Tab 1

6 7

# Step 1: Import required libraries

from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn.linear\_model import LogisticRegression from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder from sklearn.metrics import classification\_report

# Step 2: Sample labeled sentiment dataset data = [

("I love this product, it's amazing!", "positive"), ("Worst experience I've ever had.", "negative"), ("The service was okay, not great.", "neutral"),

("Absolutely fantastic! Highly recommend.", "positive"), ("Terrible food, will not come back.", "negative"),

("It's fine, not too bad, not too good.", "neutral"), ("Very satisfied with the performance.", "positive"), ("This is disappointing.", "negative")

from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn.decomposition import NMF

import numpy as np

# Sample documents documents = [

"The economy is growing steadily", "Market conditions have improved",

"The financial crisis affected global economy", "New tech startups are emerging rapidly",

"AI and machine learning are transforming industries", "Quantum computing is a promising new field",

]

# Step 1: TF-IDF Vectorization

vectorizer = TfidfVectorizer(stop\_words='english') X = vectorizer.fit\_transform(documents)

]

# Step 3: Separate texts and labels texts = [t[0] for t in data]

labels = [t[1] for t in data]

# Step 4: Vectorize text using TF-IDF

vectorizer = TfidfVectorizer(lowercase=True, stop\_words='english') X = vectorizer.fit\_transform(texts)

# Step 5: Encode labels (text → numbers) encoder = LabelEncoder()

y = encoder.fit\_transform(labels)

# Step 6: Split into train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=42)

# Step 7: Train the log-linear model (Logistic Regression) model = LogisticRegression(max\_iter=200) model.fit(X\_train, y\_train)

# Step 8: Evaluate the model y\_pred = model.predict(X\_test) print("Classification Report:\n")

print(classification\_report(y\_test, y\_pred, target\_names=encoder.classes\_))

# Step 9: Predict sentiment on new text

new\_text = ["The experience was delightful and smooth."] X\_new = vectorizer.transform(new\_text)

predicted\_label = encoder.inverse\_transform(model.predict(X\_new))[0] print("Predicted Sentiment for new text:", predicted\_label)

—

import spacy

nlp = spacy.load("en\_core\_web\_sm")

text = """Deepak Jasani, Head of retail research, HDFC Securities, said: “Investors will look

to the European Central Bank later Thursday for reassurance that surging prices are just transitory, and not about to spiral out of control. In addition to the ECB policy meeting, investors are awaiting a report later Thursday on US economic growth, which is likely to show a cooling recovery, as well as weekly jobs data.”"""

doc = nlp(text)

for ent in doc.ents:

print(f"{ent.text:<30} --> {ent.label\_}")

# Step 2: Apply NMF

n\_topics = 2 # You can change this

nmf\_model = NMF(n\_components=n\_topics, random\_state=42) W = nmf\_model.fit\_transform(X)

H = nmf\_model.components\_

# Step 3: Show topics (top words per topic) feature\_names = vectorizer.get\_feature\_names\_out() for topic\_idx, topic in enumerate(H):

top\_words = [feature\_names[i] for i in topic.argsort()[:-6:-1]] print(f"Topic {topic\_idx+1}: {', '.join(top\_words)}")

# Step 4: Evaluate with Reconstruction Error reconstruction = np.dot(W, H)

error = np.linalg.norm(X.toarray() - reconstruction) print("\nReconstruction Error:", error)

—------------------------------------------------------------------------------------------------------

from nltk.wsd import lesk

from nltk.corpus import wordnet as wn from nltk.tokenize import word\_tokenize import nltk

# Ensure necessary data is downloaded nltk.download('wordnet') nltk.download('omw-1.4') nltk.download('punkt')

# Sentence containing ambiguous word sentence = "I went to the bank to deposit money" ambiguous\_word = "bank"

# Apply simplified Lesk algorithm context = word\_tokenize(sentence) sense = lesk(context, ambiguous\_word)

# Display the sense

print(f"\nBest sense for '{ambiguous\_word}': {sense}") print("Definition:", sense.definition()) print("Synonyms:", sense.lemma\_names())

9

|  |  |
| --- | --- |
| from nltk import ngrams  from collections import Counter  # Sample corpus  texts = ['the quick brown fox', 'the slow brown dog', 'the quick red dog',  'the lazy yellow fox']  # Function to get n-grams def get\_ngrams(texts, n=2):  all\_ngrams = [] for text in texts:  tokens = text.lower().split() n\_grams = list(ngrams(tokens, n)) all\_ngrams.extend(n\_grams)  return Counter(all\_ngrams)  # Bigrams  bigrams = get\_ngrams(texts, n=2) print("Top Bigrams:\n", bigrams)  # Trigrams  trigrams = get\_ngrams(texts, n=3) print("\nTop Trigrams:\n", trigrams)  — | # topic\_modeling.py  # Step 1: Imports  from sklearn.feature\_extraction.text import CountVectorizer, TfidfVectorizer from sklearn.decomposition import LatentDirichletAllocation, TruncatedSVD  # Step 2: Define your corpus corpus = [  'the quick brown fox', 'the slow brown dog', 'the quick red dog', 'the lazy yellow fox'  ]  # Step 3: Vectorize the corpus # - Count Vectorizer for LDA # - TF-IDF Vectorizer for LSA count\_vec = CountVectorizer() tfidf\_vec = TfidfVectorizer()  count\_matrix = count\_vec.fit\_transform(corpus) tfidf\_matrix = tfidf\_vec.fit\_transform(corpus)  count\_features = count\_vec.get\_feature\_names\_out() tfidf\_features = tfidf\_vec.get\_feature\_names\_out()  # Step 4: Fit LDA model n\_topics = 2  lda = LatentDirichletAllocation(n\_components=n\_topics, random\_state=42) lda.fit(count\_matrix)  print("🔹 Topics from LDA:")  for topic\_idx, topic in enumerate(lda.components\_): top\_indices = topic.argsort()[-5:][::-1]  top\_terms = [count\_features[i] for i in top\_indices] print(f"Topic #{topic\_idx+1}: {', '.join(top\_terms)}")  # Step 5: Fit LSA model (via truncated SVD on TF-IDF)  lsa = TruncatedSVD(n\_components=n\_topics, random\_state=42) lsa.fit(tfidf\_matrix)  print("\n🔹 Topics from LSA:")  for topic\_idx, comp in enumerate(lsa.components\_): top\_indices = comp.argsort()[-5:][::-1]  top\_terms = [tfidf\_features[i] for i in top\_indices] print(f"Topic #{topic\_idx+1}: {', '.join(top\_terms)}") |

1 2

|  |  |
| --- | --- |
| import nltk  from nltk.tokenize import word\_tokenize, TreebankWordTokenizer,  TweetTokenizer, MWETokenizer  from nltk.stem import PorterStemmer, SnowballStemmer from nltk.stem import WordNetLemmatizer  # Download necessary resources nltk.download('punkt') nltk.download('wordnet') nltk.download('omw-1.4')  # Sample text  text = "I'm learning NLP! NLTK's tools like tokenizers, stemmers, and lemmatizers are useful."  # 1. Tokenization  # Whitespace-based whitespace\_tokens = text.split()  # Punctuation-based using word\_tokenize punct\_tokens = word\_tokenize(text)  # Treebank tokenizer  treebank = TreebankWordTokenizer() treebank\_tokens = treebank.tokenize(text)  # Tweet tokenizer  tweet\_tokenizer = TweetTokenizer() tweet\_tokens = tweet\_tokenizer.tokenize(text)  # Multi-Word Expression Tokenizer (custom MWE)  mwe\_tokenizer = MWETokenizer([('natural', 'language'), ('machine', 'learning')]) mwe\_text = "I love natural language processing and machine learning." mwe\_tokens = mwe\_tokenizer.tokenize(mwe\_text.split())  # 2. Stemming  porter = PorterStemmer()  snowball = SnowballStemmer("english")  porter\_stems = [porter.stem(word) for word in punct\_tokens] snowball\_stems = [snowball.stem(word) for word in punct\_tokens]  # 3. Lemmatization  lemmatizer = WordNetLemmatizer()  lemmas = [lemmatizer.lemmatize(word) for word in punct\_tokens]  # Print all outputs  print("Whitespace Tokenization:", whitespace\_tokens) print("Punctuation Tokenization:", punct\_tokens) print("Treebank Tokenization:", treebank\_tokens) print("Tweet Tokenization:", tweet\_tokens) print("MWE Tokenization:", mwe\_tokens)  print("\nPorter Stemmer:", porter\_stems) print("Snowball Stemmer:", snowball\_stems) print("Lemmatization:", lemmas) | # text\_vectorization\_word2vec.py  import subprocess import sys  # Auto-install required packages def install(package):  subprocess.check\_call([sys.executable, "-m", "pip", "install", package])  try:  import nltk except ImportError:  install("nltk") import nltk  try:  import gensim except ImportError:  install("gensim") import gensim  try:  import sklearn except ImportError:  install("scikit-learn")  from sklearn.feature\_extraction.text import CountVectorizer, TfidfVectorizer from sklearn.preprocessing import normalize  else:  from sklearn.feature\_extraction.text import CountVectorizer, TfidfVectorizer from sklearn.preprocessing import normalize  from gensim.models import Word2Vec import numpy as np  import nltk nltk.download('punkt')  # -------------------------------  # Main Processing Script # -------------------------------  # Sample corpus corpus = [  "Natural language processing is fascinating.", "I love learning about NLP.",  "Gensim helps in building Word2Vec models.",  "TF-IDF and Bag-of-Words are vectorization techniques."  ]  # 1. Bag of Words (Count Vectorizer) count\_vectorizer = CountVectorizer()  count\_matrix = count\_vectorizer.fit\_transform(corpus) print("🔹 Count Vectorizer (BoW):") print(count\_matrix.toarray())  print("Vocabulary:", count\_vectorizer.get\_feature\_names\_out())  # 2. Normalized Count  norm\_count = normalize(count\_matrix, norm='l1', axis=1) print("\n🔹 Normalized Count Matrix (L1 Norm):") print(norm\_count.toarray())  # 3. TF-IDF Vectorizer tfidf\_vectorizer = TfidfVectorizer()  tfidf\_matrix = tfidf\_vectorizer.fit\_transform(corpus) print("\n🔹 TF-IDF Matrix:") print(tfidf\_matrix.toarray())  print("TF-IDF Features:", tfidf\_vectorizer.get\_feature\_names\_out()) |

# 4. Word2Vec Embeddings

# Preprocess corpus into tokens

tokenized\_corpus = [nltk.word\_tokenize(sent.lower()) for sent in corpus]

# Train Word2Vec

w2v\_model = Word2Vec(sentences=tokenized\_corpus, vector\_size=50, window=3, min\_count=1, workers=2, sg=1)

# Display word vectors for a sample word print("\n🔹 Word2Vec vector for 'nlp':")

if 'nlp' in w2v\_model.wv: print(w2v\_model.wv['nlp'])

else:

print("Word 'nlp' not in vocabulary.")

3 4

|  |  |
| --- | --- |
| # text\_cleaning\_tfidf\_label.py  import subprocess import sys  # Auto-install dependencies def install(package):  subprocess.check\_call([sys.executable, "-m", "pip", "install", package])  for pkg in ["nltk", "scikit-learn", "pandas"]:  try:  import (pkg) except ImportError:  install(pkg)  # Imports import nltk import re  import pandas as pd  from sklearn.feature\_extraction.text import TfidfVectorizer from sklearn.preprocessing import LabelEncoder  nltk.download('punkt') nltk.download('wordnet') nltk.download('stopwords')  from nltk.corpus import stopwords  from nltk.tokenize import word\_tokenize from nltk.stem import WordNetLemmatizer  # Sample text dataset data = [  ("Natural language processing is amazing!", "positive"), ("I hate spam emails and junk messages", "negative"), ("Lemmatization helps reduce word forms", "positive"), ("This is the worst model ever", "negative"),  ]  texts, labels = zip(\*data)  # 1. Clean text, lemmatize, remove stopwords stop\_words = set(stopwords.words("english")) lemmatizer = WordNetLemmatizer()  def clean\_text(text):  text = re.sub(r'[^a-zA-Z ]', '', text) # Remove punctuation/numbers tokens = word\_tokenize(text.lower())  filtered = [lemmatizer.lemmatize(w) for w in tokens if w not in stop\_words] return " ".join(filtered)  cleaned\_texts = [clean\_text(t) for t in texts] print("🔹 Cleaned Texts:")  for t in cleaned\_texts:  print(t)  # 2. Label Encoding label\_encoder = LabelEncoder()  encoded\_labels = label\_encoder.fit\_transform(labels) print("\n🔹 Encoded Labels:", encoded\_labels)  # 3. TF-IDF Representation tfidf = TfidfVectorizer()  tfidf\_matrix = tfidf.fit\_transform(cleaned\_texts)  # Convert to DataFrame  tfidf\_df = pd.DataFrame(tfidf\_matrix.toarray(), columns=tfidf.get\_feature\_names\_out()) | # transformer\_from\_scratch.py  import math import torch  import torch.nn as nn  # Positional Encoding  class PositionalEncoding(nn.Module):  def init (self, d\_model, max\_len=5000):  super(). init ()  pe = torch.zeros(max\_len, d\_model)  position = torch.arange(0, max\_len).unsqueeze(1) div\_term = torch.exp(  torch.arange(0, d\_model, 2) \* (-math.log(10000.0) / d\_model)  )  pe[:, 0::2] = torch.sin(position \* div\_term) pe[:, 1::2] = torch.cos(position \* div\_term)  pe = pe.unsqueeze(0) self.register\_buffer("pe", pe)  def forward(self, x):  return x + self.pe[:, :x.size(1)]  # Multi-head Attention  class MultiHeadAttention(nn.Module):  def init (self, d\_model, num\_heads):  super(). init ()  assert d\_model % num\_heads == 0  self.d\_k = d\_model // num\_heads self.num\_heads = num\_heads  self.q\_linear = nn.Linear(d\_model, d\_model) self.k\_linear = nn.Linear(d\_model, d\_model) self.v\_linear = nn.Linear(d\_model, d\_model) self.out = nn.Linear(d\_model, d\_model)  def forward(self, q, k, v, mask=None):  batch\_size = q.size(0)  def transform(x, linear):  x = linear(x)  x = x.view(batch\_size, -1, self.num\_heads, self.d\_k) return x.transpose(1, 2)  q = transform(q, self.q\_linear) k = transform(k, self.k\_linear) v = transform(v, self.v\_linear)  scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(self.d\_k) if mask is not None:  scores = scores.masked\_fill(mask == 0, -1e9)  attn = torch.softmax(scores, dim=-1) context = torch.matmul(attn, v)  context = context.transpose(1, 2).contiguous().view(batch\_size, -1, self.num\_heads \* self.d\_k)  return self.out(context)  # Feedforward Network  class FeedForward(nn.Module):  def init (self, d\_model, d\_ff=2048): |

|  |  |
| --- | --- |
| tfidf\_df['Label'] = encoded\_labels  # 4. Save outputs tfidf\_df.to\_csv("tfidf\_output.csv", index=False)  print("\n✅ TF-IDF with labels saved to tfidf\_output.csv") | super(). init ()  self.linear1 = nn.Linear(d\_model, d\_ff) self.relu = nn.ReLU()  self.linear2 = nn.Linear(d\_ff, d\_model)  def forward(self, x):  return self.linear2(self.relu(self.linear1(x)))  # Encoder Layer  class TransformerEncoderLayer(nn.Module):  def init (self, d\_model, num\_heads):  super(). init ()  self.attn = MultiHeadAttention(d\_model, num\_heads) self.ff = FeedForward(d\_model)  self.norm1 = nn.LayerNorm(d\_model) self.norm2 = nn.LayerNorm(d\_model) self.dropout = nn.Dropout(0.1)  def forward(self, x, mask=None):  x2 = self.attn(x, x, x, mask)  x = self.norm1(x + self.dropout(x2)) x2 = self.ff(x)  x = self.norm2(x + self.dropout(x2)) return x  # Transformer Encoder Model  class TransformerEncoder(nn.Module):  def init (self, input\_dim, d\_model, num\_heads, num\_layers, max\_len=100): super(). init ()  self.embedding = nn.Embedding(input\_dim, d\_model) self.pos\_encoder = PositionalEncoding(d\_model, max\_len) self.layers = nn.ModuleList([  TransformerEncoderLayer(d\_model, num\_heads) for \_ in range(num\_layers)  ])  self.out = nn.Linear(d\_model, input\_dim)  def forward(self, src, mask=None):  x = self.embedding(src) \* math.sqrt(self.embedding.embedding\_dim) x = self.pos\_encoder(x)  for layer in self.layers:  x = layer(x, mask) return self.out(x)  # Sample usage  if name == " main ":  vocab\_size = 1000  model\_dim = 64  num\_heads = 8  num\_layers = 2  seq\_len = 10  batch\_size = 2  model = TransformerEncoder(vocab\_size, model\_dim, num\_heads, num\_layers) dummy\_input = torch.randint(0, vocab\_size, (batch\_size, seq\_len))  output = model(dummy\_input) print("Transformer output shape:", output.shape) |

5 8

|  |  |
| --- | --- |
| # parsing\_techniques.py  import nltk  from nltk import pos\_tag, word\_tokenize, RegexpParser from nltk.chunk import ne\_chunk  # Ensure required NLTK resources are downloaded nltk.download('punkt')  nltk.download('punkt\_tab') nltk.download('averaged\_perceptron\_tagger') nltk.download('averaged\_perceptron\_tagger\_eng') nltk.download('maxent\_ne\_chunker') nltk.download('maxent\_ne\_chunker\_tab') nltk.download('words')  # Sample text  text = "Barack Obama was the 44th President of the United States and lives in Washington."  # Tokenize and POS tag tokens = word\_tokenize(text) pos\_tags = pos\_tag(tokens)  print("\n🔹 Part-of-Speech Tags:") print(pos\_tags)  # --- 1. Shallow Parsing (Chunking using Noun Phrases) --- chunk\_grammar = "NP: {<DT>?<JJ>\*<NN.\*>+}" # Noun Phrase chunk rule chunk\_parser = RegexpParser(chunk\_grammar)  shallow\_tree = chunk\_parser.parse(pos\_tags)  print("\n🔹 Shallow Parsing (Chunking Tree):") print(shallow\_tree)  # --- 2. Named Entity Recognition using ne\_chunk (Built-in Shallow Parser) --- ner\_tree = ne\_chunk(pos\_tags)  print("\n🔹 Named Entities (from ne\_chunk):") print(ner\_tree)  # --- 3. Regex Parser Example (Custom Verb Phrase Extraction) ---  # Regex Grammar: VP -> Verb followed by NP or Verb followed by PP vp\_grammar = r"""  VP: {<VB.\*><NP|PP|CLAUSE>+$} # Verb Phrase NP: {<DT>?<JJ>\*<NN.\*>+} # Noun Phrase PP: {<IN><NP>} # Prepositional Phrase CLAUSE: {<NP><VP>} # Sub-Clause  """  regex\_parser = RegexpParser(vp\_grammar) regex\_tree = regex\_parser.parse(pos\_tags)  print("\n🔹 Regex-based Parsing Tree:") print(regex\_tree)  # Optional: Visualize trees (commented out for headless environments) # shallow\_tree.draw()  # regex\_tree.draw() # ner\_tree.draw() | # covid\_word\_embeddings.py  import nltk  import pandas as pd import numpy as np  from nltk.tokenize import word\_tokenize from nltk.corpus import stopwords  from gensim.models import FastText  from sklearn.feature\_extraction.text import TfidfVectorizer import gensim.downloader as api  import os import re import string  # Download necessary resources nltk.download('punkt') nltk.download('stopwords')  # Load a sample COVID-19 dataset (replace with your own file if needed) # For demonstration, using dummy data  corpus = [  "COVID-19 is caused by the novel coronavirus.", "Vaccines help prevent the spread of COVID-19.", "Social distancing is important during the pandemic.", "Masks reduce the risk of transmission.",  ]  # Preprocessing function def preprocess(text):  text = text.lower()  text = re.sub(f"[{re.escape(string.punctuation)}]", "", text) tokens = word\_tokenize(text)  stop\_words = set(stopwords.words('english'))  tokens = [t for t in tokens if t not in stop\_words and t.isalpha()] return tokens  # Preprocess all sentences  tokenized\_corpus = [preprocess(doc) for doc in corpus]  # Train FastText embeddings  fasttext\_model = FastText(sentences=tokenized\_corpus, vector\_size=100, window=5, min\_count=1, workers=4, epochs=10)  # Save FastText model fasttext\_model.save("covid\_fasttext.model")  # Load pre-trained GloVe vectors from gensim  glove\_vectors = api.load("glove-wiki-gigaword-100") # 100d GloVe  # Function to get document vector using GloVe def get\_glove\_vector(doc):  vectors = [glove\_vectors[word] for word in doc if word in glove\_vectors] if vectors:  return np.mean(vectors, axis=0) else:  return np.zeros(100)  # Create GloVe-based document embeddings  glove\_doc\_vectors = np.array([get\_glove\_vector(doc) for doc in tokenized\_corpus])  # Save GloVe vectors np.save("covid\_glove\_doc\_vectors.npy", glove\_doc\_vectors)  # Optional: Print sample vectors  print("Sample FastText Word Vector (e.g., 'covid'):") print(fasttext\_model.wv["covid"] if "covid" in fasttext\_model.wv else "Word not |

found in vocab")

print("\nSample GloVe Document Vector (Doc 1):") print(glove\_doc\_vectors[0])

pip install nltk gensim numpy scikit-learn

10

pip install transformers datasets scikit-learn torch nltk gensim

# transformer\_classification\_lda.py import torch

from transformers import AutoTokenizer, AutoModelForSequenceClassification, Trainer, TrainingArguments from datasets import Dataset

from sklearn.model\_selection import train\_test\_split from sklearn.metrics import accuracy\_score

import numpy as np

from gensim import corpora, models import nltk

from nltk.tokenize import word\_tokenize import re

import string nltk.download('punkt')

#

# 1. TEXT CLASSIFICATION WITH TRANSFORMER #

# Dummy classification data (replace with your own) texts = [

"I love machine learning!", "This movie was terrible...", "The food was delicious.", "I'm not feeling well today.", "Transformers are amazing!", "I hate being ignored.",

"He is very kind.",

"That was the worst day ever.", "I'm so happy for you!",

"This app is a disaster.",

]

labels = [1, 0, 1, 0, 1, 0, 1, 0, 1, 0] # 1 = positive, 0 = negative

# Train/test split

train\_texts, val\_texts, train\_labels, val\_labels = train\_test\_split(texts, labels, test\_size=0.2)

# Load tokenizer and model model\_name = "distilbert-base-uncased"

tokenizer = AutoTokenizer.from\_pretrained(model\_name)

# Tokenize

def tokenize(batch):

return tokenizer(batch["text"], padding=True, truncation=True)

# Prepare datasets

train\_dataset = Dataset.from\_dict({"text": train\_texts, "label": train\_labels}) val\_dataset = Dataset.from\_dict({"text": val\_texts, "label": val\_labels}) train\_dataset = train\_dataset.map(tokenize, batched=True)

val\_dataset = val\_dataset.map(tokenize, batched=True) train\_dataset.set\_format("torch", columns=["input\_ids", "attention\_mask", "label"]) val\_dataset.set\_format("torch", columns=["input\_ids", "attention\_mask", "label"])

# Model

model = AutoModelForSequenceClassification.from\_pretrained(model\_name, num\_labels=2)

# Metrics

def compute\_metrics(eval\_pred):

logits, labels = eval\_pred

predictions = np.argmax(logits, axis=-1)

return {"accuracy": accuracy\_score(labels, predictions)}

# Trainer setup

training\_args = TrainingArguments( output\_dir="./results", evaluation\_strategy="epoch", per\_device\_train\_batch\_size=2, per\_device\_eval\_batch\_size=2, num\_train\_epochs=2, logging\_dir="./logs", logging\_steps=5, load\_best\_model\_at\_end=True, save\_total\_limit=1,

)

trainer = Trainer( model=model, args=training\_args, train\_dataset=train\_dataset, eval\_dataset=val\_dataset,

compute\_metrics=compute\_metrics,

)

# Train the model trainer.train()

#

# 2. TOPIC MODELING USING LDA #

lda\_corpus = [

"The cat sat on the mat.", "Dogs are great pets.",

"I love to play football.",

"Data science is an interdisciplinary field.", "Python is a great programming language.",

"Machine learning is a subset of artificial intelligence.", "Artificial intelligence and machine learning are popular topics.", "Deep learning is a type of machine learning.",

"Natural language processing involves analyzing text data.", "I enjoy hiking and outdoor activities."

]

def preprocess\_lda(doc):

doc = doc.lower()

doc = re.sub(f"[{re.escape(string.punctuation)}]", "", doc)

return word\_tokenize(doc)

tokenized\_docs = [preprocess\_lda(doc) for doc in lda\_corpus] dictionary = corpora.Dictionary(tokenized\_docs)

corpus = [dictionary.doc2bow(text) for text in tokenized\_docs]

# LDA Model

lda\_model = models.LdaModel(corpus, num\_topics=3, id2word=dictionary, passes=10)

print("\nTop Topics from LDA:")

for idx, topic in lda\_model.print\_topics(-1): print(f"Topic {idx + 1}: {topic}")