Tab 1

6

```
# 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")
]

# 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_}")
```

7

```
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 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

```
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)
```

2

```
# 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

```
# 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())
```

4

```
# 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

```
# 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()
```

8

```
# 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}")
```