STAN49 - Assignment 2

William Nordansjö

March 13, 2025

Introduction

This assignment will consist of two parts covering two different aspects of textual data analysis. The first part relates to word embeddings, a skip-gram method will be employed in order to create word embeddings, these will then be compared to a pre-trained set of embeddings in training a deep neural network to preform classification. The second part of this assignment is more free form, and the task is to identify the odd one out of a collection of French plays. This will be done using an unsupervised learning approach by application of clustering and dimension reduction methods.

Task 1: AG's News Topic Classification

In this task I aim to classify the topic of 7600 news articles by training a deep learning neural network on 120 000 other articles. The documents are courtesy of Gulli (2015) and consists of news articles from the four categories: *World*, *Sports*, *Business*, and *Sci/Tech*. To help me in this endeavor I will use two different word embeddings: A locally trained model using skip-grams with *Word2Vec*, in addition the embeddings of pre-trained model called GloVe will be used as a benchmark when training and fitting the neural network. Word embeddings assign a value to a word in a vocabulary based on its similarity to neighboring words, and can as such be a very powerful tool in natural language processing by condensing a large amount of information, including context and a proxy for sentiment of a word in a single value.

The first word embeddings is thus the skip-gram embeddings. In order to obtain these, the dataset was loaded into python, the articles content was concatenated with their titles in order to form a dataframe of documents with a category and some text. This text was preprocessed in the usual fashion; predefined english stopwords was removed, the sentences was forced into lowercase, dots and commas was removed and finally each word was converted into a token. After this, each word in the vocabulary was assigned an index, this was then flipped so that each indexed value was paired with a word. The result of the preprocessing can be found in table 1.

Table 1: Example Text and Sequence after Preprocessing.

Text	**Sequence of indices**
wall st bears claw back black reuters reuters ...	[31825, 26071, 69558, 76225, 77386, 68455, 287...]
carlyle looks toward commercial aerospace reut...	[23682, 21340, 65845, 56306, 80426, 28778, 287...]

The next step in acquiring the embeddings is to generate the training data required to fit the *Word2Vec* function. The training data consist of positive skip-gram pairs which are true empirical pairs of words which creates a target and negative skip-grams which are out of sample pairs or false context. The *Word2Vec* function is then applied to the training data and thus creates the word embedding vector by trying to predict the context of a given word. Applied on the data created above, the function achieved a very high degree of accuracy as can be seen in table 2. After training we now have our locally trained, skip-gram, 100 dimensional word embeddings.

Table 2: Model Evaluation Results

Metric	**Value**
Accuracy	0.9962
Loss	0.0221

The other set of word embeddings is as mentioned previously transferred from a pre-trained GloVe model. A GloVe model is in contrast to the skip-gram model trained on global co-occurrences between word pairs, this means that it should be able to capture larger trends in the vocabulary. The pre-trained word vector was extracted from Pennington et. al (2014) and in order to align with the skip-gram model, the 100 dimension version was used. The extracted data file contained 400000 word vectors which was matched with words in the vocabulary of our news article dataset.

The model architecture chosen is based on the Bidirectional Gated Recurrent Unit, (BiGru) and consist of an embedding layer, three recurrent units and one dense layer before a final dense softmax output layer. Each BiGru and dense layer has an accompanying dropout regularization layer. This architecture will be applied two times the news data in a multiclass setting in order to predict which news category a given article is about, both times the model will be tuned over overall dropout rate but also unit size of the recurrent and dense layer. The main difference of the two applications will be which word embeddings that are used.

The result of the tuning of the skip-gram based model can be found in table 3. And the performance of the model can be seen in table 5 as well as in figure 1. Similarly, the tuning made using the GloVe embeddings is stored in table 4, overall classification performance metrics can be found in table 6 and finally the classifications themselves are visualized in figure 2.

Table 3: Optimal Hyperparameters for Skip-gram Model Found by Keras Tuner

Hyperparameter	**Optimal Value**
GRU Unit 1	128
GRU Unit 2	256
GRU Unit 3	128
Dense Units	64
Dropout Rate	0.2
Batch Size	512
Optimizer	Adam

Table 4: Optimal Hyperparameters for GloVe Model Found by Keras Tuner

Hyperparameter	**Optimal Value**
GRU Unit 1	192
GRU Unit 2	128
GRU Unit 3	128
Dense Units	128
Dropout Rate	0.2
Batch Size	512
Optimizer	Adam

Overall, both models performed well by achieving a high degree of accuracy, with the Skip-Gram model correctly classifying 84% of the sequences and the GloVe model scored even higher at 92%. Seeing however that this is a multi-class classification scenario, simply looking at accuracies only gets you so far. In the classification reports, we can get a sense of how successful the models were in classifying specific categories. For both models, *Sports* was by far the easiest category to classify while *Business* was the hardest. Even more interesting, by looking at the confusion matrices, we can see how the categories interact with each other. Here we can see that both models had trouble discerning between *Business* and *Sci/Tech*, probably because many of the finance related news articles are related to scientific advances. Given that we have designed our experiment correctly, the only difference between the models are the word embeddings. Thus we can accredit the difference in performance the GloVe models capability of capturing global trends, and while the Skip-Gram generated embeddings produced a competent model, it is restricted by its local nature.

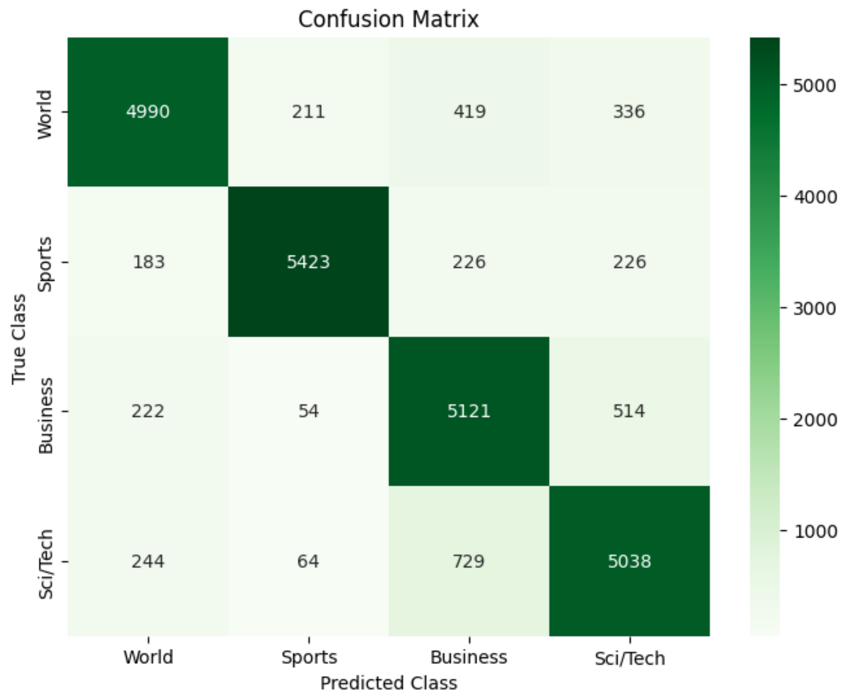

Figure 1: Confusion Matrix of Classifications Made by the Skip-gram Model

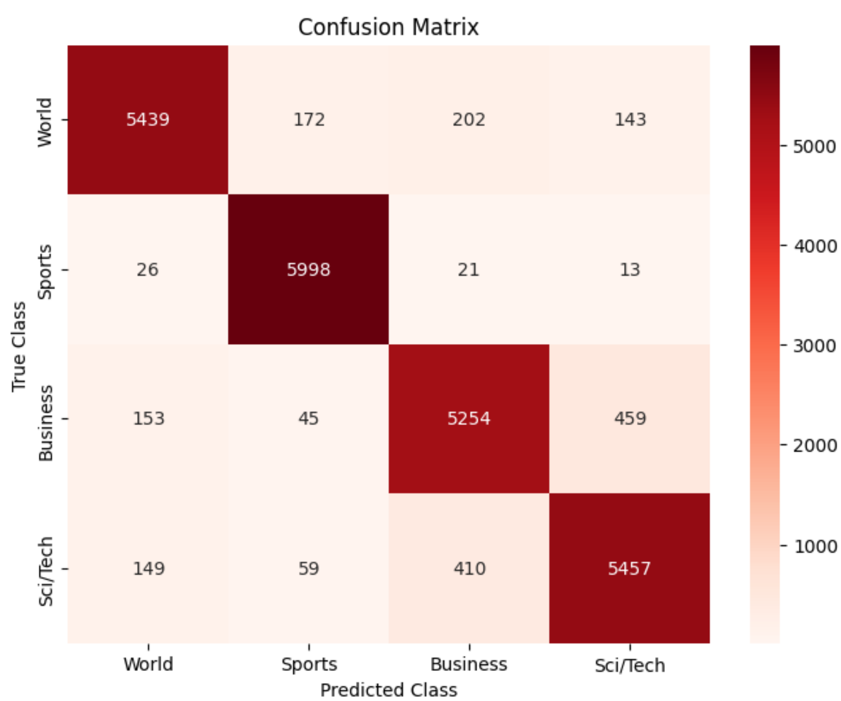

Figure 2: Confusion Matrix of Classifications Made by the GloVe Model

Table 5: Classification Report: Precision, Recall, and F1-Score

Class	Precision	Recall	F1-Score	Support
World	0.88	0.84	0.86	5956
Sports	0.94	0.90	0.92	6058
Business	0.79	0.87	0.83	5911
Sci/Tech	0.82	0.83	0.83	6075
Accuracy		0.86 (24000 samples)		
Macro Avg	0.86	0.86	0.86	24000
Weighted Avg	0.86	0.86	0.86	24000

Table 6: Classification Report: Precision, Recall, and F1-Score

Class	Precision	Recall	F1-Score	Support
World	0.94	0.91	0.93	5956
Sports	0.96	0.99	0.97	6058
Business	0.89	0.89	0.89	5911
Sci/Tech	0.90	0.90	0.90	6075
Accuracy		0.92 (24000 samples)		
Macro Avg	0.92	0.92	0.92	24000
Weighted Avg	0.92	0.92	0.92	24000

Task 2: Identify the Odd Tragedy

In this section, the task is to identify the odd one out from a collection of French plays, tragedies to be specific. This collection makes up a corpus of the speaking lines of ten plays written in French. Nine of the plays are written by one author and one play is written by a different author. In aid of our mission to root which of the plays it is, we have the statistical field of unsupervised learning, of which we will use three methods: K-means clustering, Hierarchical clustering and finally PCA or Principal Component Analysis.

First of however, we have to pre-process our text into a more usable format. This will be done by first removing dots and commas and coercing lowercase lettering, before removing *French* stopwords (relying on the nltk package to make those distinctions) and turning the words into tokens. Finally the documents were converted to tf-idf vectors, meaning that each term was weighted in relation to how rare it is in its document.

The first clustering approach applied to the text was k-means clustering. This is an iterative EM-, or Expectation Maximization method which tries to sort the data into a given amount of groups, by calculating the mean or centroid of those groups and stepwise tries to find observations that are closest to this centroid. In this case we are interested in finding two groups, one of plays written by the main

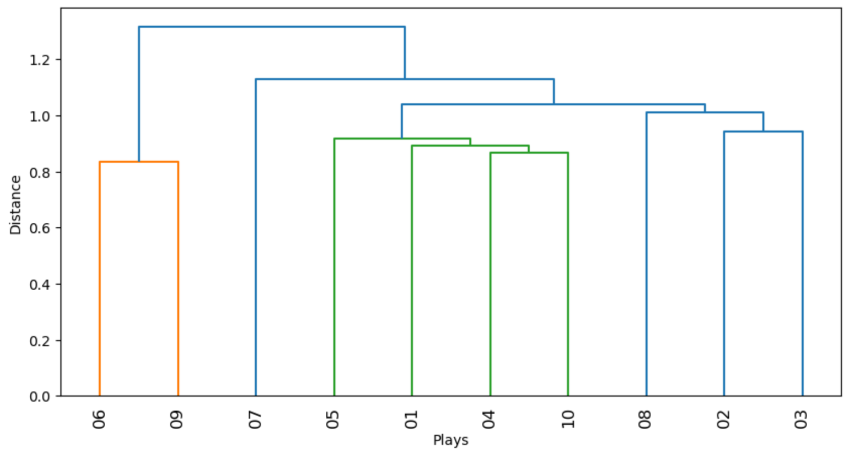

Figure 3: Hierarchical Clustering Dendrogram

author and the other consisting of the play written by the other author. The package used courtesy of sklearn utilizes the k-means++ initialization algorithm which places the initial clusters evenly among the data and then iteratively shifts the centroid until two distinct groups are formed. The result can be found in table 7, and shows that the algorithm found two groups, one with 8 plays and only containing plays 6 and 9.

Table 7: Cluster Assignments of Plays

Play	**Cluster**	**Play**	**Cluster**
Play 01	1	Play 06	0
Play 02	1	Play 07	1
Play 03	1	Play 08	1
Play 04	1	Play 09	0
Play 05	1	Play 10	1

The second clustering approach is hierarchical clustering which attempts to group the observations based upon some measure of distance. There are many of these measures, often called linkages, to choose from but in this case *Ward's* method is employed. *Ward* linkage tries to find clusters such that they minimize within cluster variance, starting from one cluster and then splitting and grouping observations until each is its own cluster. This process is visualized in the dendrogram in figure 3, and interestingly, when cutting the diagram horizontally such that two clusters are formed, these are identical to the clusters formed by the k-means algorithm. If however the dendrogram was cut into three clusters, play number 7 would make a cluster of its own.

The final method employed in this analysis is Principal Component Analysis (PCA). PCA is a widely used dimensionality reduction technique that transforms high-dimensional data into a lower-dimensional space

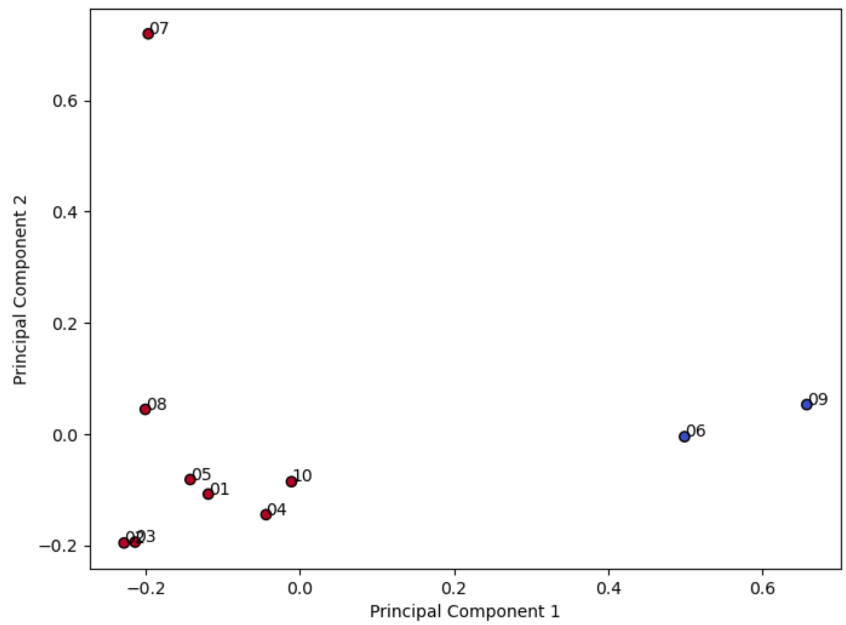

Figure 4: PCA Projection of Plays

while preserving as much variance as possible. The method works by computing the eigenvectors and eigenvalues of the datas covariance matrix, where each principal component corresponds to an eigenvector that maximizes the variance along that direction. These principal components are orthogonal to each other and ordered by their ability to explain variance in the dataset. In figure 4 the first two principal components are plotted against each other and is assigned to one of two groups. The pattern seen previously is again repeated here where plays 6 and 9 are separate when looking at two groups but when a third is considered play number 7 again stands out.

From our analysis we have found compelling evidence for the play written by the other author. First of all, when only looking at two clusters, the suspect is hidden among its stylized neighbors and its true nature is only revealed by examining a three cluster scenario. The working theory is thus that we have two authors, but also two types or styles of plays. Our previous knowledge is that all of the plays are tragedies, but apparently these two types of tragedies are distinct to our models. These styles are nigh on impossible to discern without knowledge of either French literary style or even French as a language, but perhaps there is a different tone to it. This leads us to conclude: There are two styles, the first contains two plays written by our main author, and the second contains 8 plays, 7 of which are written by the same, and one written by the other writer. As such, play number 7 is the odd tragedy.

Conclusion

In this assignment, to two distinct textual analysis tasks were completed using deep-learning and unsupervised learning techniques. The first task was classifying news topics, where a deep-learning model was trained on locally generated skip-gram embeddings and compared to a similar model using pre-trained GloVe embeddings instead. The latter outperformed the former, highlighting the benefits of capturing global word relationships. In the second task an odd French tragedy was supposed to be found. Here k-means, hierarchical clustering, and PCA consistently found that plays 6 and 9 differed from the rest, but examining a three-cluster scenario revealed play 7 to be a separate cluster. This suggests the existence of two stylistic groups, with play 7 standing apart as the odd one out.

References

Gulli, A. 2015. *AG's corpus of news articles*, Link: http://groups.di.unipi.it/˜gulli/AG_corpus_of_news_articles.html

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. *GloVe: Global Vectors for Word Representation*, Link: https://nlp.stanford.edu/projects/glove/

Appendix A: Python Code

```

import numpy as np
import pandas as pd
import os
import random
import re
import tensorflow as tf
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, LSTM, GRU, Dense,
    Dropout, Input, Bidirectional
from tensorflow.keras.callbacks import EarlyStopping
import keras_tuner as kt
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
import tqdm
from tensorflow.keras import layers
from collections import Counter
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import nltk

#Data
column_names = ["Category","Title","Description"]
```

```python

train = pd.read_csv("/content/AG_train.csv", names = column_names, header = None
    )
test = pd.read_csv("/content/AG_test.csv", names = column_names, header = None)

train["Text"] = train["Title"] + " " + train["Description"]
test["Text"] = test["Title"] + " " + test["Description"]
train.drop(columns=["Title", "Description"], inplace=True)
test.drop(columns=["Title", "Description"], inplace=True)

#Prepocessing
nltk.download("stopwords")
stop_words = set(stopwords.words("english"))

def preprocess_text(sentence):
    sentence = re.sub(r"[^\w\s]", "", sentence)
    tokens = sentence.lower().split()
    tokens = [word for word in tokens if word not in stop_words]
    return " ".join(tokens)

train["Text"] = train["Text"].apply(preprocess_text)
test["Text"] = test["Text"].apply(preprocess_text)

#Creating vocabulary
all_tokens = []
for text in train["Text"]:
    all_tokens.extend(text.split())  # Tokenizing by space

# Get unique words
unique_tokens = set(all_tokens)
print(f"Total unique words: {len(unique_tokens)}")

# Initialize vocab with a padding token
vocab = {"<pad>": 0}  # Start indexing from 1
index = 1

# Assign an index to each unique word
for token in unique_tokens:
```

```
    vocab[token] = index
    index += 1

vocab_size = len(vocab)  # Number of unique words including <pad>
print(f"Vocabulary size: {vocab_size}")

# Reverse dictionary to get index  word mapping
inverse_vocab = {index: token for token, index in vocab.items()}

# Convert dataset text into sequences of word indices
train["Sequences"] = train["Text"].apply(lambda x: [vocab[word] for word in x.
    split() if word in vocab])
test["Sequences"] = test["Text"].apply(lambda x: [vocab[word] for word in x.
    split() if word in vocab])

# Show an example
print(train[["Text", "Sequences"]].head())

# Generates skip-gram pairs with negative sampling for a list of sequences
# (int-encoded sentences) based on window size, number of negative samples
# and vocabulary size.
def generate_training_data(sequences, window_size, num_ns, vocab_size, seed):
    # Elements of each training example are appended to these lists.
    targets, contexts, labels = [], [], []

    # Build the sampling table for `vocab_size` tokens.
    sampling_table = tf.keras.preprocessing.sequence.make_sampling_table(
        vocab_size)

    # Iterate over all sequences (sentences) in the dataset.
    for sequence in tqdm.tqdm(sequences):

        # Generate positive skip-gram pairs for a sequence (sentence).
        positive_skip_grams, _ = tf.keras.preprocessing.sequence.skipgrams(
              sequence,
              vocabulary_size=vocab_size,
              sampling_table=sampling_table,
              window_size=window_size,
```

```
              negative_samples=0,
              seed=seed)

        # Iterate over each positive skip-gram pair to produce training examples
        # with a positive context word and negative samples.
        for target_word, context_word in positive_skip_grams:
            context_class = tf.expand_dims(
              tf.constant([context_word], dtype="int64"), 1)
            negative_sampling_candidates, _, _ = tf.random.log_uniform_candidate_sampler(
              true_classes=context_class,
              num_true=1,
              num_sampled=num_ns,
              unique=True,
              range_max=vocab_size,
              seed=seed,
              name="negative_sampling")

          # Build context and label vectors (for one target word)
            context = tf.concat([tf.squeeze(context_class,1), negative_sampling_candidates], 0)
            label = tf.constant([1] + [0]*num_ns, dtype="int64")

          # Append each element from the training example to global lists.
            targets.append(target_word)
            contexts.append(context)
            labels.append(label)

    return targets, contexts, labels

#targets, contexts, labels = generate_training_data(
#    sequences=train["Sequences"],
#    window_size=3,
#    num_ns=4,
#    vocab_size=vocab_size,
#    seed=407)

#targets = np.array(targets)
```

```
#contexts = np.array(contexts)
#labels = np.array(labels)

#print('\n')
#print(f"targets.shape: {targets.shape}")
#print(f"contexts.shape: {contexts.shape}")
#print(f"labels.shape: {labels.shape}")

BATCH_SIZE = 200
BUFFER_SIZE = 5000
AUTOTUNE = tf.data.AUTOTUNEa

dataset = tf.data.Dataset.from_tensor_slices(((targets, contexts), labels))
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE, drop_remainder=True)
dataset = dataset.cache().prefetch(buffer_size=AUTOTUNE)

#Word2Vec Model
class Word2Vec(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim):
        super(Word2Vec, self).__init__()
        self.target_embedding = layers.Embedding(vocab_size,
                                      embedding_dim,
                                      input_length=1,
                                      name="w2v_embedding")
        self.context_embedding = layers.Embedding(vocab_size,
                                       embedding_dim,
                                       input_length=num_ns+1)

    def call(self, pair):
        target, context = pair
        # target: (batch, )
        # context: (batch, context)
        if len(target.shape) == 2:
            target = tf.squeeze(target, axis=1)
        # target: (batch,)
        word_emb = self.target_embedding(target)
        # word_emb: (batch, embed)
        context_emb = self.context_embedding(context)
```

```
        # context_emb: (batch, context, embed)
        dots = tf.einsum('be,bce->bc', word_emb, context_emb) # 'be, bce ->'
            indicates the output shape
        # dots: (batch, context)
        return tf.nn.softmax(dots)

num_ns=4
embedding_dim = 100
word2vec = Word2Vec(vocab_size, embedding_dim)
word2vec.compile(optimizer='adam',
                 loss="CategoricalCrossentropy",
                 metrics=['accuracy'])

#weights = word2vec.get_layer('w2v_embedding').get_weights()[0]

MAX_SEQUENCE_LENGTH = 100
NUM_CLASSES = len(train["Category"].unique())

train_labels = train["Category"].values -1

padded_sequences = pad_sequences(train["Sequences"], maxlen=MAX_SEQUENCE_LENGTH,
     padding="post")

X_train, X_test, y_train, y_test = train_test_split(padded_sequences,
    train_labels, test_size=0.2, random_state=42)
y_train = tf.keras.utils.to_categorical(y_train, num_classes=NUM_CLASSES)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=NUM_CLASSES)

#Model 1: Deep learning model using skip-gram word embedding

def model_skip(hp):
    input_layer = Input(shape=(MAX_SEQUENCE_LENGTH,), name="Input_Layer")
    embedding_layer = Embedding(input_dim=weights.shape[0],
                                output_dim=weights.shape[1],
                                weights=[weights], trainable=False)(input_layer)

    dropout_rate = hp.Choice("dropout_rate", [0.1, 0.2, 0.3])

```

```
    # First BiGRU layer
    BI_GRU_1 = tf.keras.layers.Bidirectional(GRU(units=hp.Int("GRU_unit_1",
        min_value=64, max_value=256, step=64),
                                                    return_sequences=True,
                                                        activation="leaky_relu"))(
                                                        embedding_layer)
    BI_GRU_1 = Dropout(dropout_rate)(BI_GRU_1)

    # Second BiGRU layer
    BI_GRU_2 = tf.keras.layers.Bidirectional(GRU(units=hp.Int("GRU_unit_2",
        min_value=64, max_value=192, step=64),
                                                    return_sequences=True,
                                                        activation="leaky_relu"))(
                                                        BI_GRU_1)
    BI_GRU_2 = Dropout(dropout_rate)(BI_GRU_2)

    # Third BiGRU layer
    BI_GRU_3 = tf.keras.layers.Bidirectional(GRU(units=hp.Int("GRU_unit_3",
        min_value=64, max_value=128, step=64),
                                                    return_sequences=False,
                                                        activation="leaky_relu"))(
                                                        BI_GRU_2)
    BI_GRU_3 = Dropout(dropout_rate)(BI_GRU_3)

    # Dense layer
    dense_1 = Dense(units=hp.Int("dense_unit_1", min_value=64, max_value=256,
        step=64), activation="leaky_relu")(BI_GRU_3)
    dense_1 = Dropout(dropout_rate)(dense_1)

    # Output layer
    output_layer = Dense(NUM_CLASSES, activation="softmax")(dense_1)

    model = tf.keras.Model(inputs=input_layer, outputs=output_layer, name="
        Text_Classification_RNN")

    model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["
        accuracy"])

```

```
    return model

early_stopping = EarlyStopping(
    monitor="val_loss",
    patience=3,
    restore_best_weights=True
)

tuner = kt.RandomSearch(
      model_skip,
      objective="val_loss",
      max_trials=10,
      executions_per_trial = 1
  )

tuner.search(
    X_train,
    y_train,
    epochs = 50,
    batch_size = 512,
    validation_data = (X_test, y_test),
    callbacks=[early_stopping]
)

best_hyperparams_skip = tuner.get_best_hyperparameters(num_trials=1)[0]
best_model_skip = tuner.hypermodel.build(best_hyperparams_skip)
history_skip = best_model_skip.fit(X_train, y_train,
               epochs = 50, batch_size = 512,
               validation_data = (X_test, y_test),
               callbacks=[early_stopping]
              )

best_model_skip.summary()

best_hparams = tuner.get_best_hyperparameters(num_trials=1)[0]

# Print the best hyperparameters
for param in best_hparams.values:
```

```
    print(f"{param}: {best_hparams.get(param)}")

from sklearn.metrics import classification_report, f1_score, precision_score,
    recall_score, confusion_matrix

# Predictions
y_pred_prob = best_model_skip.predict(X_test)
y_pred = np.argmax(y_pred_prob, axis=1)
y_true = np.argmax(y_test, axis=1)

print("\nClassification Report:")
print(classification_report(y_true, y_pred))

class_names = ["World", "Sports", "Business", "Sci/Tech"]
# Confusion Matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Greens", xticklabels=class_names,
    yticklabels=class_names)

plt.xlabel("Predicted Class")
plt.ylabel("True Class")
plt.title("Confusion Matrix")
plt.show()

path_to_glove_file = "glove.6B.100d.txt"

embeddings_index = {}
with open(path_to_glove_file) as f:
    for line in f:
        word, coefs = line.split(maxsplit=1)
        coefs = np.fromstring(coefs, "f", sep=" ")
        embeddings_index[word] = coefs

print("Found %s word vectors." % len(embeddings_index))

embedding_dim = 100
vocab_size = len(vocab)
```

```python

glove_embedding_matrix = np.zeros((vocab_size, embedding_dim))

for word, i in vocab.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        glove_embedding_matrix[i] = embedding_vector

#Model 2: Deep learning model using glove word embedding

def model_glove(hp):
    input_layer = Input(shape=(MAX_SEQUENCE_LENGTH,), name="Input_Layer")
    embedding_layer = Embedding(input_dim=weights.shape[0],
                                output_dim=weights.shape[1],
                                weights=[glove_embedding_matrix], trainable=False)(input_layer)

    dropout_rate = hp.Choice("dropout_rate", [0.1, 0.2, 0.3])

    # First BiGRU layer
    BI_GRU_1 = tf.keras.layers.Bidirectional(GRU(units=hp.Int("GRU_unit_1", min_value=64, max_value=256, step=64),
                                                 return_sequences=True, activation="leaky_relu"))(embedding_layer)
    BI_GRU_1 = Dropout(dropout_rate)(BI_GRU_1)

    # Second BiGRU layer
    BI_GRU_2 = tf.keras.layers.Bidirectional(GRU(units=hp.Int("GRU_unit_2", min_value=64, max_value=192, step=64),
                                                 return_sequences=True, activation="leaky_relu"))(BI_GRU_1)
    BI_GRU_2 = Dropout(dropout_rate)(BI_GRU_2)

    # Third BiGRU layer
    BI_GRU_3 = tf.keras.layers.Bidirectional(GRU(units=hp.Int("GRU_unit_3", min_value=64, max_value=128, step=64),
```

```
                                                   return_sequences=False,
                                                   activation="leaky_relu"))(BI_GRU_2)
    BI_GRU_3 = Dropout(dropout_rate)(BI_GRU_3)

    # Dense layer
    dense_1 = Dense(units=hp.Int("dense_unit_1", min_value=64, max_value=256, step=64), activation="leaky_relu")(BI_GRU_3)
    dense_1 = Dropout(dropout_rate)(dense_1)

    # Output layer
    output_layer = Dense(NUM_CLASSES, activation="softmax")(dense_1)

    model = tf.keras.Model(inputs=input_layer, outputs=output_layer, name="Text_Classification_RNN")

    model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"])

    return model

early_stopping = EarlyStopping(
    monitor="val_loss",
    patience=3,
    restore_best_weights=True
)

tuner = kt.RandomSearch(
      model_glove,
      objective="val_loss",
      max_trials=10,
      executions_per_trial = 1
  )

tuner.search(
    X_train,
    y_train,
    epochs = 50,
```

```
    batch_size = 512,
    validation_data = (X_test, y_test),
    callbacks=[early_stopping]
)

best_hyperparams_glove = tuner.get_best_hyperparameters(num_trials=1)[0]
best_model_glove = tuner.hypermodel.build(best_hyperparams_glove)
history_glove = best_model_glove.fit(X_train, y_train,
               epochs = 50, batch_size = 512,
               validation_data = (X_test, y_test),
               callbacks=[early_stopping])

best_model_glove.summary()

best_hparams = tuner.get_best_hyperparameters(num_trials=1)[0]

# Print the best hyperparameters
for param in best_hparams.values:
    print(f"{param}: {best_hparams.get(param)}")

# Predictions
y_pred_prob = best_model_glove.predict(X_test)
y_pred = np.argmax(y_pred_prob, axis=1)
y_true = np.argmax(y_test, axis=1)

print("\nClassification Report:")
print(classification_report(y_true, y_pred))

class_names = ["World", "Sports", "Business", "Sci/Tech"]
# Confusion Matrix
cm = confusion_matrix(y_true, y_pred)

plt.figure(figsize=(8,6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Reds", xticklabels=class_names,
    yticklabels=class_names)
plt.xlabel("Predicted Class")
plt.ylabel("True Class")
plt.title("Confusion Matrix")
```

```python
plt.show()

# Task 2

import numpy as np
import pandas as pd
import os
import random
import re
import tensorflow as tf
import tqdm
from tensorflow.keras import layers
from collections import Counter
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
import nltk
from sklearn.feature_extraction.text import TfidfVectorizer
import scipy.cluster.hierarchy as sch
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans


random.seed(407)

file_path = "french-theater.txt"

with open(file_path, "r", encoding="utf-8") as f:
    text = f.read()

#Splitting the text and sorting them into a dictionary
plays = re.split(r'###(\d{2})###\n', text)[1:]
plays_dict = {plays[i]: plays[i+1].strip().split("#######")[0] for i in range(0,
    len(plays), 2)}

#Test
print(f"\nPlay {1}:")
```

```
print(play_text[:500])

def preprocess_text(text):
    text = text.lower()
    text = re.sub(r"[^\w\s]", "", text)  # Remove punctuation
    tokens = word_tokenize(text)
    tokens = [t for t in tokens if t not in stopwords.words("french")]  # Remove stopwords
    return tokens

document_texts = []
for plays in plays_dict.values():
        document_texts.append(preprocess_text(plays))

document_texts_joined = [" ".join(tokens) for tokens in document_texts]

tfidf_vectorizer = TfidfVectorizer(max_features=5000)

tfidf_matrix = tfidf_vectorizer.fit_transform(document_texts_joined)
tfidf_matrix_dense = tfidf_matrix.toarray()

linkage_matrix = sch.linkage(tfidf_matrix_dense, method='ward')

plt.figure(figsize=(10, 5))
sch.dendrogram(linkage_matrix, labels=list(plays_dict.keys()), leaf_rotation=90)
plt.title("Hierarchical Clustering Dendrogram")
plt.xlabel("Plays")
plt.ylabel("Distance")
plt.show()

num_clusters = 2
kmeans = KMeans(n_clusters=num_clusters, random_state=407, n_init=10)

kmeans.fit(tfidf_matrix_dense)
cluster_labels = kmeans.labels_

# Clusters
for play, label in zip(plays_dict.keys(), cluster_labels):
```

```python
    print(f"Play {play} -> Cluster {label}")

pca = PCA(n_components=2)
tfidf_pca = pca.fit_transform(tfidf_matrix_dense)

plt.figure(figsize=(8,6))
plt.scatter(tfidf_pca[:,0], tfidf_pca[:,1], c=cluster_labels, cmap="coolwarm",
    edgecolors="k")
for i, play_num in enumerate(plays_dict.keys()):
    plt.annotate(play_num, (tfidf_pca[i,0], tfidf_pca[i,1]))

plt.title("PCA Projection of Plays")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.show()
```

Appendix B: Generative AI usage

ChatGPT was used to brainstorm and to help me develop my ideas for approaching the tasks, also as an alternative to google for debugging code.