Parker Christenson Assignment 2.1: Text Classification using Word Embeddings

Instructions

`Prior to beginning work on this assignment, review the week's lab session.`

Dataset

The dataset comprises newsgroup documents categorized into 20 different topics. Each document is labeled with its corresponding newsgroup category.

Download the dataset from the following link:
https://www.kaggle.com/datasets/crawford/20-newsgroups?resource=download

- Load the "20 Newsgroups" dataset into a pandas DataFrame.
- Preprocess the text data: remove stopwords, punctuation, convert to lowercase, etc.
- Split the data into training and testing sets.
- Utilize pre-trained word embeddings (e.g., Word2Vec, GloVe) and/or train your own embeddings using the training data.
- Build a text classification model using a classifier of your choice (e.g., Logistic Regression, Support Vector Machine, Neural Network).
- Train the model using the transformed training data (with embeddings).
- Predict the categories for the testing data.
- Evaluate the model's performance
- Evaluate the model's performance using the following classification metrics:
  - Accuracy
  - Precision (weighted)
  - Recall (weighted)
  - F1 Score (weighted)
  - Confusion Matrix
  - Area Under the Receiver Operating Characteristic curve (AUC-ROC)

In [100…

```
# text pre processing imports
import re
import string
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.stem import PorterStemmer
from nltk.corpus import wordnet
from nltk import ne_chunk, pos_tag
from nltk.tree import Tree
from nltk import pos_tag
```

```python
from tqdm import tqdm
import os


# standard imports
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# sklearn imports
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import roc_auc_score

# tensorflow imports
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.utils import to_categorical
```

In [49]:
```python
print("Tensorflow version: ", tf.__version__)
print("The GPU is", "available" if tf.config.experimental.list_physical_d
print("Num GPUs Available: ", len(tf.config.experimental.list_physical_de
```

```
Tensorflow version:  2.17.0
The GPU is available
Num GPUs Available:  1
```

In [50]:
```python
# Loading the data set from sklearn
from sklearn.datasets import fetch_20newsgroups
newsgroups = fetch_20newsgroups(subset='all')

print("There are total of", len(newsgroups.target_names), "categories")
```

```
There are total of 20 categories
```

In [51]:
```python
# displaying some of the data in the dataset
print(newsgroups.data[0])
```

```
From: Mamatha Devineni Ratnam <mr47+@andrew.cmu.edu>
Subject: Pens fans reactions
Organization: Post Office, Carnegie Mellon, Pittsburgh, PA
Lines: 12
NNTP-Posting-Host: po4.andrew.cmu.edu



I am sure some bashers of Pens fans are pretty confused about the lack
of any kind of posts about the recent Pens massacre of the Devils. Actuall
y,
I am  bit puzzled too and a bit relieved. However, I am going to put an en
d
to non-PIttsburghers' relief with a bit of praise for the Pens. Man, they
are killing those Devils worse than I thought. Jagr just showed you why
he is much better than his regular season stats. He is also a lot
fo fun to watch in the playoffs. Bowman should let JAgr have a lot of
fun in the next couple of games since the Pens are going to beat the pulp
out of Jersey anyway. I was very disappointed not to see the Islanders los
e the final
regular season game.          PENS RULE!!!
```

In [52]:
```python
# printing the target names
print(newsgroups.target_names)
```

```
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.
pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x', 'misc.forsale',
'rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 'rec.sport.hockey',
'sci.crypt', 'sci.electronics', 'sci.med', 'sci.space', 'soc.religion.chri
stian', 'talk.politics.guns', 'talk.politics.mideast', 'talk.politics.mis
c', 'talk.religion.misc']
```

In [53]:
```python
# getting a count of the target names and their frequency
unique, counts = np.unique(newsgroups.target, return_counts=True)
print(np.asarray((unique, counts)).T)
```

```
[[  0 799]
 [  1 973]
 [  2 985]
 [  3 982]
 [  4 963]
 [  5 988]
 [  6 975]
 [  7 990]
 [  8 996]
 [  9 994]
 [ 10 999]
 [ 11 991]
 [ 12 984]
 [ 13 990]
 [ 14 987]
 [ 15 997]
 [ 16 910]
 [ 17 940]
 [ 18 775]
 [ 19 628]]
```

In [54]:
```python
# making the data into a dataframe
data = pd.DataFrame({'text': newsgroups.data, 'target': newsgroups.target
```

```
print(data.head())
```

```
                                                text  target
0  From: Mamatha Devineni Ratnam <mr47+@andrew.cm...      10
1  From: mblawson@midway.ecn.uoknor.edu (Matthew ...       3
2  From: hilmi-er@dsv.su.se (Hilmi Eren)\nSubject...      17
3  From: guyd@austin.ibm.com (Guy Dawson)\nSubjec...       3
4  From: Alexander Samuel McDiarmid <am2o+@andrew...       4
```

In [55]:
```
# printing the first text in the data
print(data['text'][5])
```

```
From: tell@cs.unc.edu (Stephen Tell)
Subject: Re: subliminal message flashing on TV
Organization: The University of North Carolina at Chapel Hill
Lines: 25
NNTP-Posting-Host: rukbat.cs.unc.edu

In article <7480237@hpfcso.FC.HP.COM> myers@hpfcso.FC.HP.COM (Bob Myers) w
rites:
>> Hi.  I was doing research on subliminal suggestion for a psychology
>> paper, and I read that one researcher flashed hidden messages on the
>> TV screen at 1/200ths of a second.  Is that possible?

> Might
>even be a vector ("strokewriter") display, in which case the lower limit
>on image time is anyone's guess (and is probably phosphor-persistence lim
ited).

Back in high school I worked as a lab assistant for a bunch of experimenta
l
psychologists at Bell Labs.  When they were doing visual perception and
memory experiments, they used vector-type displays, with 1-millisecond
refresh rates common.

So your case of 1/200th sec is quite practical, and the experimenters were
probably sure that it was 5 milliseconds, not 4 or 6 either.

>Bob Myers  KC0EW >myers@fc.hp.com

Steve
--
Steve Tell       tell@cs.unc.edu H: 919 968 1792  | #5L Estes Park apts
UNC Chapel Hill Computer Science W: 919 962 1845  | Carrboro NC 27510
Engineering is a _lot_ like art:  Some circuits are like lyric poems, some
are like army manuals, and some are like The Hitchhiker's Guide to the Gal
axy..
```

In [70]:
```
# downloading the punk_tab
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('punkt_tab')
nltk.download('averaged_perceptron_tagger_eng')
nltk.download('maxent_ne_chunker_tab')
nltk.download('words')
```

```
[nltk_data] Downloading package punkt to /home/parker/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to /home/parker/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package punkt_tab to /home/parker/nltk_data...
[nltk_data]   Package punkt_tab is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger_eng to
[nltk_data]     /home/parker/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger_eng is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package maxent_ne_chunker_tab to
[nltk_data]     /home/parker/nltk_data...
[nltk_data]   Package maxent_ne_chunker_tab is already up-to-date!
[nltk_data] Downloading package words to /home/parker/nltk_data...
[nltk_data]   Unzipping corpora/words.zip.
```

Out[70]: True

In [71]:
```python
def Preprocess(text_data):
    """"
    Function to preprocess text file, generate preprocessed email domain,
    """

    preprocessed_email = []
    preprocessed_subject = []
    preprocessed_text = []

    for sentence in tqdm(text_data):

        # Preprocessing email
        domain = re.findall("@[\w.]+", sentence)
        email = ""
        for items in domain:
            items = items.replace("@", "")
            items = items.split(".")
            for i in set(items):
                if((len(i) > 2) and i != "com" and i != "COM"):
                    email += i + " "
        preprocessed_email.append(email.strip())

        # Preprocessing subject
        subject = ""  # Initialize subject as an empty string
        text_split = sentence.split("\n")
        for item in text_split:
            if item.startswith("Subject:"):
                for word in item.split():
                    if not word.endswith(":"):
                        subject += word + " "
                subject = re.sub("[^0-9a-zA-Z\s]", " ", subject)
                subject = " ".join(subject.split()).strip()
        preprocessed_subject.append(subject.lower())

        # Preprocessing text
        text = re.sub(r"(.*)Subject:(.*?)(.*)\n", " ", sentence)   # Remo
        text = re.sub(r"(.*)From:(.*?)(.*)\n", " ", text)          # Remo
        text = re.sub(r"(.*)Write to:(.*?)(.*)\n", " ", text)      # Remo
        text = re.sub(r"(.*):(.*?)", " ", text)                    # Remo

        # Decontraction
        text = re.sub(r"won't", "will not", text)
```

```
        text = re.sub(r"can\'t", "can not", text)
        text = re.sub(r"n\'t", " not", text)
        text = re.sub(r"\'re", " are", text)
        text = re.sub(r"\'s", " is", text)
        text = re.sub(r"\'d", " would", text)
        text = re.sub(r"\'ll", " will", text)
        text = re.sub(r"\'t", " not", text)
        text = re.sub(r"\'ve", " have", text)
        text = re.sub(r"\'m", " am", text)

        text = re.sub(r"[\w\-\.]+@[\w\.-]+\b", " ", text)          # Remo
        text = re.sub(r"[\n\t]", " ", text)                         # Remo
        text = re.sub(r"<.*>", " ", text)                           # Remo
        text = re.sub(r"\(.*\)", " ", text)                         # Remo
        text = re.sub(r"\[.*\]", " ", text)                         # Remo
        text = re.sub(r"\{.*\}", " ", text)                         # Remo
        text = re.sub("[0-9]+", " ", text)                          # Remo
        text = re.sub("[^A-Za-z\s]", " ", text)                     # Remo

        # Named Entity Recognition and Replacement
        chunks = list(ne_chunk(pos_tag(word_tokenize(text))))

        for i in chunks:
            if isinstance(i, Tree):
                if i.label() == "GPE":
                    j = i.leaves()
                    if len(j) > 1:   # If a city name has two or more wor
                        gpe = "_".join([term for term, pos in j])
                        text = re.sub(rf"{j[1][0]}", gpe, text)
                        text = re.sub(rf"{j[0][0]}", " ", text)
                if i.label() == "PERSON":
                    for term, pos in i.leaves():
                        text = re.sub(re.escape(term), "", text)

        # Clean up underscores
        text = re.sub(r"\b_([a-zA-z]+)_\b", r"\1", text)  # Replace _word
        text = re.sub(r"\b_([a-zA-z]+)\b", r"\1", text)   # Replace _word
        text = re.sub(r"\b([a-zA-z]+)_\b", r"\1", text)   # Replace word_
        text = re.sub(r"\b[a-zA-Z]{1}_([a-zA-Z]+)", r"\1", text)
        text = re.sub(r"\b[a-zA-Z]{2}_([a-zA-Z]+)", r"\1", text)

        text = text.lower()
        text_split = text.split()
        text = " ".join(word for word in text_split if 2 < len(word) < 15
        preprocessed_text.append(text.strip())

    return preprocessed_email, preprocessed_subject, preprocessed_text
```

In [72]:
```
# applying the function to the text column
emails, subjects, texts = Preprocess(data['text'])
```

```
100%|██████████| 18846/18846 [1:08:49<00:00,  4.56it/s]
```

In [77]:
```
# making the final dataframe with the subjects and texts columns
final_data = pd.DataFrame({'email': emails, 'subject': subjects, 'text':
final_data.head()
```

Out[77]:

	email	subject	text	target
0	edu cmu andrew	pens fans reactions	office carnegie mellon pittsburgh andrew cmu e...	10
1	edu midway uoknor ecn edu essex uoknor ecn	which high performance vlb video card	seek for video card midway ecn uoknor edu engi...	3
2	dsv dsv	armenia says it could shoot down turkish plane...	viktoria dsv dept computer and would stretch f...	17
3	austin ibm austin ibm pal500 julian uwo heart ...	ide vs scsi dma and detach	ibm can anyone explain fairly simple terms why...	3
4	edu cmu andrew	driver	sophomore mechanical engineering carnegie mell...	4

Building the text classification model using tensorflow

In [79]:
```
# using word2vec to convert the text into vectors after tokenizing
from gensim.models import Word2Vec
from gensim.models import KeyedVectors
```

In [81]:
```
# tokenize the text data
sentences = [word_tokenize(text) for text in data['text']]

# train the Word2Vec model
word2vec_model = Word2Vec(sentences, vector_size=100, window=5, min_count

# printing the number of words in the model
print(f'Trained Word2Vec model with {len(word2vec_model.wv)} words in the
```

```
Trained Word2Vec model with 276455 words in the vocabulary.
```

In [82]:
```
# function to conver the text
def text_to_embedding(text, model, embedding_dim=100):
    words = word_tokenize(text)
    word_vectors = [model.wv[word] for word in words if word in model.wv]

    if not word_vectors:  # if no word retun none
        return np.zeros(embedding_dim)

    return np.mean(word_vectors, axis=0)

# apply the function to the text col
data['text_embedding'] = data['text'].apply(lambda x: text_to_embedding(x
```

In [85]:
```
X = np.array(data['text_embedding'].tolist())
y = final_data['target']

# spliting the data for the testing and training split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
```

In [93]:
```
# setting the number of topics
num_topics = len(set(y_train))

# building the model
model = Sequential()
model.add(Dense(512, activation='relu', input_shape=(X_train.shape[1],)))
model.add(Dropout(0.5))
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_topics, activation='softmax'))

# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', m
```

```
/home/parker/anaconda3/envs/tf_ml/lib/python3.10/site-packages/keras/src/l
ayers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_d
im` argument to a layer. When using Sequential models, prefer using an `In
put(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In [94]:
```
# Train the model
model.fit(X_train, y_train, epochs=100, batch_size=64, validation_split=0
```

```
Epoch 1/100
189/189 - 6s - 29ms/step - accuracy: 0.0993 - loss: 2.8669 - val_accuracy:
0.1983 - val_loss: 2.5268
Epoch 2/100
189/189 - 0s - 1ms/step - accuracy: 0.1912 - loss: 2.4187 - val_accuracy:
0.2974 - val_loss: 2.0984
Epoch 3/100
189/189 - 0s - 1ms/step - accuracy: 0.2402 - loss: 2.2201 - val_accuracy:
0.3253 - val_loss: 2.0165
Epoch 4/100
189/189 - 0s - 1ms/step - accuracy: 0.2734 - loss: 2.1200 - val_accuracy:
0.3501 - val_loss: 1.9224
Epoch 5/100
189/189 - 0s - 1ms/step - accuracy: 0.2980 - loss: 2.0479 - val_accuracy:
0.3528 - val_loss: 1.8736
Epoch 6/100
189/189 - 0s - 1ms/step - accuracy: 0.3185 - loss: 1.9996 - val_accuracy:
0.3840 - val_loss: 1.8023
Epoch 7/100
189/189 - 0s - 1ms/step - accuracy: 0.3345 - loss: 1.9431 - val_accuracy:
0.4208 - val_loss: 1.7514
Epoch 8/100
189/189 - 0s - 1ms/step - accuracy: 0.3514 - loss: 1.9133 - val_accuracy:
0.4171 - val_loss: 1.7295
Epoch 9/100
189/189 - 0s - 1ms/step - accuracy: 0.3547 - loss: 1.8831 - val_accuracy:
0.4188 - val_loss: 1.7254
Epoch 10/100
189/189 - 0s - 1ms/step - accuracy: 0.3621 - loss: 1.8670 - val_accuracy:
0.4377 - val_loss: 1.6781
Epoch 11/100
189/189 - 0s - 1ms/step - accuracy: 0.3710 - loss: 1.8374 - val_accuracy:
0.4380 - val_loss: 1.6658
Epoch 12/100
189/189 - 0s - 1ms/step - accuracy: 0.3749 - loss: 1.8288 - val_accuracy:
0.4254 - val_loss: 1.6717
Epoch 13/100
189/189 - 0s - 1ms/step - accuracy: 0.3750 - loss: 1.8152 - val_accuracy:
0.4486 - val_loss: 1.6544
Epoch 14/100
189/189 - 0s - 1ms/step - accuracy: 0.3883 - loss: 1.7900 - val_accuracy:
0.4446 - val_loss: 1.6438
Epoch 15/100
189/189 - 0s - 1ms/step - accuracy: 0.3905 - loss: 1.7903 - val_accuracy:
0.4463 - val_loss: 1.6387
Epoch 16/100
189/189 - 0s - 1ms/step - accuracy: 0.3971 - loss: 1.7686 - val_accuracy:
0.4277 - val_loss: 1.6792
Epoch 17/100
189/189 - 0s - 1ms/step - accuracy: 0.4001 - loss: 1.7607 - val_accuracy:
0.4632 - val_loss: 1.6067
Epoch 18/100
189/189 - 0s - 1ms/step - accuracy: 0.4046 - loss: 1.7328 - val_accuracy:
0.4622 - val_loss: 1.5923
Epoch 19/100
189/189 - 0s - 1ms/step - accuracy: 0.4020 - loss: 1.7475 - val_accuracy:
0.4542 - val_loss: 1.6007
Epoch 20/100
189/189 - 0s - 1ms/step - accuracy: 0.4108 - loss: 1.7368 - val_accuracy:
0.4493 - val_loss: 1.5830
```

```
Epoch 21/100
189/189 - 0s - 1ms/step - accuracy: 0.4038 - loss: 1.7266 - val_accuracy:
0.4649 - val_loss: 1.5814
Epoch 22/100
189/189 - 0s - 1ms/step - accuracy: 0.4154 - loss: 1.7205 - val_accuracy:
0.4721 - val_loss: 1.5718
Epoch 23/100
189/189 - 0s - 1ms/step - accuracy: 0.4164 - loss: 1.7143 - val_accuracy:
0.4569 - val_loss: 1.5873
Epoch 24/100
189/189 - 0s - 1ms/step - accuracy: 0.4175 - loss: 1.6962 - val_accuracy:
0.4755 - val_loss: 1.5709
Epoch 25/100
189/189 - 0s - 1ms/step - accuracy: 0.4241 - loss: 1.6950 - val_accuracy:
0.4589 - val_loss: 1.5723
Epoch 26/100
189/189 - 0s - 1ms/step - accuracy: 0.4227 - loss: 1.6906 - val_accuracy:
0.4801 - val_loss: 1.5633
Epoch 27/100
189/189 - 0s - 1ms/step - accuracy: 0.4235 - loss: 1.6885 - val_accuracy:
0.4619 - val_loss: 1.5684
Epoch 28/100
189/189 - 0s - 1ms/step - accuracy: 0.4266 - loss: 1.6908 - val_accuracy:
0.4758 - val_loss: 1.5635
Epoch 29/100
189/189 - 0s - 1ms/step - accuracy: 0.4292 - loss: 1.6584 - val_accuracy:
0.4765 - val_loss: 1.5470
Epoch 30/100
189/189 - 0s - 1ms/step - accuracy: 0.4336 - loss: 1.6726 - val_accuracy:
0.4735 - val_loss: 1.5413
Epoch 31/100
189/189 - 0s - 1ms/step - accuracy: 0.4343 - loss: 1.6584 - val_accuracy:
0.4755 - val_loss: 1.5339
Epoch 32/100
189/189 - 0s - 1ms/step - accuracy: 0.4318 - loss: 1.6580 - val_accuracy:
0.4814 - val_loss: 1.5253
Epoch 33/100
189/189 - 0s - 1ms/step - accuracy: 0.4425 - loss: 1.6469 - val_accuracy:
0.4907 - val_loss: 1.5390
Epoch 34/100
189/189 - 0s - 1ms/step - accuracy: 0.4368 - loss: 1.6565 - val_accuracy:
0.4834 - val_loss: 1.5365
Epoch 35/100
189/189 - 0s - 1ms/step - accuracy: 0.4360 - loss: 1.6557 - val_accuracy:
0.4891 - val_loss: 1.5125
Epoch 36/100
189/189 - 0s - 1ms/step - accuracy: 0.4386 - loss: 1.6449 - val_accuracy:
0.4761 - val_loss: 1.5291
Epoch 37/100
189/189 - 0s - 1ms/step - accuracy: 0.4494 - loss: 1.6235 - val_accuracy:
0.4924 - val_loss: 1.5133
Epoch 38/100
189/189 - 0s - 1ms/step - accuracy: 0.4463 - loss: 1.6294 - val_accuracy:
0.4980 - val_loss: 1.5048
Epoch 39/100
189/189 - 0s - 1ms/step - accuracy: 0.4361 - loss: 1.6412 - val_accuracy:
0.4828 - val_loss: 1.5061
Epoch 40/100
189/189 - 0s - 1ms/step - accuracy: 0.4487 - loss: 1.6275 - val_accuracy:
0.4920 - val_loss: 1.4947
```

```
Epoch 41/100
189/189 - 0s - 1ms/step - accuracy: 0.4427 - loss: 1.6314 - val_accuracy: 
0.4944 - val_loss: 1.4953
Epoch 42/100
189/189 - 0s - 1ms/step - accuracy: 0.4502 - loss: 1.6142 - val_accuracy: 
0.4927 - val_loss: 1.5105
Epoch 43/100
189/189 - 0s - 1ms/step - accuracy: 0.4542 - loss: 1.6136 - val_accuracy: 
0.4924 - val_loss: 1.4888
Epoch 44/100
189/189 - 0s - 1ms/step - accuracy: 0.4471 - loss: 1.6194 - val_accuracy: 
0.4841 - val_loss: 1.5263
Epoch 45/100
189/189 - 0s - 1ms/step - accuracy: 0.4566 - loss: 1.6068 - val_accuracy: 
0.4728 - val_loss: 1.5534
Epoch 46/100
189/189 - 0s - 1ms/step - accuracy: 0.4512 - loss: 1.6081 - val_accuracy: 
0.5070 - val_loss: 1.4848
Epoch 47/100
189/189 - 0s - 1ms/step - accuracy: 0.4522 - loss: 1.6023 - val_accuracy: 
0.4957 - val_loss: 1.4971
Epoch 48/100
189/189 - 0s - 1ms/step - accuracy: 0.4593 - loss: 1.6012 - val_accuracy: 
0.4993 - val_loss: 1.4950
Epoch 49/100
189/189 - 0s - 1ms/step - accuracy: 0.4531 - loss: 1.6054 - val_accuracy: 
0.5139 - val_loss: 1.4755
Epoch 50/100
189/189 - 0s - 1ms/step - accuracy: 0.4554 - loss: 1.5912 - val_accuracy: 
0.4877 - val_loss: 1.4966
Epoch 51/100
189/189 - 0s - 1ms/step - accuracy: 0.4585 - loss: 1.5880 - val_accuracy: 
0.5050 - val_loss: 1.4760
Epoch 52/100
189/189 - 0s - 1ms/step - accuracy: 0.4547 - loss: 1.5855 - val_accuracy: 
0.5070 - val_loss: 1.4690
Epoch 53/100
189/189 - 0s - 1ms/step - accuracy: 0.4612 - loss: 1.5841 - val_accuracy: 
0.5043 - val_loss: 1.4877
Epoch 54/100
189/189 - 0s - 1ms/step - accuracy: 0.4615 - loss: 1.5845 - val_accuracy: 
0.5003 - val_loss: 1.4868
Epoch 55/100
189/189 - 0s - 1ms/step - accuracy: 0.4629 - loss: 1.5819 - val_accuracy: 
0.5007 - val_loss: 1.4899
Epoch 56/100
189/189 - 0s - 1ms/step - accuracy: 0.4630 - loss: 1.5769 - val_accuracy: 
0.5093 - val_loss: 1.4840
Epoch 57/100
189/189 - 0s - 1ms/step - accuracy: 0.4762 - loss: 1.5583 - val_accuracy: 
0.5123 - val_loss: 1.4701
Epoch 58/100
189/189 - 0s - 1ms/step - accuracy: 0.4677 - loss: 1.5684 - val_accuracy: 
0.5116 - val_loss: 1.4681
Epoch 59/100
189/189 - 0s - 1ms/step - accuracy: 0.4626 - loss: 1.5699 - val_accuracy: 
0.5030 - val_loss: 1.4767
Epoch 60/100
189/189 - 0s - 1ms/step - accuracy: 0.4680 - loss: 1.5755 - val_accuracy: 
0.5093 - val_loss: 1.4680
```

```
Epoch 61/100
189/189 - 0s - 1ms/step - accuracy: 0.4655 - loss: 1.5663 - val_accuracy:
0.5090 - val_loss: 1.4641
Epoch 62/100
189/189 - 0s - 1ms/step - accuracy: 0.4711 - loss: 1.5545 - val_accuracy:
0.5126 - val_loss: 1.4476
Epoch 63/100
189/189 - 0s - 1ms/step - accuracy: 0.4723 - loss: 1.5444 - val_accuracy:
0.5179 - val_loss: 1.4575
Epoch 64/100
189/189 - 0s - 1ms/step - accuracy: 0.4700 - loss: 1.5622 - val_accuracy:
0.5149 - val_loss: 1.4600
Epoch 65/100
189/189 - 0s - 1ms/step - accuracy: 0.4759 - loss: 1.5468 - val_accuracy:
0.5113 - val_loss: 1.4473
Epoch 66/100
189/189 - 0s - 1ms/step - accuracy: 0.4781 - loss: 1.5514 - val_accuracy:
0.5209 - val_loss: 1.4502
Epoch 67/100
189/189 - 0s - 1ms/step - accuracy: 0.4760 - loss: 1.5448 - val_accuracy:
0.5229 - val_loss: 1.4482
Epoch 68/100
189/189 - 0s - 1ms/step - accuracy: 0.4781 - loss: 1.5385 - val_accuracy:
0.5172 - val_loss: 1.4588
Epoch 69/100
189/189 - 0s - 1ms/step - accuracy: 0.4763 - loss: 1.5428 - val_accuracy:
0.5179 - val_loss: 1.4404
Epoch 70/100
189/189 - 0s - 1ms/step - accuracy: 0.4736 - loss: 1.5356 - val_accuracy:
0.5252 - val_loss: 1.4525
Epoch 71/100
189/189 - 0s - 1ms/step - accuracy: 0.4759 - loss: 1.5434 - val_accuracy:
0.5066 - val_loss: 1.4726
Epoch 72/100
189/189 - 0s - 1ms/step - accuracy: 0.4729 - loss: 1.5415 - val_accuracy:
0.5196 - val_loss: 1.4447
Epoch 73/100
189/189 - 0s - 1ms/step - accuracy: 0.4784 - loss: 1.5425 - val_accuracy:
0.5186 - val_loss: 1.4377
Epoch 74/100
189/189 - 0s - 1ms/step - accuracy: 0.4740 - loss: 1.5432 - val_accuracy:
0.5222 - val_loss: 1.4643
Epoch 75/100
189/189 - 0s - 1ms/step - accuracy: 0.4811 - loss: 1.5332 - val_accuracy:
0.5225 - val_loss: 1.4321
Epoch 76/100
189/189 - 0s - 1ms/step - accuracy: 0.4779 - loss: 1.5390 - val_accuracy:
0.5212 - val_loss: 1.4500
Epoch 77/100
189/189 - 0s - 1ms/step - accuracy: 0.4834 - loss: 1.5300 - val_accuracy:
0.5219 - val_loss: 1.4380
Epoch 78/100
189/189 - 0s - 1ms/step - accuracy: 0.4821 - loss: 1.5199 - val_accuracy:
0.5070 - val_loss: 1.4642
Epoch 79/100
189/189 - 0s - 1ms/step - accuracy: 0.4795 - loss: 1.5294 - val_accuracy:
0.5159 - val_loss: 1.4419
Epoch 80/100
189/189 - 0s - 1ms/step - accuracy: 0.4734 - loss: 1.5224 - val_accuracy:
0.5169 - val_loss: 1.4497
```

```
Epoch 81/100
189/189 - 0s - 1ms/step - accuracy: 0.4881 - loss: 1.5033 - val_accuracy:
0.5202 - val_loss: 1.4447
Epoch 82/100
189/189 - 0s - 1ms/step - accuracy: 0.4808 - loss: 1.5129 - val_accuracy:
0.5212 - val_loss: 1.4329
Epoch 83/100
189/189 - 0s - 1ms/step - accuracy: 0.4837 - loss: 1.5262 - val_accuracy:
0.5179 - val_loss: 1.4424
Epoch 84/100
189/189 - 0s - 1ms/step - accuracy: 0.4936 - loss: 1.4941 - val_accuracy:
0.5232 - val_loss: 1.4242
Epoch 85/100
189/189 - 0s - 1ms/step - accuracy: 0.4828 - loss: 1.5225 - val_accuracy:
0.5202 - val_loss: 1.4379
Epoch 86/100
189/189 - 0s - 1ms/step - accuracy: 0.4923 - loss: 1.4993 - val_accuracy:
0.5196 - val_loss: 1.4451
Epoch 87/100
189/189 - 0s - 1ms/step - accuracy: 0.4854 - loss: 1.5171 - val_accuracy:
0.5169 - val_loss: 1.4480
Epoch 88/100
189/189 - 0s - 1ms/step - accuracy: 0.4897 - loss: 1.5069 - val_accuracy:
0.5285 - val_loss: 1.4266
Epoch 89/100
189/189 - 0s - 1ms/step - accuracy: 0.4871 - loss: 1.5023 - val_accuracy:
0.5136 - val_loss: 1.4453
Epoch 90/100
189/189 - 0s - 1ms/step - accuracy: 0.4903 - loss: 1.4866 - val_accuracy:
0.5206 - val_loss: 1.4383
Epoch 91/100
189/189 - 0s - 1ms/step - accuracy: 0.4925 - loss: 1.5015 - val_accuracy:
0.5235 - val_loss: 1.4370
Epoch 92/100
189/189 - 0s - 1ms/step - accuracy: 0.4882 - loss: 1.5082 - val_accuracy:
0.5186 - val_loss: 1.4360
Epoch 93/100
189/189 - 0s - 1ms/step - accuracy: 0.4964 - loss: 1.4898 - val_accuracy:
0.5249 - val_loss: 1.4248
Epoch 94/100
189/189 - 0s - 1ms/step - accuracy: 0.4911 - loss: 1.4975 - val_accuracy:
0.5279 - val_loss: 1.4427
Epoch 95/100
189/189 - 0s - 1ms/step - accuracy: 0.4855 - loss: 1.4981 - val_accuracy:
0.5288 - val_loss: 1.4228
Epoch 96/100
189/189 - 0s - 1ms/step - accuracy: 0.4973 - loss: 1.4901 - val_accuracy:
0.5275 - val_loss: 1.4211
Epoch 97/100
189/189 - 0s - 1ms/step - accuracy: 0.4890 - loss: 1.4920 - val_accuracy:
0.5255 - val_loss: 1.4252
Epoch 98/100
189/189 - 0s - 1ms/step - accuracy: 0.4968 - loss: 1.4908 - val_accuracy:
0.5176 - val_loss: 1.4685
Epoch 99/100
189/189 - 0s - 1ms/step - accuracy: 0.4957 - loss: 1.5008 - val_accuracy:
0.5212 - val_loss: 1.4182
Epoch 100/100
189/189 - 0s - 1ms/step - accuracy: 0.4938 - loss: 1.4967 - val_accuracy:
0.5242 - val_loss: 1.4352
```

Out[94]: <keras.src.callbacks.history.History at 0x71f6fbfe1c90>

In [95]:
```
# predict
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
```

```
118/118 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
```

In [96]:
```
# evaluate the model
accuracy = accuracy_score(y_test, y_pred_classes)
precision = precision_score(y_test, y_pred_classes, average='weighted')
recall = recall_score(y_test, y_pred_classes, average='weighted')
f1 = f1_score(y_test, y_pred_classes, average='weighted')
conf_matrix = confusion_matrix(y_test, y_pred_classes)
```

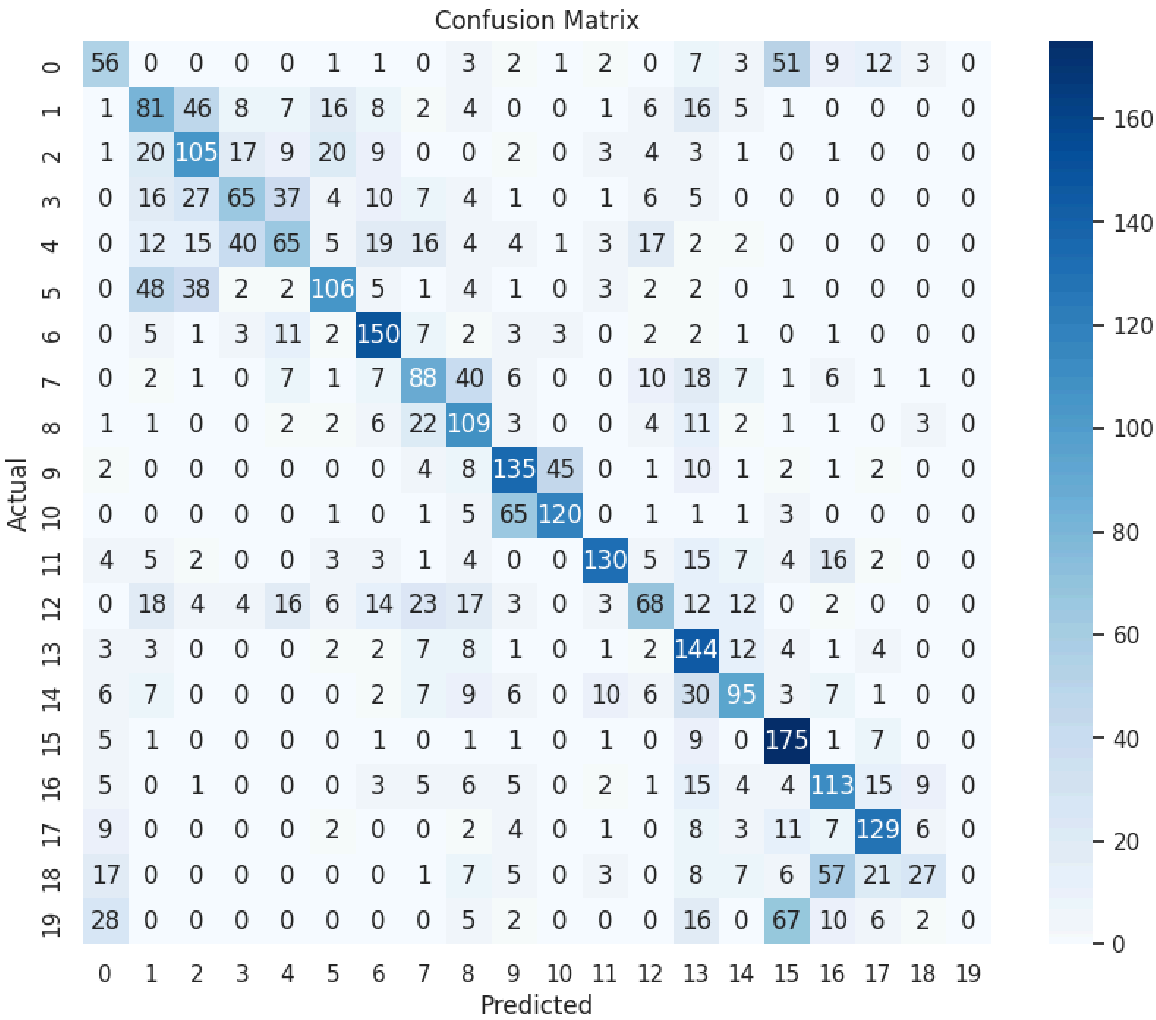
```
/home/parker/anaconda3/envs/tf_ml/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1531: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

In [101…
```
# Calculate AUC-ROC
y_test_categorical = to_categorical(y_test, num_classes=num_topics)
auc_roc = roc_auc_score(y_test_categorical, y_pred, average='weighted', m
```

In [102…
```
print(f'Accuracy: {accuracy}')
print(f'Precision (weighted): {precision}')
print(f'Recall (weighted): {recall}')
print(f'F1 Score (weighted): {f1}')
print(f'Area Under ROC Curve (AUC-ROC): {auc_roc}')
```

```
Accuracy: 0.520159151193634
Precision (weighted): 0.5091006819709496
Recall (weighted): 0.520159151193634
F1 Score (weighted): 0.5013975361823386
Area Under ROC Curve (AUC-ROC): 0.9420493090049796
```

In [103…
```
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```


Confusion Matrix
Actual
Predicted
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0: 56 0 0 0 0 1 1 0 3 2 1 2 0 7 3 51 9 12 3 0
1: 1 81 46 8 7 16 8 2 4 0 0 1 6 16 5 1 0 0 0 0
2: 1 20 105 17 9 20 9 0 0 2 0 3 4 3 1 0 1 0 0 0
3: 0 16 27 65 37 4 10 7 4 1 0 1 6 5 0 0 0 0 0 0
4: 0 12 15 40 65 5 19 16 4 4 1 3 17 2 2 0 0 0 0 0
5: 0 48 38 2 2 106 5 1 4 1 0 3 2 2 0 1 0 0 0 0
6: 0 5 1 3 11 2 150 7 2 3 3 0 2 2 1 0 1 0 0 0
7: 0 2 1 0 7 1 7 88 40 6 0 0 10 18 7 1 6 1 1 0
8: 1 1 0 0 2 2 6 22 109 3 0 0 4 11 2 1 1 0 3 0
9: 2 0 0 0 0 0 0 4 8 135 45 0 1 10 1 2 1 2 0 0
10: 0 0 0 0 0 1 0 1 5 65 120 0 1 1 1 3 0 0 0 0
11: 4 5 2 0 0 3 3 1 4 0 0 130 5 15 7 4 16 2 0 0
12: 0 18 4 4 16 6 14 23 17 3 0 3 68 12 12 0 2 0 0 0
13: 3 3 0 0 0 2 2 7 8 1 0 1 2 144 12 4 1 4 0 0
14: 6 7 0 0 0 0 2 7 9 6 0 10 6 30 95 3 7 1 0 0
15: 5 1 0 0 0 0 1 0 1 1 0 1 0 9 0 175 1 7 0 0
16: 5 0 1 0 0 0 3 5 6 5 0 2 1 15 4 4 113 15 9 0
17: 9 0 0 0 0 2 0 0 2 4 0 1 0 8 3 11 7 129 6 0
18: 17 0 0 0 0 0 0 1 7 5 0 3 0 8 7 6 57 21 27 0
19: 28 0 0 0 0 0 0 0 5 2 0 0 0 16 0 67 10 6 2 0
160
140
120
100
80
60
40
20
0

In [ ]: