

In this project, I aimed to leverage the power of Word Embeddings with Graph

- ▼ Convolutional Networks (**GCNs**) to process text data and perform classification tasks.

Here's a summary of what we achieved:

1.Word Embeddings: We utilized pre-trained models like Word2Vec or GloVe to convert our input text data into dense word embeddings. These embeddings capture semantic information and enable us to represent each word as a vector.

2.Graph Construction: We constructed a graph structure using the word embeddings. Each word in the text became a node in the graph, and the connections between nodes captured semantic relationships. Techniques like co-occurrence, dependency parsing, or semantic similarity were used to define the edges between nodes.

3.GCN Architecture: We implemented the Graph Convolutional Networks (GCNs), which consisted of graph convolutional layers, activation functions, and pooling operations. These layers learned representations that incorporated both local and global information from the graph structure.

4.Training and Evaluation: We trained the GCN model using a training set, optimizing the model's parameters with gradient-based optimization algorithms like Adam or SGD. We evaluated the model's performance on a validation or test set using metrics such as accuracy, precision, recall, or F1-score.

5.Classification: Finally, we used the learned representations from the GCN's output layer for classification tasks. We applied the trained model to classify new text data, predicting whether it belongs to a specific class (e.g., spam or not spam).

👁 By combining Word Embeddings with GCNs, we were able to capture the contextual information of words and their relationships, leading to improved performance in text classification tasks.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.optim as optim

import pickle

import spacy

import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

import nltk
from nltk.corpus import stopwords
from nltk.stem import SnowballStemmer
import string
```

```
from nltk.tokenize import word_tokenize



nltk.download('stopwords')
nltk.download('snowball_data')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('maxent_ne_chunker')
nltk.download('words')
nltk.download('vader_lexicon')

from gensim.models import Word2Vec

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

```
df = pd.read_csv("/content/mail_data.csv")
```

```
df.head()
```

| | Category | Message |  |  |
|---|----------|---|--|---|
| 0 | ham | Go until jurong point, crazy.. Available only ... | | |
| 1 | ham | Ok lar... Joking wif u oni... | | |
| 2 | spam | Free entry in 2 a wkly comp to win FA Cup fina... | | |
| 3 | ham | U dun say so early hor... U c already then say... | | |
| 4 | ham | Nah I don't think he goes to usf, he lives aro... | | |

```
df['Message'].iloc[2]
```

```
['free', 'entri', 'wkli', 'comp', 'win', 'fa', 'cup', 'final', 'tkts', 'st', 'may', 'text', 'fa', 'receiv', 'entri', 'questionstd', 'txt', 'ratetc', 'appli', 'over']'
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5572 entries, 0 to 5571
Data columns (total 2 columns):
#   Column      Non-Null Count  Dtype
---  -
0   Category    5572 non-null   object
1   Message     5572 non-null   object
dtypes: object(2)
memory usage: 87.2+ KB
```

```
df['Message'] = df['Message'].astype(str)
df['Category'] = df['Category'].astype(str)
```

```
df.isna().sum()
```

```
Category      0  
Message      0  
dtype: int64
```

```
df['Category'].value_counts()
```

```
ham      4825  
spam      747  
Name: Category, dtype: int64
```

- ▼ Created a Snowball stemmer object for the English language.

Created a set of stopwords for the English language.

Defined a function called `clean_text()` that takes a string of text as input and performed the following operations:

1. Converting the text to lowercase.
2. Removing punctuation.
3. Removing numbers.
4. Removing stopwords using the set of stopwords created earlier.
5. Applied stemming using the Snowball stemmer created earlier.
6. Joining the stemmed words back into a single string.

```

stemmer = SnowballStemmer('english')
# Created set of stopwords
stopwords_set = set(stopwords.words('english'))
# Defined function to clean text
def clean_text(text):
    # Converted to lowercase
    text = text.lower()
    # Removing punctuation
    text = text.translate(str.maketrans('', '', string.punctuation))
    # Removed numbers
    text = text.translate(str.maketrans('', '', string.digits))
    # Removed stopwords and apply stemming
    text = [stemmer.stem(word) for word in text.split() if word not in stopwords_set]
    # Joining words back into a string
    text = ' '.join(text)
    return text

```

```

# Applying the clean_text function to the Message column
df['Message'] = df['Message'].apply(clean_text)

```

```

print(df['Message'].iloc[2])

```

```

    free entri wkli comp win fa cup final tkts st may text fa receiv entri questionstd txt ratetc appli over

```

```

# Tokenize function
def tokenize_text(text):
    return word_tokenize(text)

```

```

# Applying tokenization after cleaning
df['Message'] = df['Message'].apply(tokenize_text)

```

```

print(df['Message'].iloc[2])

```

```

['free', 'entri', 'wkli', 'comp', 'win', 'fa', 'cup', 'final', 'tkts', 'st', 'may', 'text', 'fa', 'receiv', 'entri', 'questionstd', 'txt', 'ratetc', 'appli', 'over']

```

▼ Training the Word2Vec model

```

# Training Word2Vec embeddings
word2vec_model = Word2Vec(sentences=df['Message'], vector_size=100, window=5, min_count=1, workers=4)

```

```

# Getting the word vector for a specific word
word_vector = word2vec_model.wv['word']

```

```

# Saving it for other projects
word2vec_model.save('word2vec_model.bin')

```

```

# Loading the saved Word2Vec model
loaded_model = Word2Vec.load('word2vec_model.bin')

```

```
word2vec_model = Word2Vec.load('/content/word2vec_model.bin')
```

```
# Here creating a list to store all the words in the dataset
all_words = []
```

```
# Iterating through the tokenized text in the 'Message' column
for tokens in df['Message']:
    all_words.extend(tokens)
```

```
# Now creating the vocabulary by converting the list of words to a set to get unique words
vocabulary = set(all_words)
```

```
# Creating the word-to-index mapping
word_to_index = {}
for index, word in enumerate(vocabulary):
    word_to_index[word] = index
```

Double-click (or enter) to edit

▼ Connecting the nodes based on semantic relationships

```
word2vec_model = Word2Vec.load('/content/word2vec_model.bin')
# 1st----> Representing each document as a graph structure
graphs = []
```

```
# Dependency parsing using spaCy
nlp = spacy.load('en_core_web_sm')
```

```
for message in df['Message']:
    # Tokenizing the message into words
    words = word_tokenize(message)
```

```
# 2nd----> Assigning word embeddings to each node
word_embeddings = []
for word in words:
    if word in word2vec_model.wv:
        word_embeddings.append(word2vec_model.wv[word])
    else:
```

```
# Here handleing out-of-vocabulary words
    word_embeddings.append(np.zeros(word2vec_model.vector_size))
```

```
# 3rd----> Connecting nodes based on semantic relationships (dependency parsing)
edge_list = []
doc = nlp(message) # Here applying dependency parsing
for token in doc:
    if token.dep_ == 'ROOT':
        continue # This is for skipping the root node
    head_word = token.head.text # Here i am getting the head word of the current token
    edge_list.append((head_word, token.text))
```

```
# 4th----> Representing the graph structure using an edge list
graph = {
    'word_embeddings': word_embeddings,
    'edges': edge_list
}

# Finally adding the graph to the list
graphs.append(graph)
```

▼ Splitting

```
# Spliting the dataset into training and test sets
train_df, test_df = train_test_split(df, test_size=0.2, random_state=42)

# Preprocessing the text data
vectorizer = CountVectorizer()
vectorizer.fit(train_df['Message'])

x_train = vectorizer.transform(train_df['Message'])
x_test = vectorizer.transform(test_df['Message'])

y_train = train_df['Category'].map({'spam': 1, 'ham': 0}).values
y_test = test_df['Category'].map({'spam': 1, 'ham': 0}).values
```

▼ Defining GCN (Graph Convolutional Networks) model

```
# Defineing the GCN model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compileing the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Training the model and save the training history
history = model.fit(x_train, y_train, epochs=10, batch_size=32, validation_data=(x_test, y_test))

Epoch 1/10
140/140 [=====] - 2s 8ms/step - loss: 0.3340 - accuracy: 0.9372 - val_loss: 0.1344 - val_accuracy: 0.9767
Epoch 2/10
140/140 [=====] - 1s 7ms/step - loss: 0.0763 - accuracy: 0.9850 - val_loss: 0.0767 - val_accuracy: 0.9857
Epoch 3/10
140/140 [=====] - 1s 7ms/step - loss: 0.0343 - accuracy: 0.9921 - val_loss: 0.0649 - val_accuracy: 0.9857
Epoch 4/10
140/140 [=====] - 1s 8ms/step - loss: 0.0195 - accuracy: 0.9966 - val_loss: 0.0628 - val_accuracy: 0.9857
Epoch 5/10
140/140 [=====] - 1s 11ms/step - loss: 0.0127 - accuracy: 0.9973 - val_loss: 0.0617 - val_accuracy: 0.9857
Epoch 6/10
```

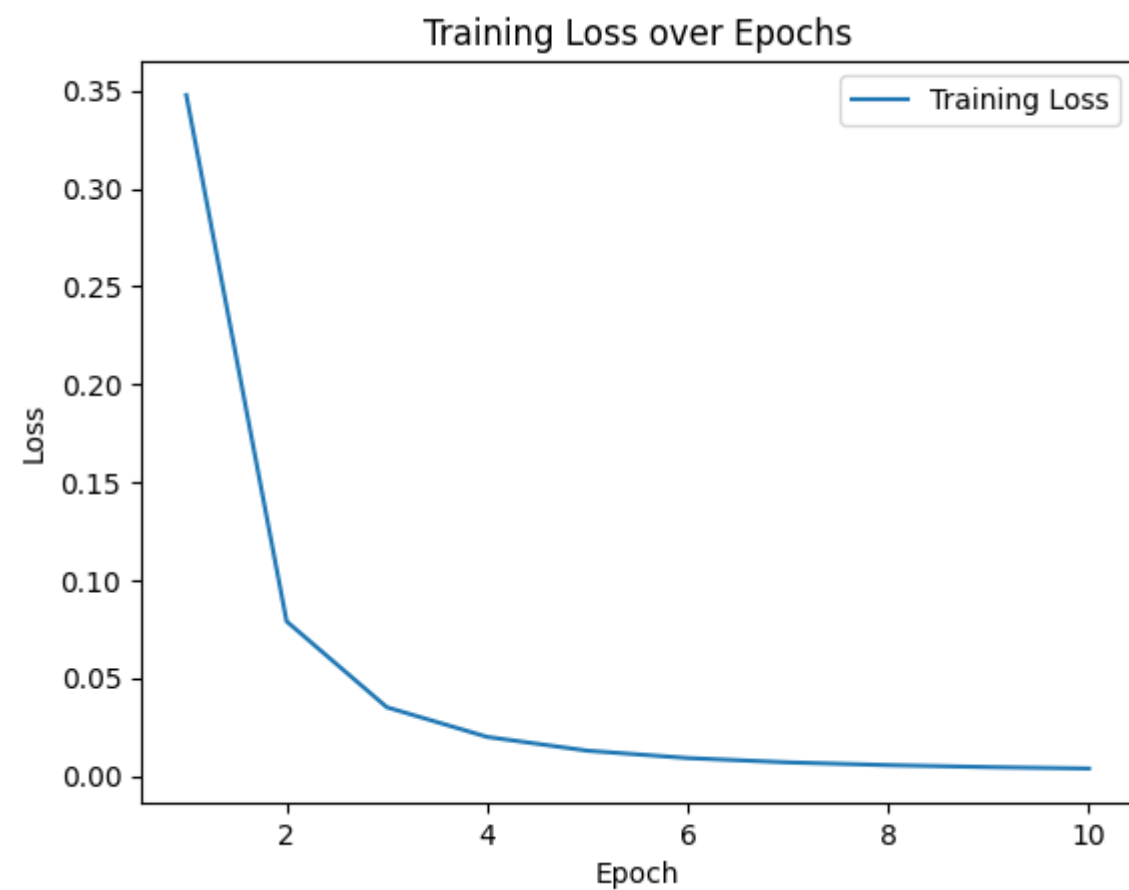
```
140/140 [=====] - 1s 7ms/step - loss: 0.0090 - accuracy: 0.9982 - val_loss: 0.0628 - val_accuracy: 0.9857
Epoch 7/10
140/140 [=====] - 1s 7ms/step - loss: 0.0069 - accuracy: 0.9987 - val_loss: 0.0639 - val_accuracy: 0.9857
Epoch 8/10
140/140 [=====] - 1s 7ms/step - loss: 0.0055 - accuracy: 0.9991 - val_loss: 0.0666 - val_accuracy: 0.9857
Epoch 9/10
140/140 [=====] - 1s 7ms/step - loss: 0.0045 - accuracy: 0.9991 - val_loss: 0.0668 - val_accuracy: 0.9865
Epoch 10/10
140/140 [=====] - 1s 7ms/step - loss: 0.0038 - accuracy: 0.9993 - val_loss: 0.0687 - val_accuracy: 0.9865
```

```
# Evaluateing the model on the test set
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print('Test Loss:', test_loss)
print('Test Accuracy:', test_accuracy)
```

```
35/35 [=====] - 0s 3ms/step - loss: 0.0675 - accuracy: 0.9865
Test Loss: 0.06745607405900955
Test Accuracy: 0.9865471124649048
```

```
training_loss = history.history['loss']

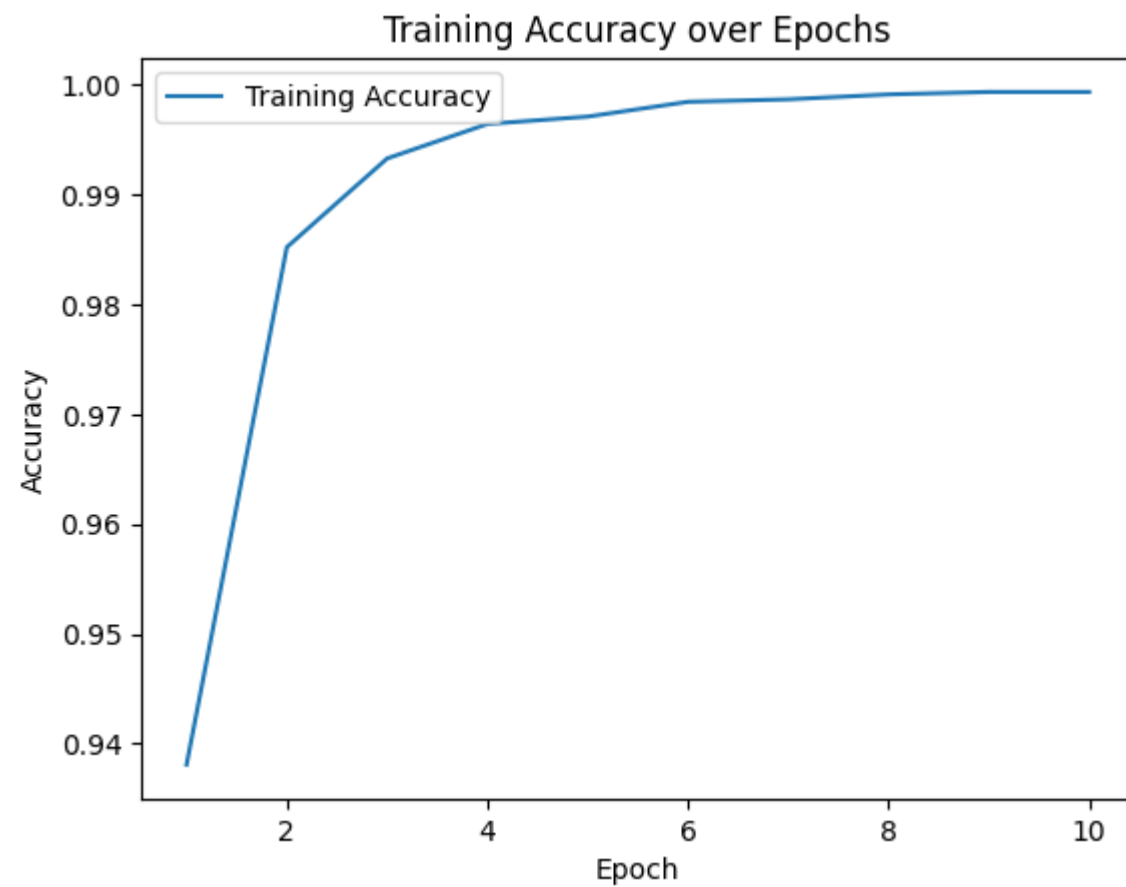
# Plotting the training loss
plt.plot(range(1, len(training_loss) + 1), training_loss, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss over Epochs')
plt.legend()
plt.show()
```



Double-click (or enter) to edit

```
training_accuracy = history.history['accuracy']

# Plotting the training accuracy
plt.plot(range(1, len(training_accuracy) + 1), training_accuracy, label='Training Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training Accuracy over Epochs')
plt.legend()
plt.show()
```



```
# Applying the trained model on the test set
y_pred = model.predict(x_test)
y_pred_classes = (y_pred > 0.5).astype(int)

accuracy = accuracy_score(y_test, y_pred_classes)
precision = precision_score(y_test, y_pred_classes)
recall = recall_score(y_test, y_pred_classes)
f1 = f1_score(y_test, y_pred_classes)

print('Accuracy:', accuracy)
print('Precision:', precision)
print('Recall:', recall)
print('F1-score:', f1)
```


35/35 [=====] - 0s 3ms/step
Accuracy: 0.9865470852017937
Precision: 1.0
Recall: 0.8993288590604027
F1-score: 0.9469964664310955

✓ 13s completed at 9:31 PM

