start}")
print(tags)
```

```
Time taken = 0:00:00.000170
[('This', 'DT'), ('is', 'VBZ'), ('an', 'DT'), ('example', 'NN'), ('sentence',
'NN')]
```

[4]:
```
tags = pos_tag(sent.split())
print(tags)
```

```
[('This', 'DT'), ('is', 'VBZ'), ('an', 'DT'), ('example', 'NN'), ('sentence',
'NN')]
```

1.2 DL Based POS Tagging

[6]:
```
from spacy import load
```

[7]:
```
sent = "This is an example sentence"
start = datetime.now()
nlp = load("en_core_web_sm")
print(f"Time taken = {datetime.now() - start}")
start = datetime.now()
doc = nlp(sent)
print(f"Time taken = {datetime.now() - start}")
```

```
Time taken = 0:00:00.284408
Time taken = 0:00:00.018162
```

[8]:
```
for token in doc:
    print(f"{token.text:10}: {token.pos_}")
```

```
This      : PRON
is        : AUX
an        : DET
example   : NOUN
sentence  : NOUN
```

Lab 8

Q8. To Implement text processing with neural network.

Implementing text processing with neural networks involves converting raw text into numerical representations and then training neural architectures to perform tasks such as sentiment analysis, POS tagging, text classification, translation, summarization, and more. Key steps include data preprocessing, vectorization, model design, training, and evaluation.

Essential components:

- **Tokenization**: Splitting text into words, subwords, or characters.
- **Text normalization**: Lowercasing, removing noise, handling punctuation, stopword removal (optional depending on model).
- **Numerical representation**: Converting tokens into vectors using methods like one-hot encoding, TF-IDF, word embeddings (Word2Vec, GloVe, FastText), or contextual embeddings (BERT).
- **Neural architectures**: Using RNNs (LSTM, GRU), CNNs, Transformer models, or hybrid combinations to capture semantic relationships and contextual meaning.
- **Training and optimization**: Applying backpropagation, loss functions (e.g., cross-entropy), gradient descent to optimize model performance.
- **Evaluation**: Using metrics such as accuracy, F1-score, precision, recall on test data.

Benefits:

- Automatically learns feature representations
- Handles large and complex datasets effectively
- Achieves high accuracy and generalization with modern transformer architectures

Challenges:

- Requires significant computational power and large training data
- Can be difficult to interpret and explain decisions

NLP-NN-code

December 1, 2025

1 Using Neural Nets for text processing

[1]:
```
import nltk
from nltk.corpus import stopwords
import spacy

nltk.download("stopwords")

nlp = spacy.load("en_core_web_sm")
stop_words = set(stopwords.words("english"))


def preprocess(text):
    doc = nlp(text.lower())
    tokens = [
        tok.lemma_ for tok in doc if tok.is_alpha and tok.lemma_ not in␣
 ↪stop_words
    ]
    return " ".join(tokens)
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/tushya/nltk_data…
[nltk_data]   Unzipping corpora/stopwords.zip.
```

[3]:
```
from nltk.corpus import movie_reviews
import random

nltk.download("movie_reviews")
nltk.download("stopwords")

docs = [
    (movie_reviews.raw(fid), category)
    for category in movie_reviews.categories()
    for fid in movie_reviews.fileids(category)
]

random.shuffle(docs)
```

```
texts = [t for t, _ in docs]
labels = [1 if c == "pos" else 0 for _, c in docs]
```

```
[nltk_data] Downloading package movie_reviews to
[nltk_data]     /Users/tushya/nltk_data…
[nltk_data]   Unzipping corpora/movie_reviews.zip.
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/tushya/nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
```

1.1 Using the TF-IDF model for vector embeddings

[ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer

X = [preprocess(t) for t in texts]  # texts: raw dataset list
y = labels  # labels: list

vectorizer = TfidfVectorizer(max_features=5000)
X_vec = vectorizer.fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_vec, y, test_size=0.2)
```

[ ]:
```
import torch
from torch.utils.data import TensorDataset, DataLoader

train_data = TensorDataset(
    torch.tensor(X_train.toarray(), dtype=torch.float32), torch.tensor(y_train)
)
train_loader = DataLoader(train_data, batch_size=32, shuffle=True)

test_data = TensorDataset(
    torch.tensor(X_test.toarray(), dtype=torch.float32), torch.tensor(y_test)
)
test_loader = DataLoader(test_data, batch_size=32)
```

1.2 Using a linear NN model

[ ]:
```
import torch.nn as nn


class TextClassifier(nn.Module):
    def __init__(self, input_dim, hidden=256, num_classes=2):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_dim, hidden),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(hidden, num_classes),
        )

    def forward(self, x):
        return self.net(x)


model = TextClassifier(input_dim=X_train.shape[1])
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

[7]:
```
for epoch in range(10):
    model.train()
    for xb, yb in train_loader:
        optimizer.zero_grad()
        pred = model(xb)
        loss = loss_fn(pred, yb)
        loss.backward()
        optimizer.step()

    print("epoch", epoch, "loss", loss.item())
```

```
epoch 0 loss 0.5786972045898438
epoch 1 loss 0.29843029379844666
epoch 2 loss 0.16156941652297974
epoch 3 loss 0.04248003289103508
epoch 4 loss 0.03821723163127899
epoch 5 loss 0.030771629884839058
epoch 6 loss 0.007813837379217148
epoch 7 loss 0.006546510849148035
epoch 8 loss 0.004782829899340868
epoch 9 loss 0.0019606593996286392
```

[8]:
```
from sklearn.metrics import accuracy_score

model.eval()
preds = []
true = []
for xb, yb in test_loader:
    with torch.no_grad():
        p = model(xb).argmax(dim=1)
        preds.extend(p.tolist())
        true.extend(yb.tolist())

print("Accuracy:", accuracy_score(true, preds))
```

```
Accuracy: 0.84
```

Lab 9

Q9. To Implement text processing with LSTM.

LSTM, or Long Short-Term Memory, is a type of recurrent neural network (RNN) designed to effectively learn from sequences of data by addressing the vanishing gradient problem. It uses memory cells and gates to retain or forget information over long periods, making it suitable for tasks like language translation, speech recognition, and time series forecasting.

Key points

- LSTM processes text sequentially, remembering important information through memory cells and gates (input, forget, output).
- Converts sentences into numerical form using embeddings (Word2Vec, GloVe, or trainable embedding layers).
- Learns context and order of words, unlike bag-of-words or pure statistical models.
- Can predict next words, classify text sentiment, tag parts of speech, or label named entities.

Advantages:

- Captures long-term dependencies well
- Better context understanding than traditional ML models
- Works with variable-length text sequences

Limitations:

- Training is slower than simple RNN or ML models
- Requires significant data and tuning
- Can be outperformed by transformer models (e.g., BERT) in many modern tasks

NLP-LSTM-code

December 1, 2025

1 LSTM based text processing

[1]:
```
import nltk
from nltk.corpus import movie_reviews
import random

nltk.download("movie_reviews")
nltk.download("stopwords")

docs = [
    (movie_reviews.words(fid), category)
    for category in movie_reviews.categories()
    for fid in movie_reviews.fileids(category)
]

random.shuffle(docs)

texts = [list(t) for t, _ in docs]
labels = [1 if c == "pos" else 0 for _, c in docs]
```

```
[nltk_data] Downloading package movie_reviews to
[nltk_data]     /Users/tushya/nltk_data…
[nltk_data]   Package movie_reviews is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/tushya/nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
```

[2]:
```
import spacy
from nltk.corpus import stopwords

stop_words = set(stopwords.words("english"))

nlp = spacy.load("en_core_web_md")


def preprocess(tokens):
    doc = nlp(" ".join(tokens).lower())
    return [tok.lemma_ for tok in doc if tok.is_alpha and tok.lemma_ not in␣
↪stop_words]


texts = [preprocess(t) for t in texts]
```

[3]:
```
from collections import Counter

vocab = Counter()
for t in texts:
    vocab.update(t)

max_vocab = 20000
vocab = {w: i + 2 for i, (w, _) in enumerate(vocab.most_common(max_vocab))}
vocab["<PAD>"] = 0
vocab["<UNK>"] = 1


def encode(tokens, max_len=1000):
    ids = [vocab.get(w, 1) for w in tokens][:max_len]
    return ids + [0] * (max_len - len(ids))


X = [encode(t) for t in texts]
y = labels
```

[4]:
```
from sklearn.model_selection import train_test_split
import torch
from torch.utils.data import TensorDataset, DataLoader

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

train_data = TensorDataset(torch.tensor(X_train), torch.tensor(y_train))
test_data = TensorDataset(torch.tensor(X_test), torch.tensor(y_test))

train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
test_loader = DataLoader(test_data, batch_size=32)
```

1.1 Using pre-trained ebmeddings with a bi-directional LSTM model

[11]:
```
import torch.nn as nn

embedding_matrix = torch.zeros((len(vocab), 300))
for word, idx in vocab.items():
    if word in nlp.vocab:
        embedding_matrix[idx] = torch.tensor(nlp.vocab[word].vector)
```

```
class BiLSTMClassifier(nn.Module):
    def __init__(self, embedding_matrix, hidden_dim=128, num_classes=2,␣
 ↪dropout=0.3):
        super().__init__()

        # load pretrained embeddings
        self.embedding = nn.Embedding.from_pretrained(
            embedding_matrix, freeze=False, padding_idx=0
        )

        embed_dim = embedding_matrix.size(1)  # 300 for GloVe
        self.lstm = nn.LSTM(
            input_size=embed_dim,
            hidden_size=hidden_dim,
            batch_first=True,
            bidirectional=True,
        )

        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Linear(hidden_dim * 2, num_classes)  # bi-directional => x2

    def forward(self, x):
        x = self.embedding(x)
        output, (h_n, c_n) = self.lstm(x)

        # concatenate final forward and backward hidden states
        h_final = torch.cat((h_n[-2], h_n[-1]), dim=1)

        h_final = self.dropout(h_final)
        logits = self.fc(h_final)
        return logits


model = BiLSTMClassifier(
    embedding_matrix=torch.tensor(embedding_matrix, dtype=torch.float32),
    hidden_dim=128,
    num_classes=2,
)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

```
/var/folders/q6/q617z1552s929bppjqdg260h0000gn/T/ipykernel_66712/2625552409.py:4
2: UserWarning: To copy construct from a tensor, it is recommended to use
sourceTensor.detach().clone() or
sourceTensor.detach().clone().requires_grad_(True), rather than
torch.tensor(sourceTensor).
```

```
 embedding_matrix=torch.tensor(embedding_matrix, dtype=torch.float32),
```

[ ]:
```
for epoch in range(10):
    model.train()
    for batch_x, batch_y in train_loader:
        optimizer.zero_grad()
        logits = model(batch_x)
        loss = criterion(logits, batch_y)
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch + 1}, loss = {loss.item():.4f}")
```

```
Epoch 1, loss = 0.7075
Epoch 2, loss = 0.6480
Epoch 3, loss = 0.4769
Epoch 4, loss = 0.1951
Epoch 5, loss = 0.0871
Epoch 6, loss = 0.0120
Epoch 7, loss = 0.0076
Epoch 8, loss = 0.0237
Epoch 9, loss = 0.0260
Epoch 10, loss = 0.0024
```

[13]:
```
from sklearn.metrics import accuracy_score

model.eval()
preds = []
true = []

with torch.no_grad():
    for xb, yb in test_loader:
        p = model(xb).argmax(dim=1)
        preds.extend(p.tolist())
        true.extend(yb.tolist())

print("Accuracy:", accuracy_score(true, preds))
```

```
Accuracy: 0.615
```

Lab 10

Q10. To Develop any one NLP application.

Text summarization is the process of reducing a long document into a shorter version while preserving the most important information and meaning. In modern NLP, neural abstractive summarization uses deep learning models that generate new sentences rather than simply extracting existing ones.

Here we use a T5(Text to Text transformer model)
The T5 model (Text-to-Text Transfer Transformer) treats summarization as:

- Input: "summarize: <original text>"
- Output: "<generated summary>"

It is an encoder-decoder Transformer architecture:

- Encoder processes and understands the input text.
- Decoder generates a shorter version based on learned language patterns.
- Uses self-attention to capture relationships between all words in the sequence.
- Learns meaning, context, and semantics from large training corpora.

Steps Involved in Building the Text Summarization NLP Application

1. Collect and Preprocess Input Text
    - Read text from user or file.
    - Clean minor formatting and handle long content if needed.
2. Tokenization
    - Convert raw text into token IDs using a tokenizer.
    - Add task prefix like "summarize: " for T5.
3. Embedding + Encoding
    - Text is encoded into numerical contextual embeddings by the Transformer encoder.
4. Sequence Generation with Decoder
    - Decoder generates a shorter sequence word-by-word using beam search for quality output.
5. Convert Token IDs Back to Text
    - Decode generated output and produce human-readable summary.
6. Display or Save Output
    - Printing the summary or store it in a file (e.g., summary.txt).
7. Evaluation (Optional for reports)
    - Metrics such as ROUGE score or qualitative comparison.

Benefits of Neural Summarization

- Produces fluent and natural summaries
- Understands context and meaning rather than just picking sentences
- Works for different domains without manual rules

Challenges

- Requires computational power
- Limited by maximum token length
- Quality depends on model size and training corpus

NLP-application-code

December 1, 2025

1 NLP application focused on text summarization

```
[1]: from transformers import T5ForConditionalGeneration, T5Tokenizer
```

```
/Users/tushya/programs/college/nlp/.venv/lib/python3.10/site-
packages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please update
jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

1.1 Using the pre-trained T5 model

T5, or Text-to-Text Transfer Transformer, is a series of large language models developed by Google AI that treats all natural language processing tasks as text-to-text problems. It uses an encoder-decoder architecture to process input text and generate output text, making it versatile for tasks like translation, summarization, and question answering.

```
[13]: model_name = "t5-base"
      tokenizer = T5Tokenizer.from_pretrained(model_name)
      model = T5ForConditionalGeneration.from_pretrained(model_name)
```

1.1.1 Embed the given document and query on it using the summarize keyword

```
[14]: def summarize(text, max_length=100, min_length=30):
          # T5 expects the task prefix "summarize: "
          input_text = "summarize: " + text.strip().replace("\n", " ")
          inputs = tokenizer.encode(input_text, return_tensors="pt", truncation=True)

          # Generate summary
          summary_ids = model.generate(
              inputs,
              max_length=max_length,
              min_length=min_length,
              length_penalty=2.0,
              num_beams=4,
              early_stopping=True,
          )
```

```
    # Decode and return summary
    summary = tokenizer.decode(summary_ids[0], skip_special_tokens=True)
    return summary
```

```
[15]: long_text = """
        Natural Language Processing (NLP) is a field of artificial intelligence␣
  ↪that focuses on
        the interaction between computers and humans through natural language.␣
  ↪The ultimate objective
        of NLP is to read, decipher, understand, and make sense of human␣
  ↪languages in a manner that
        is valuable. Many challenges in NLP involve natural language␣
  ↪understanding, natural language
        generation, and sentiment analysis.
        """
print("==== Long Text ====")
summary = summarize(long_text)
print("Summary:\n", summary, "\n")
```

```
Asking to truncate to max_length but no maximum length is provided and the model
has no predefined maximum length. Default to no truncation.
```

```
==== Long Text ====
Summary:
 natural language processing (NLP) is a field of artificial intelligence . it
focuses on the interaction between computers and humans through natural language
. many challenges in NLP involve natural language understanding and sentiment
analysis .
```

```
[16]: from pathlib import Path

for file in Path(".").glob("*.txt"):
    with open(file, "r") as f:
        print(f"==== File {file.name} ====")
        summary = summarize(f.read())
        print("Summary:\n", summary, "\n")
```

```
==== File linux.txt ====
Summary:
 Linux is a family of open source Unix-like operating systems based on the Linux
kernel . it is designed as a clone of Unix and released under the copyleft GPL
license . Linux is the predominant operating system for servers and is used on
all of the world's 500 fastest supercomputers .

==== File stockfish.txt ====
Summary:
 stockfish is a free and open-source chess engine . it can be used in chess
```

```
software through the Universal Chess Interface . stockfish is the strongest CPU
chess engine in the world .

==== File unix.txt ====
Summary:
 UNIX is a family of multi-tasking, multi-user computer operating systems . it
was originally intended for use inside the Bell System, but was later
commercialized . in the 1980s, it became the operating system of choice for over
90% of the world's top 500 supercomputers .
```