National College of Ireland

Enhancing Legal Guidance by Utilizing Natural Language Processing-Based Document Embeddings

MSc Research Project
MSc Artificial Intelligence

Tanmay Laxmikant Mukim
Student ID: 22127933

School of Computing
National College of Ireland

Supervisor: Dr. Muslim Jamal Syed

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Tanmay Laxmikant Mukim
Student ID:	22127933
Programme:	MSc Artificial Intelligence
Year:	2023
Module:	MSc Research Project
Supervisor:	Dr. Muslim Jamal Syed
Submission Due Date:	14/12/2023
Project Title:	Enhancing Legal Guidance by Utilizing Natural Language Processing-Based Document Embeddings
Word Count:	619
Page Count:	14

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	29th January 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Enhancing Legal Guidance by Utilizing Natural Language Processing-Based Document Embeddings

Tanmay Laxmikant Mukim
22127933

1 Introduction

The procedures for improving legal guidance through the utilization of document embeddings based on natural language processing are outlined in this setup manual. The procedures for executing the code artifact, which comprises the Python coding and associated processes, are as follows:

2 The configuration of the hardware

Here are the details about certain devices as well as Windows specifications to support the code artifact, as illustrated in Figures 1 and 2, respectively.

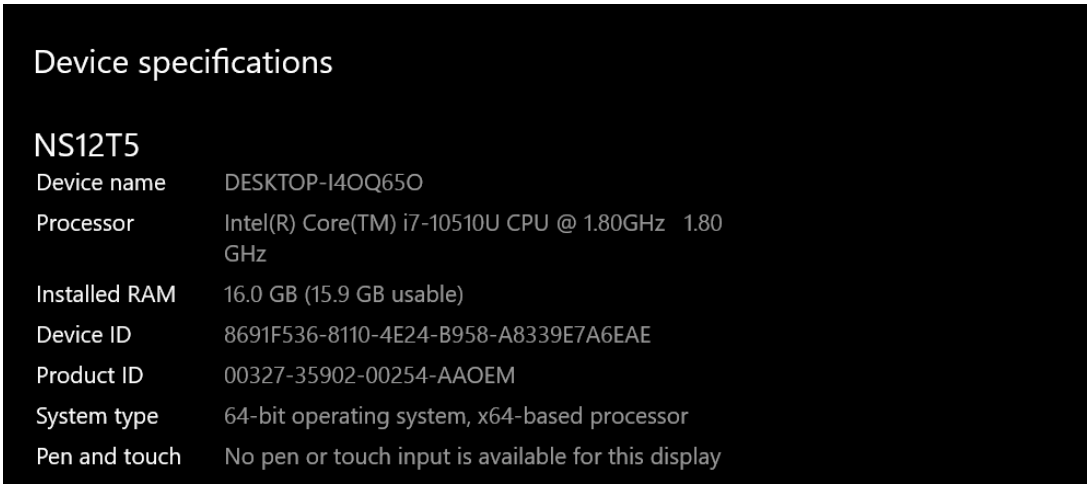

Figure 1: Design specifications

Figure 2 displays the Windows specification because the project is being completed and executed on the Windows 10 operating framework. The device specification includes an Intel (R) Core(TM) I7 10th generation processor and 16 GB of RAM.

3 Program configuration

The Anaconda Rolon-Mérette et al. (2016) Python Jupyter Notebook is utilized for coding implementation and execution. The respective installed version of Python is 3.11. The

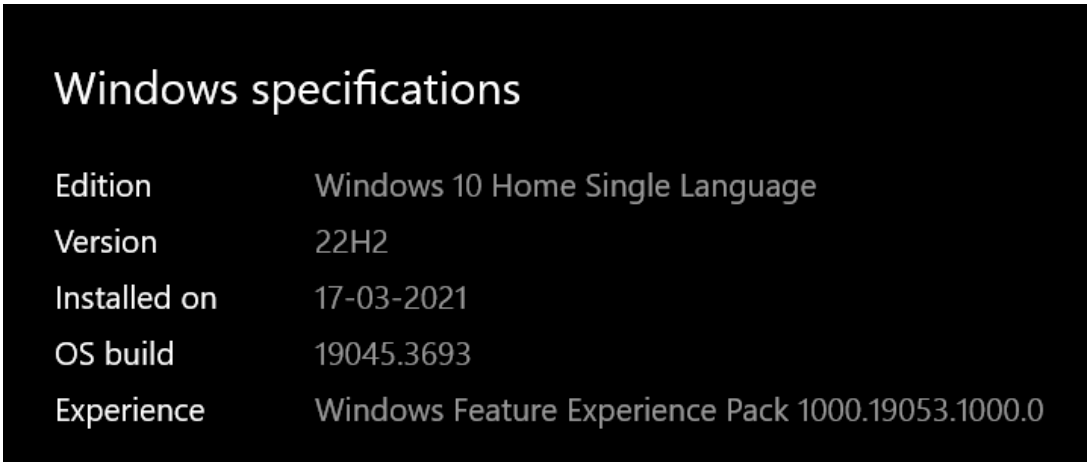

Figure 2: Windows specifications

installation instructions for Python, as well as Jupyter Notebook, are provided in the following subsections.

3.1 Anaconda Python Setup

Python versions and package versions can be easily created in an environment with the aid of Anaconda software. The Jupiter Notebook is one of the packages available with Anaconda. The first step in installing Anaconda Python on Windows is to go to `https://www.anaconda.com` and click on the download button. Anaconda is available for Windows in a multitude of versions. Following the download of the anaconda.exe file, the Python package can then be installed on the device by following the instructions. the following figures: 3 and 4 are the representations of instructions to install Anaconda Python, respectively.

3.2 Library Imports

The corresponding implementation code can be run once Anaconda Python and Jupyter Notebook are installed; however, a few libraries need to be installed first. The natural language processing model libraries that are pre-installed in Anaconda Navigator are listed below. The transformer libraries Rothman (2021) are nothing but a collection of pre-trained modules such as Bidirectional Encoder Representations from Transformers (BERT), ALBERT, and DEBERTa. The installation of the transformer libraries is shown in the following figure: 5 In the next step, the necessary libraries must be installed before being imported using Jupyter Notebook. Libraries are imported, as mentioned in Figure 6

4 Implemented Datasets

This corresponding section provides the details of the implemented datasets for this study. The following two datasets are collections of legal citations in textual format, respectively.

1. Shivam Bansal, 2018. Legal Citation Text Classification, Kaggle, Version 1. `https://www.kaggle.com/datasets/shivamb/legal-citation-text-classification`

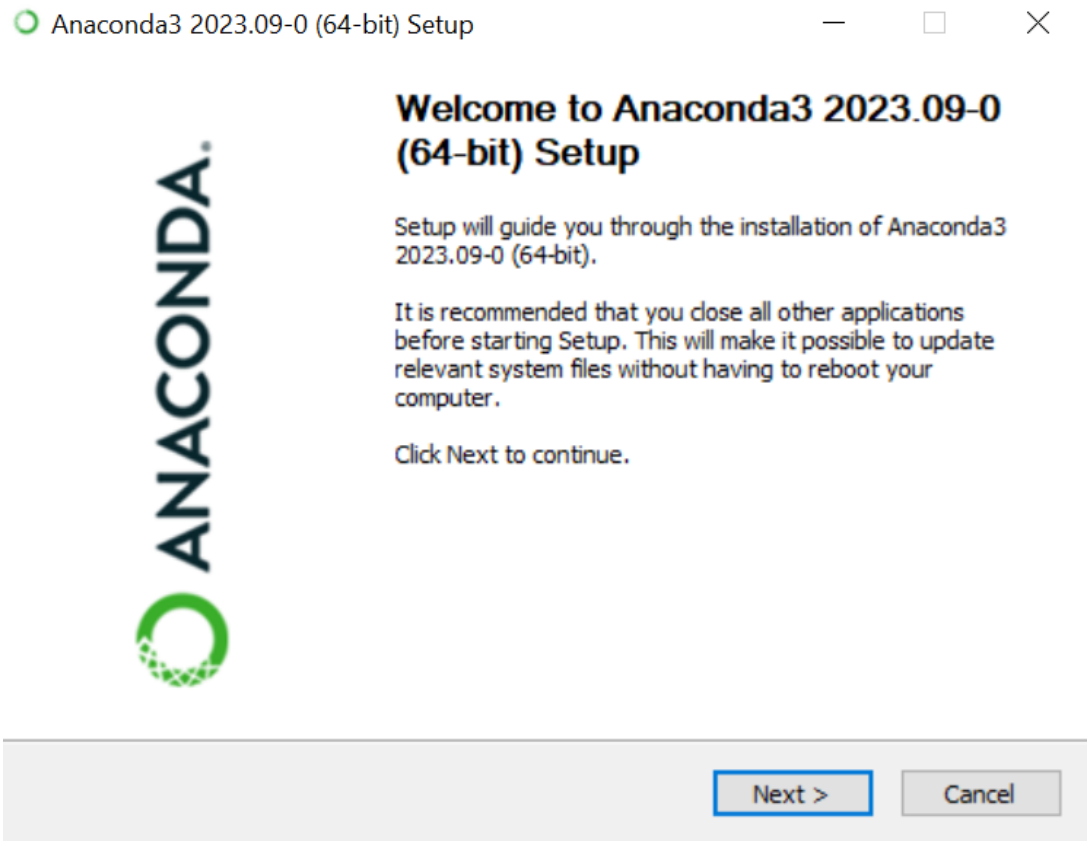

Figure 3: Installing Anaconda Python

2. Washington University in St. Louis (2021). Supreme Court Decisions Text Analysis Kaggle. Version 1. `https://www.kaggle.com/datasets/wustl/supreme-court/code`

5 Data Pre-processing

This section is the representation of the data pre-processing method, which primarily involves Removing Punctuation, Spaces, and Sentence segments and turning them into a list, as well as creating term frequency (inverse document frequency), as mentioned in the following figures: 7 and 8

6 Natural Language Processing Models

The following four natural language processing models, Word2Vec in combination with TF-IDF 9 with Bidirectional Encoder Representations from Transformers (BERT) 10, ALBERT 11, together with DEBERTa 12, are implemented concerning the two legal textual citation datasets, as mentioned as follows:

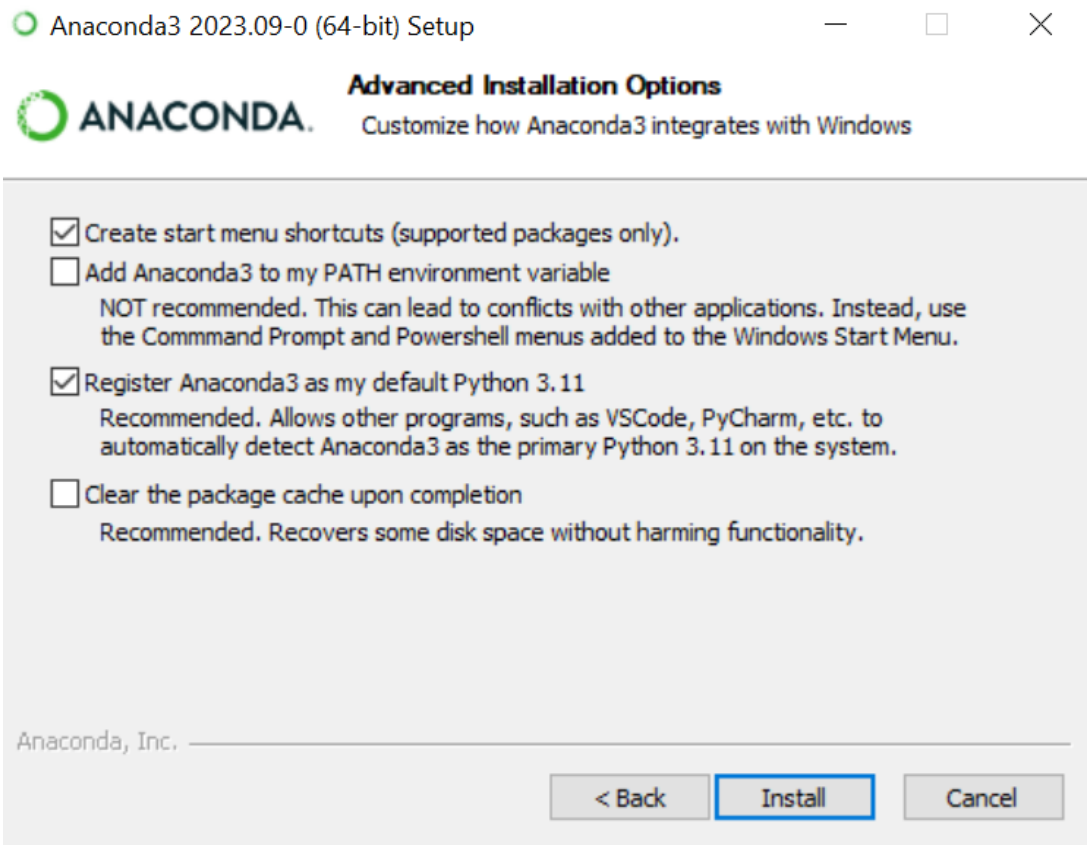

Figure 4: Installing Anaconda Python

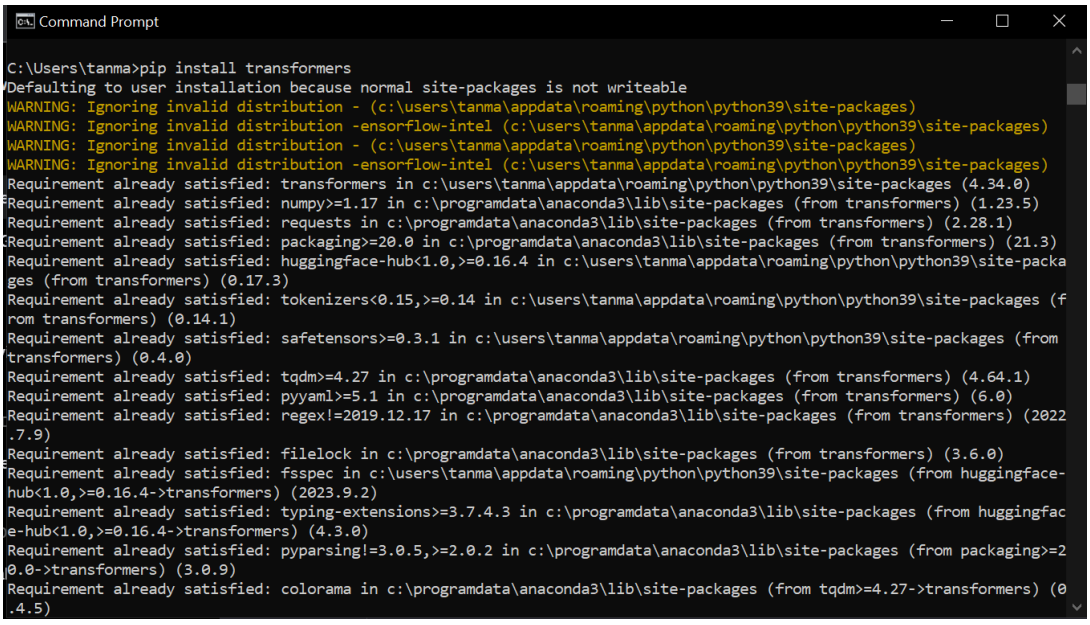

Figure 5: Installing Transformeres

Importing Labraries

```
In [2]: import pandas as pd
        import numpy as np
        from transformers import BertTokenizer, BertModel
        import torch
        from gensim.models import Word2Vec
        from sklearn.feature_extraction.text import TfidfVectorizer
        from sklearn.metrics.pairwise import cosine_similarity
        import re
        import datetime
        import seaborn as sns
        import matplotlib.pyplot as Plot
        import pickle
        from transformers import BertTokenizer, BertForNextSentencePrediction, AdamW
        from sklearn.model_selection import train_test_split
        from torch.utils.data import DataLoader, TensorDataset, random_split
        from torch.nn import functional as F
        from transformers import AlbertTokenizer, AlbertModel
        from transformers import DebertaTokenizer, DebertaModel
```

Figure 6: Importing Labraries

```
In [5]: Data['case_name'] = Data['case_name'].str.replace('@', ' ')
```

Creating Term Frequency - Inverse Document Frequency

```
In [6]: TFIDFML = TfidfVectorizer()
        TFIDFML.fit(Data.case_name)
        # Creating TF-IDF Dicitonary
        Dictionary = dict(zip(TFIDFML.get_feature_names(), list(TFIDFML.idf_)))
        # Defining Feature Name
        TFIDFfeature = TFIDFML.get_feature_names()
        for term, idf_value in Dictionary.items(): # Printing The Dictonary
            print(f'Term: {term}, IDF Value: {idf_value}')
        print(TFIDFML.fit)
        print(TFIDFfeature)
```

```
Term: 011, IDF Value: 9.382289428951436
Term: 10, IDF Value: 9.382289428951436
Term: 100, IDF Value: 8.97682432084327
Term: 1000, IDF Value: 9.382289428951436
Term: 1001, IDF Value: 9.382289428951436
Term: 1029, IDF Value: 9.382289428951436
Term: 103, IDF Value: 9.382289428951436
Term: 104, IDF Value: 9.382289428951436
Term: 107, IDF Value: 9.382289428951436
Term: 11, IDF Value: 8.68914224839149
```

Figure 7: Creating term frequency (inverse document frequency)

Removing Punctuation, Spaces, Sentence Segment and turning into list

```
In [7]: def PreProcess(Text):
            # Remove punctuation, spaces, and unwanted characters
            Rule = re.compile(r'[^a-zA-Z0-9\u4e00-\u9fa5]')
            Text = Rule.sub(' ', str(Text))
            Text = re.sub(' +', ' ', Text)

            # Split the text into words (tokenization) and return as a list
            Text = Text.split()  # Split text into words based on spaces

            return Text
```

Figure 8: Data Pre-processing

7 Recommendation Engine

The following section represents the implementation of recommendation engines concerning four natural language processing models: Word2Vec in combination with TF-IDF 13 with Bidirectional Encoder Representations from Transformers (BERT) 14 as well as ALBERT 15, together with DEBERTa 16.

In [8]:
```python
# Implementing Data Sclicing Procedure With Respect To 1000 Samples
SubsetSize = 1000
SubsetData = Data.head(SubsetSize).copy()

# Training the Word2Vec model on the basis of the corresponding subset
SubsetSentences = SubsetData['case_name'].tolist()
SubsetSentences = [PreProcess(text) for text in SubsetSentences]

Model = Word2Vec(
    SubsetSentences,
    vector_size=100,
    window=5,
    min_count=1,
    sg=0
)

# Incorporating TF-IDF information into Word2Vec word vectors
for Word in Model.wv.index_to_key:
    WordVector = Model.wv[Word]
    if Word in Dictionary:
        TFIDFWeight = Dictionary[Word]
        WeightedVector = WordVector * TFIDFWeight
        Model.wv[Word] = WeightedVector

# Saving the Word2Vec model
Model.save("word2vec_with_tfidf_1000samples.model")
# Printing the data details
print(len(Data))
print(len(SubsetSentences))

# Loading the Word2Vec model with TF-IDF weighting
Word2VecModel = Word2Vec.load("word2vec_with_tfidf_1000samples.model")
```

```
8737
1000
```

Figure 9: Implementing Word2Vec Natural Language Model

In [13]:

```
# Implementing Data Sclicing Procedure With Respect To 1000 Samples
SubsetSize = 1000

SmallerSubset = Data.head(SubsetSize).copy()

Corpus = SmallerSubset['case_name'].tolist()

# Preprocessing the corpus and feature names for TF-IDF
Corpus = [text.lower() for text in Corpus]  # Converting text to lowercase

# Initializing a BERT tokenizer and model
Tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
BertModel = BertModel.from_pretrained('bert-base-uncased')

# Tokenizing and getting BERT embeddings in smaller batches
BatchSize = 32
MaxLength = 128
EmbeddingsList = []


# Implementing the Helper function for processing a batch and for obtaining
def process_batch(BatchTexts):
    # Preprocessing batch texts to lowercase
    BatchTexts = [text.lower() for text in BatchTexts]

    inputs = Tokenizer(
        BatchTexts,
        return_tensors='pt',
        padding='max_length',
        max_length=MaxLength,
        truncation=True
    )
    with torch.no_grad():
        BatchEmbeddings = BertModel(**inputs).last_hidden_state

    return BatchEmbeddings


for i in range(0, len(Corpus), BatchSize):
    BatchTexts = Corpus[i:i + BatchSize]
    BatchEmbeddings = process_batch(BatchTexts)
    EmbeddingsList.append(BatchEmbeddings)

# Concatenating embeddings from different batches for the smaller subset
Embeddings = torch.cat(EmbeddingsList, dim=0)

# Printing the data details
print(len(Data))
print(len(Embeddings))
```

```
8737
1000
```

Figure 10: Implementing BERT Natural Language Model

In [19]:

```
# Implementing Data Sclicing Procedure With Respect To 1000 Samples

SubsetSize = 1000

SmallerSubset = Data.head(SubsetSize).copy()

Corpus = SmallerSubset['case_name'].tolist()

# Preprocessing the corpus for TF-IDF
Corpus = [text.lower() for text in Corpus]  # Convert text to lowercase for

# Initializing an ALBERT tokenizer and model
Tokenizer = AlbertTokenizer.from_pretrained('albert-base-v2')
AlbertModel = AlbertModel.from_pretrained('albert-base-v2')

# Tokenizing and getting ALBERT embeddings in smaller batches
BatchSize = 32  # You can adjust this as needed
MaxLength = 128  # You can adjust this as needed
EmbeddingsList = []


# Implementing Helper function to process a batch and obtain ALBERT embeddi
def process_batch(BatchTexts):
    # Preprocessing batch texts to lowercase
    BatchTexts = [text.lower() for text in BatchTexts]

    inputs = Tokenizer(
        BatchTexts,
        return_tensors='pt',
        padding='max_length',
        max_length=MaxLength,
        truncation=True
    )
    with torch.no_grad():
        BatchOutputs = AlbertModel(**inputs)
        BatchEmbeddings = BatchOutputs.last_hidden_state

    return BatchEmbeddings


for i in range(0, len(Corpus), BatchSize):
    BatchTexts = Corpus[i:i + BatchSize]
    BatchEmbeddings = process_batch(BatchTexts)
    EmbeddingsList.append(BatchEmbeddings)

# Concatenating embeddings from different batches for the smaller subset
Embeddings = torch.cat(EmbeddingsList, dim=0)

# Printing the data details
print(len(Data))
print(len(Embeddings))
```

```
8737
1000
```

Figure 11: Implementing ALBERT Natural Language Model)

In [25]:

```
# Implementing Data Sclicing Procedure With Respect To 1000 Samples

SubsetSize = 1000

SmallerSubset = Data.head(SubsetSize).copy()

Corpus = SmallerSubset['case_name'].tolist()

# Preprocessing the corpus for TF-IDF
Corpus = [text.lower() for text in Corpus]  # Convert text to lowercase for

# Initializing a DeBERTa tokenizer and model
Tokenizer = DebertaTokenizer.from_pretrained('microsoft/deberta-base')
DebertaModel = DebertaModel.from_pretrained('microsoft/deberta-base')

# Tokenizing and getting DeBERTa embeddings in smaller batches
BatchSize = 32  # You can adjust this as needed
MaxLength = 128  # You can adjust this as needed
EmbeddingsList = []


# Implementing Helper function to process a batch and obtain DeBERTa embedd
def process_batch(BatchTexts):
    # Preprocessing batch texts to lowercase
    BatchTexts = [text.lower() for text in BatchTexts]

    inputs = Tokenizer(
        BatchTexts,
        return_tensors='pt',
        padding='max_length',
        max_length=MaxLength,
        truncation=True
    )
    with torch.no_grad():
        BatchOutputs = DebertaModel(**inputs)
        BatchEmbeddings = BatchOutputs.last_hidden_state

    return BatchEmbeddings


for i in range(0, len(Corpus), BatchSize):
    BatchTexts = Corpus[i:i + BatchSize]
    BatchEmbeddings = process_batch(BatchTexts)
    EmbeddingsList.append(BatchEmbeddings)

# Concatenating embeddings from different batches for the smaller subset
Embeddings = torch.cat(EmbeddingsList, dim=0)

# Printing the data details
print(len(Data))
print(len(Embeddings))
```

```
8737
1000
```

Figure 12: Implementing DeBERTa Natural Language Model

In [9]:
```python
# Implementing recommendation engine
def recommend_law(Text, Data, model=Word2VecModel):
    # Tokenizing and preprocessing the input text
    Text = PreProcess(Text)

    # Initializing variables to store Similarities and recommended laws
    Similarities = []
    RecommendedLaws = []

    for _, row in Data.iterrows():
        # Tokenizing and preprocessing the legal texts in your dataset
        LegalText = PreProcess(row['case_name'])

        # Calculating the Similarity between the input text and legal text
        Similarity = 0  # Initializing Similarity score to 0
        Count = 0  # Initializing a Count to keep track of common words

        for Word in Text:
            if Word in model.wv and Word in LegalText:
                Similarity += model.wv.n_similarity([Word], LegalText)
                Count += 1

        if Count > 0:
            Similarity /= Count  # Calculating the Average Similarity score

        # Saving the Similarity score and corresponding legal text
        Similarities.append(Similarity)
        RecommendedLaws.append(row['case_name'])

    # Combining the Similarities and recommended laws into a DataFrame
    Recommendationdf = pd.DataFrame({'Legal Text': RecommendedLaws, 'Simila

    # Sorting the DataFrame by Similarity in descending order
    Recommendationdf = Recommendationdf.sort_values(by='Similarity', ascend

    # Returning the top recommended laws
    TopRecommendations = Recommendationdf.head(20)

    return TopRecommendations
```

Figure 13: Implementing Recommendation Engine with Word2Vec Natural Language Model

In [14]:
```python
# Implementing recommendation engine
def recommend_law(Text, Data, Embeddings):
    Similarities = []
    RecommendedLaws = []
    AverageSimilarities = []

    # Tokenizing and preprocessing the input text
    Text = Text.lower()
    inputs1 = Tokenizer(
        [Text],
        return_tensors='pt',
        padding='max_length',
        max_length=MaxLength,
        truncation=True
    )
    with torch.no_grad():
        Embeddings1 = BertModel(**inputs1).last_hidden_state

    for i in range(len(Embeddings)):
        LegalText = Data['case_name'].iloc[i].lower()
        inputs2 = Tokenizer(
            [LegalText],
            return_tensors='pt',
            padding='max_length',
            max_length=MaxLength,
            truncation=True
        )
        with torch.no_grad():
            Embeddings2 = Embeddings[i]  # Utilizing the precomputed DeBER

        # Calculating cosine similarity between embeddings
        similarity = torch.nn.functional.cosine_similarity(Embeddings1, Emb

        Similarities.append(similarity)
        RecommendedLaws.append(Data['case_name'].iloc[i])
        # Calculating average similarity for each interaction
        AverageSimilarity = np.mean(Similarities)
        AverageSimilarities.append(AverageSimilarity)
    Recommendationdf = pd.DataFrame({'Legal Text': RecommendedLaws, 'Simila
    Recommendationdf = Recommendationdf.sort_values(by='Similarity1', ascer
    TopRecommendations = Recommendationdf.head(20)

    return TopRecommendations, AverageSimilarities
```

Figure 14: Implementing Recommendation Engine with BERT Natural Language Model

In [20]:

```
# Implementing recommendation engine
def Recommend_Laws(Text, Data, Embeddings):
    Similarities = []
    RecommendedLaws = []
    AverageSimilarities = []

    # Tokenizing and preprocess the input text
    Text = Text.lower()
    inputs1 = Tokenizer(
        [Text],
        return_tensors='pt',
        padding='max_length',
        max_length=MaxLength,
        truncation=True
    )
    with torch.no_grad():
        Embeddings1 = AlbertModel(**inputs1).last_hidden_state

    for i in range(len(Embeddings)):
        LegalText = Data['case_name'].iloc[i].lower()
        inputs2 = Tokenizer(
            [LegalText],
            return_tensors='pt',
            padding='max_length',
            max_length=MaxLength,
            truncation=True
        )
        with torch.no_grad():
            Embeddings2 = Embeddings[i]  # Use the precomputed DeBERTa embe

        # Calculating cosine similarity between embeddings
        Similarity = torch.nn.functional.cosine_similarity(Embeddings1, Emb

        Similarities.append(Similarity)
        RecommendedLaws.append(Data['case_name'].iloc[i])
        # Calculating average similarity for each interaction
        AverageSimilarity = np.mean(Similarities)
        AverageSimilarities.append(AverageSimilarity)
    Recommendationdf = pd.DataFrame({'Legal Text': RecommendedLaws, 'Simila
    Recommendationdf = Recommendationdf.sort_values(by='Similarity2', ascer
    TopRecommendations = Recommendationdf.head(20)

    return TopRecommendations, AverageSimilarities
```

Figure 15: Implementing Recommendation Engine with ALBERT Natural Language Model

In [26]:
```python
# Implementing recommendation engine
def RecommendLaw(Text, Data, Embeddings):
    Similarities = []
    RecommendedLaws = []
    AverageSimilarities = []

    # Tokenizing and preprocessing the input text
    Text = Text.lower()
    inputs1 = Tokenizer(
        [Text],
        return_tensors='pt',
        padding='max_length',
        max_length=MaxLength,
        truncation=True
    )
    with torch.no_grad():
        Embeddings1 = DebertaModel(**inputs1).last_hidden_state

    for i in range(len(Embeddings)):
        LegalText = Data['case_name'].iloc[i].lower()
        inputs2 = Tokenizer(
            [LegalText],
            return_tensors='pt',
            padding='max_length',
            max_length=MaxLength,
            truncation=True
        )
        with torch.no_grad():
            Embeddings2 = Embeddings[i]  # Utilizing the precomputed DeBERT

        # Calculating cosine similarity between embeddings
        Similarity = torch.nn.functional.cosine_similarity(Embeddings1, Emb

        Similarities.append(Similarity)
        RecommendedLaws.append(Data['case_name'].iloc[i])
        # Calculating average similarity for each interaction
        AverageSimilarity = np.mean(Similarities)
        AverageSimilarities.append(AverageSimilarity)
    Recommendationdf = pd.DataFrame({'Legal Text': RecommendedLaws, 'Simila
    Recommendationdf = Recommendationdf.sort_values(by='Similarity3', ascer
    TopRecommendations = Recommendationdf.head(20)

    return TopRecommendations, AverageSimilarities
```

Figure 16: Implementing Recommendation Engine with DeBERTa Natural Language Model

References

Rolon-Mérette, D., Ross, M., Rolon-Mérette, T. and Church, K. (2016). Introduction to anaconda and python: Installation and setup, *Quant. Methods Psychol* **16**(5): S3–S11.

Rothman, D. (2021). *Transformers for Natural Language Processing: Build innovative deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, RoBERTa, and more*, Packt Publishing Ltd.