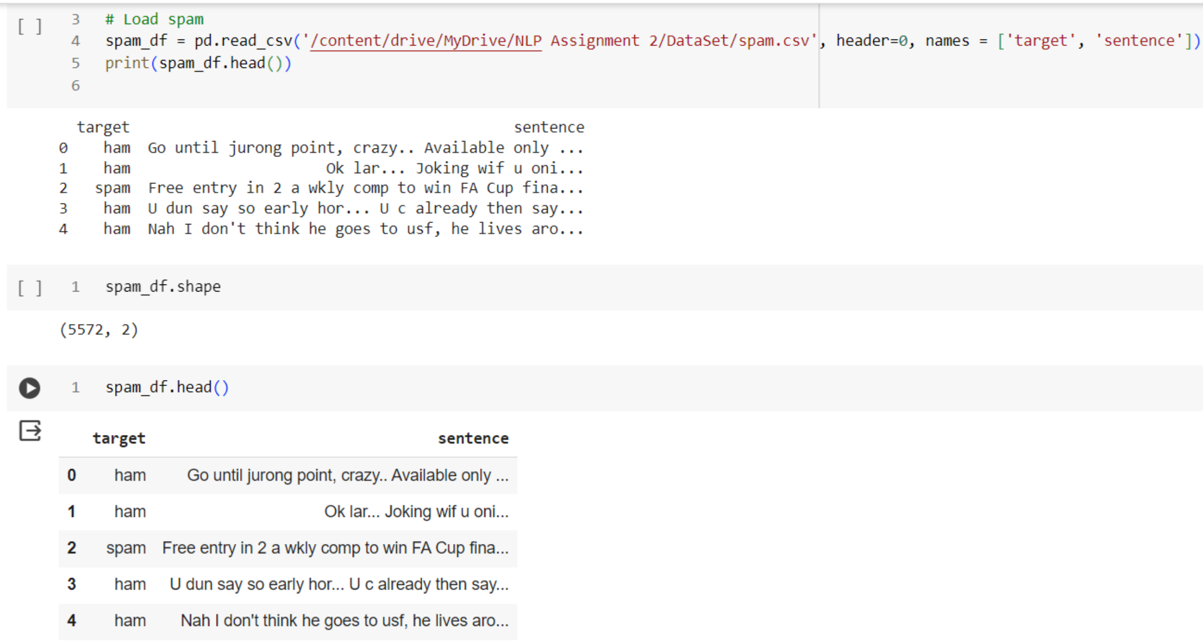
Natural Language Processing

Assignment - 3

Dataset

Like the last assignment, the dataset used is a collection of text messages labeled as ‘spam’ or ‘ham’, intended for spam detection. It is loaded from a CSV file, where each row corresponds to a message alongside its classification (spam or ham). The shape of the dataset is **5572 samples and 2 features** which we renamed as ‘target’ and ‘sentence’.



```
# Load spam
spam_df = pd.read_csv('/content/drive/MyDrive/NLP Assignment 2/DataSet/spam.csv', header=0, names = ['target', 'sentence'])
print(spam_df.head())

```

```
  target                                           sentence
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...
```

```
spam_df.shape
```

```
(5572, 2)
```

```
spam_df.head()
```

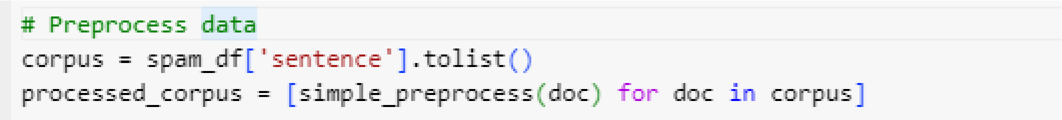
	target	sentence
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

Dataset Link: https://drive.google.com/file/d/1aCGEe3gHAw5rxJCYi-qW1eNZV42hIXzb/view?usp=drive_link

Preprocessing

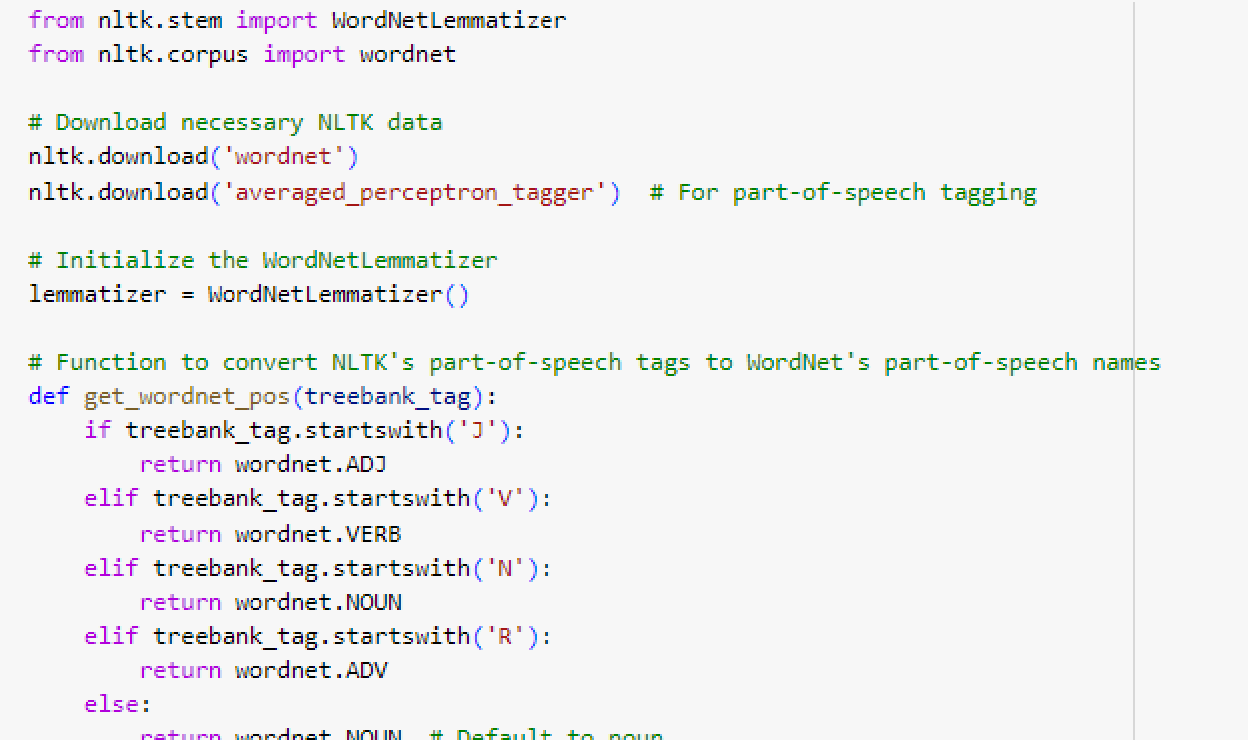
The preprocessing steps depended on the Word2VecModel:

- **Custom Word2Vec Model:** We used the simple_preprocess() in the gensim library.

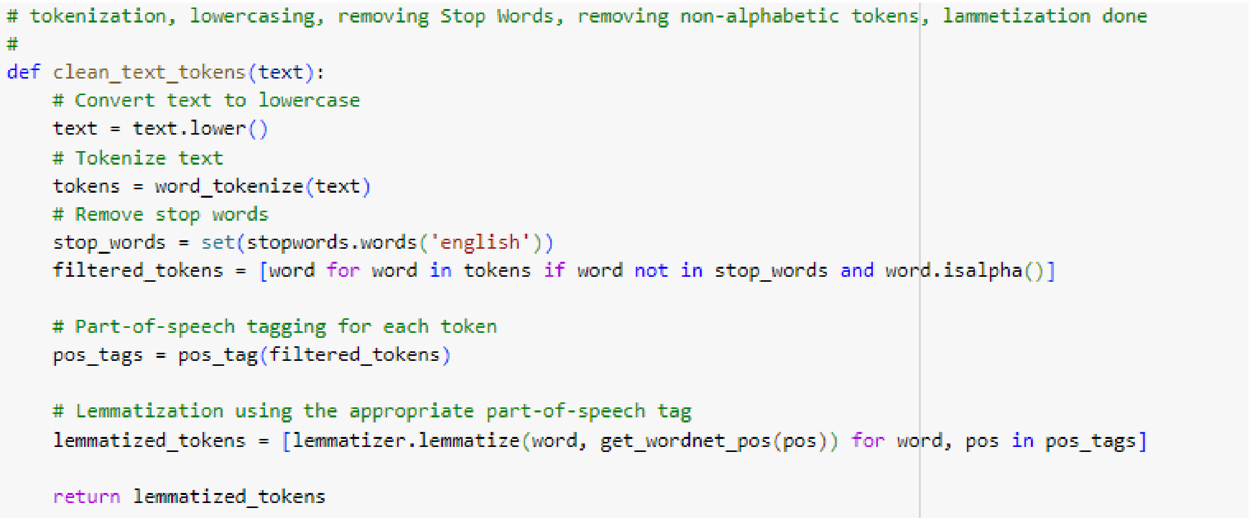


```
# Preprocess data
corpus = spam_df['sentence'].tolist()
processed_corpus = [simple_preprocess(doc) for doc in corpus]
```

- For **Pre-Trained Word2Vec** and **Pre-Trained GloVe** models: We directly applied tokenization, lowercasing, removed stop words, and removed non-alphabetic tokens from our model using our standardized pre-processing for past projects.



```
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet

# Download necessary NLTK data
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')  # For part-of-speech tagging

# Initialize the WordNetLemmatizer
lemmatizer = WordNetLemmatizer()

# Function to convert NLTK's part-of-speech tags to WordNet's part-of-speech names
def get_wordnet_pos(treebank_tag):
    if treebank_tag.startswith('J'):
        return wordnet.ADJ
    elif treebank_tag.startswith('V'):
        return wordnet.VERB
    elif treebank_tag.startswith('N'):
        return wordnet.NOUN
    elif treebank_tag.startswith('R'):
        return wordnet.ADV
    else:
        return wordnet.NOUN  # Default to noun
```



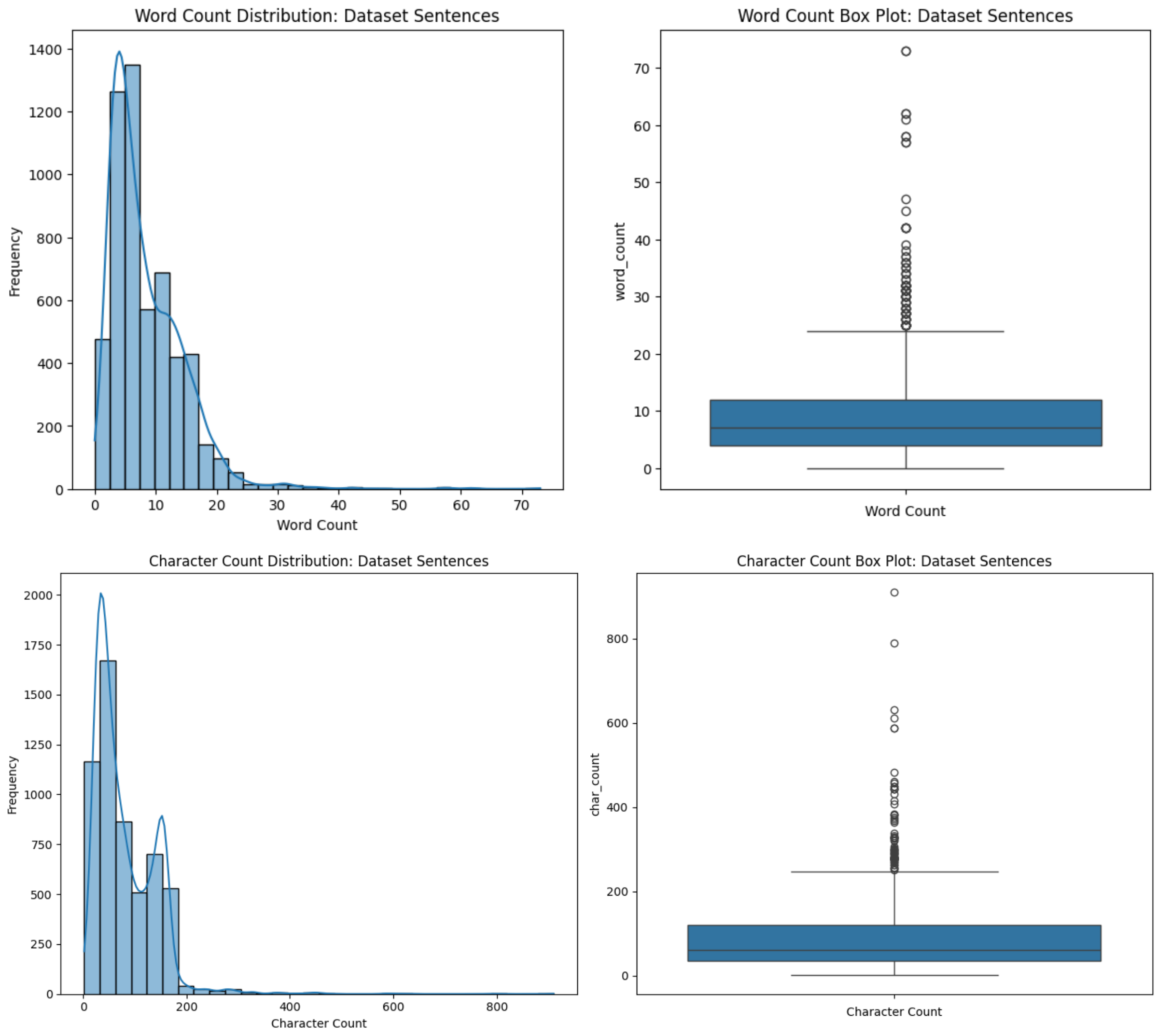
```
# tokenization, lowercasing, removing Stop Words, removing non-alphabetic tokens, lammetization done
#
def clean_text_tokens(text):
    # Convert text to lowercase
    text = text.lower()
    # Tokenize text
    tokens = word_tokenize(text)
    # Remove stop words
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [word for word in tokens if word not in stop_words and word.isalpha()]

    # Part-of-speech tagging for each token
    pos_tags = pos_tag(filtered_tokens)

    # Lemmatization using the appropriate part-of-speech tag
    lemmatized_tokens = [lemmatizer.lemmatize(word, get_wordnet_pos(pos)) for word, pos in pos_tags]

    return lemmatized_tokens
```

Vocabulary Size and Distribution Analysis

The following information is unchanged from our Assignment 2 Report

The **vocabulary size after lemmatization** was found to be **6,233 unique words**.

This indicates a fairly large lexicon for the given dataset of 5,572 messages, which means a wide variety of language use within the corpus.



Word Count Distribution: Dataset Sentences

Word Count Box Plot: Dataset Sentences

Character Count Distribution: Dataset Sentences

Character Count Box Plot: Dataset Sentences

The **histograms** for both word count and character count distributions are **skewed to the right**, indicating that most **messages contain fewer words and characters**, with the frequency tapering off as the number of words or characters increases.

Q1. Please train your own Word2Vec embedding. Please be specific about the dataset you use, the size of the corpus, the domain of the language, etc.

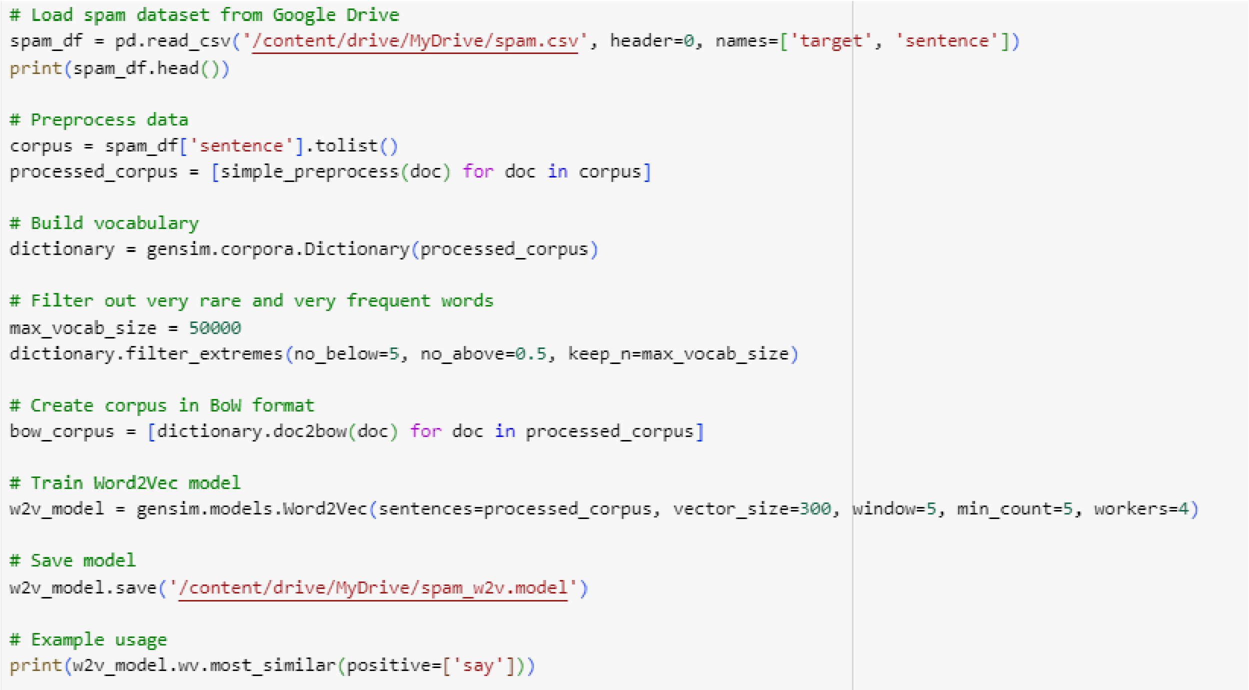
We trained our Word2Vec using the Gensim provided model and applying Gensim's simple_preprocess() to the dataset before providing it as input.

The below code shows the following steps:

- Mounts Google Drive to access the dataset stored in the Drive.
- Import the required libraries: pandas for data manipulation, and gensim for Word2Vec model training.
- Set up logging for Gensim.
- Load the Twitter spam dataset from the Google Drive into a pandas DataFrame df.

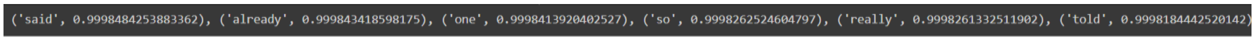
- Preprocesses the text data by converting it to a list and tokenizing using gensim.utils.simple_preprocess.
- Build a vocabulary dictionary from the processed corpus.
- Filter out very rare and very frequent words from the dictionary.
- Creates a Bag-of-Words (BoW) representation of the corpus using the dictionary.
- Train the Word2Vec skip-gram model using the processed corpus, with vector size 300, window size 5, minimum word count 5, and 4 worker threads.
- Saves the trained Word2Vec model to the Google Drive

Our Own Word2Vec (Using Gensim API)



```
# Load spam dataset from Google Drive
spam_df = pd.read_csv('/content/drive/MyDrive/spam.csv', header=0, names=['target', 'sentence'])
print(spam_df.head())

# Preprocess data
corpus = spam_df['sentence'].tolist()
processed_corpus = [simple_preprocess(doc) for doc in corpus]

# Build vocabulary
dictionary = gensim.corpora.Dictionary(processed_corpus)

# Filter out very rare and very frequent words
max_vocab_size = 50000
dictionary.filter_extremes(no_below=5, no_above=0.5, keep_n=max_vocab_size)

# Create corpus in BoW format
bow_corpus = [dictionary.doc2bow(doc) for doc in processed_corpus]

# Train Word2Vec model
w2v_model = gensim.models.Word2Vec(sentences=processed_corpus, vector_size=300, window=5, min_count=5, workers=4)

# Save model
w2v_model.save('/content/drive/MyDrive/spam_w2v.model')

# Example usage
print(w2v_model.wv.most_similar(positive=['say']))
```

- The w2v_model.wv.most_similar(positive=['say']) method finds the words in our Own_Word2Vec model's vocabulary that have the most similar vector representations (embeddings) to the word 'say'.
- The output is a list of tuples, where each tuple contains a word and its cosine similarity score with respect to the word 'say'.
- The list is sorted in descending order by the similarity scores, so the first tuple represents the word with the highest similarity to 'say'.



```
('said', 0.9998484253883362), ('already', 0.999843418598175), ('one', 0.9998413920402527), ('so', 0.9998262524604797), ('really', 0.9998261332511902), ('told', 0.9998184442520142)
```

The output shows that the words ['said', 'already', 'one', 'so', 'really', 'told'] has the highest cosine similarity scores to the word 'say' in our Own_Word2Vec model.

The **cosine similarity score**s range from 0 to 1, where a score closer to 1 indicates a higher similarity between the word vectors. In the output, all the similarity scores are close to 1, suggesting that these words are highly similar to 'say' in the vector space learned by our Own_Word2Vec model.

This output can be useful for understanding the semantic relationships captured by the Word2Vec model. Words that appear in similar contexts tend to have similar vector representations, so the most similar words to 'say' likely share some semantic or syntactic properties with the word 'say' in the training corpus.

For example, the presence of words like 'said', 'told', and 'make' in the top results suggests that the model has learned to associate 'say' with verbs related to speech or communication. The word 'him' could indicate that 'say' is often used in direct speech or dialogue contexts.
This output provides insights into the semantic and contextual information encoded in the Word2Vec embeddings, which can be valuable for various natural language processing tasks.

For future question**:**

- The pre-trained Word2Vec was **Fasttext Word2Vec** model
- **GloVe** model is **Stanford's GloVe Pre-trained** model - 6 billion tokens, 300 feature size

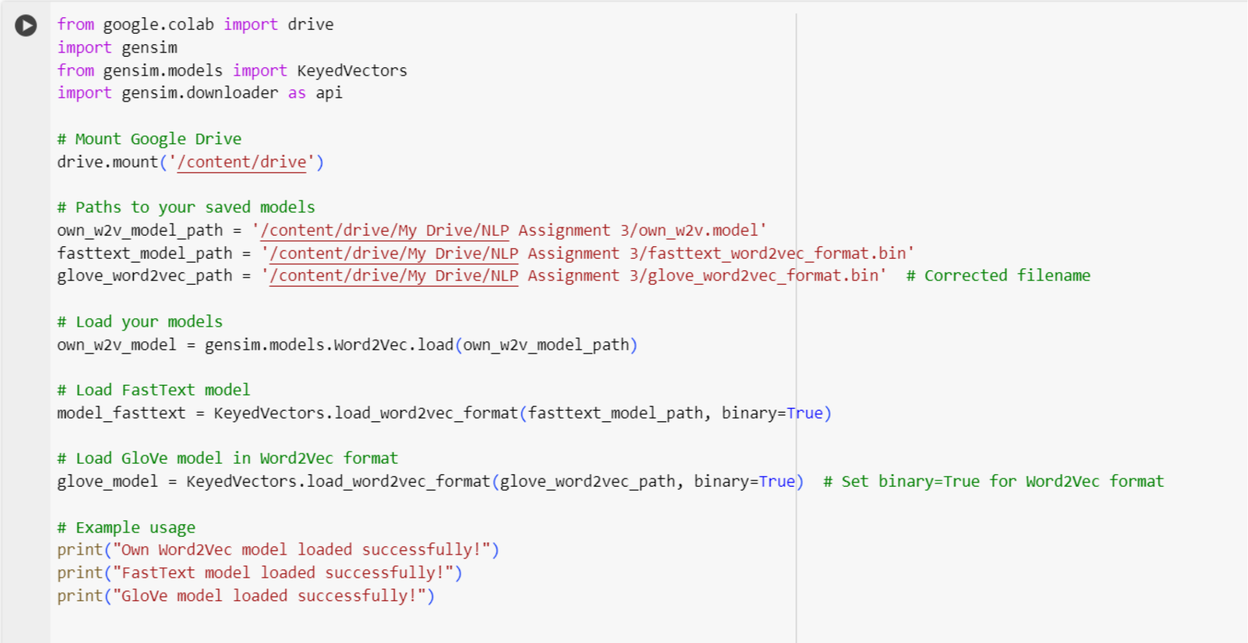
Q2a. Please compare your own Word2Vec embedding, pre-trained Word2Vec word embedding, and pre-trained GloVe word embedding via: Semantic distance calculation and visualization. Try to use interesting visualizations to assist in your analysis of the performance of each embedding at the word level

The code snippet below loads pre-trained word embedding models from Google Drive into a Google Colab environment. It begins by importing the necessary libraries, including gensim for working with word embeddings and google colab for mounting Google Drive.

The code then specifies the file paths for three pre-trained word embedding models: our own_Word2Vec model (own_w2v.model), a FastText model (fasttext_word2vec_format.bin), and a GloVe model (glove_word2vec_format.bin).

The gensim.models.Word2Vec.load function is used to load the custom Word2Vec model from the specified file path. The KeyedVectors.load_word2vec_format function from gensim is used to load both the FastText and GloVe models, with the binary=True parameter indicating that the models are in the Word2Vec binary format.

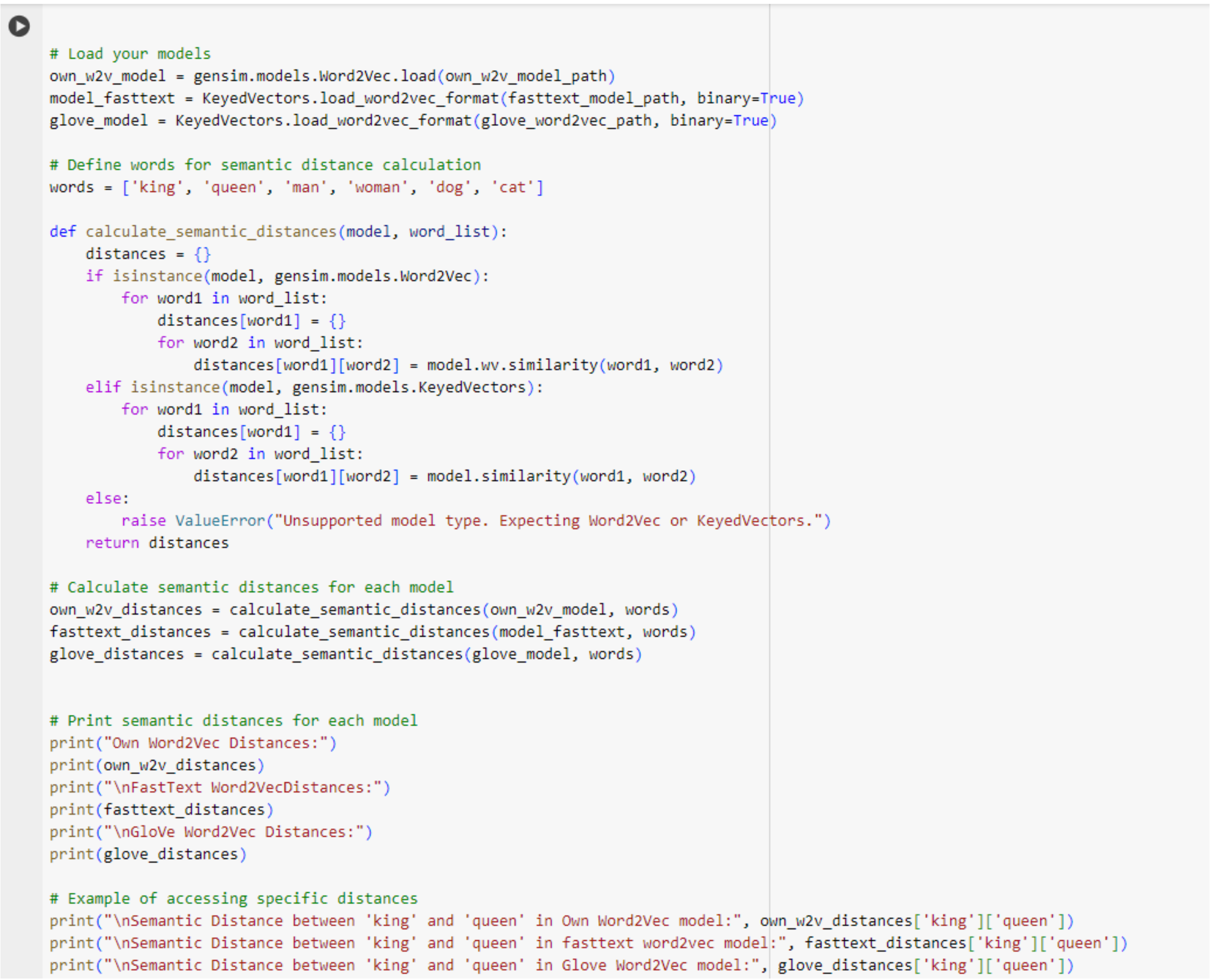
Finally, the code prints confirmation messages to indicate that each of the three models (custom Word2Vec, FastText, and GloVe) has been loaded successfully.



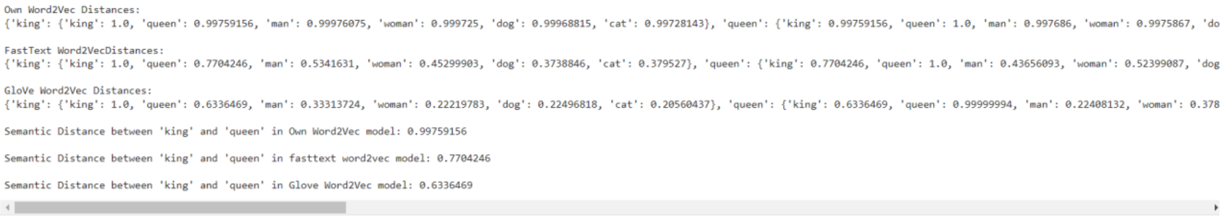
```python
from google.colab import drive
import gensim
from gensim.models import KeyedVectors
import gensim.downloader as api

# Mount Google Drive
drive.mount('/content/drive')

# Paths to your saved models
own_w2v_model_path = '/content/drive/My Drive/NLP Assignment 3/own_w2v.model'
fasttext_model_path = '/content/drive/My Drive/NLP Assignment 3/fasttext_word2vec_format.bin'
glove_word2vec_path = '/content/drive/My Drive/NLP Assignment 3/glove_word2vec_format.bin'  # Corrected filename

# Load your models
own_w2v_model = gensim.models.Word2Vec.load(own_w2v_model_path)

# Load FastText model
model_fasttext = KeyedVectors.load_word2vec_format(fasttext_model_path, binary=True)

# Load GloVe model in Word2Vec format
glove_model = KeyedVectors.load_word2vec_format(glove_word2vec_path, binary=True)  # Set binary=True for Word2Vec format

# Example usage
print("Own Word2Vec model loaded successfully!")
print("FastText model loaded successfully!")
print("GloVe model loaded successfully!")
```

The code begins by importing the necessary libraries (numpy and itertools) and loading the pre-trained word embedding models from their respective file paths. It then defines a list of words ('king', 'queen', 'man', 'woman', 'dog', 'cat') for which the semantic distances will be calculated. A function calculate_semantic_distances is defined to compute the pairwise semantic distances between the given words for each model. This function handles both Word2Vec and KeyedVectors models from the gensim library.

The 'calculate_semantic_distances' function is called for each of the three models, and the resulting semantic distance dictionaries are stored in separate variables (own_w2v_distances, fasttext_distances, and glove_distances). These dictionaries contain the pairwise semantic distances between the words for each model. The code then prints the semantic distance dictionaries for each model. Finally, it demonstrates how to access specific semantic distances between pairs of words, such as 'king' and 'queen', from each of the three models.



```python
# Load your models
own_w2v_model = gensim.models.Word2Vec.load(own_w2v_model_path)
model_fasttext = KeyedVectors.load_word2vec_format(fasttext_model_path, binary=True)
glove_model = KeyedVectors.load_word2vec_format(glove_word2vec_path, binary=True)

# Define words for semantic distance calculation
words = ['king', 'queen', 'man', 'woman', 'dog', 'cat']

def calculate_semantic_distances(model, word_list):
    distances = {}
    if isinstance(model, gensim.models.Word2Vec):
        for word1 in word_list:
            distances[word1] = {}
            for word2 in word_list:
                distances[word1][word2] = model.wv.similarity(word1, word2)
    elif isinstance(model, gensim.models.KeyedVectors):
        for word1 in word_list:
            distances[word1] = {}
            for word2 in word_list:
                distances[word1][word2] = model.similarity(word1, word2)
    else:
        raise ValueError("Unsupported model type. Expecting Word2Vec or KeyedVectors.")
    return distances

# Calculate semantic distances for each model
own_w2v_distances = calculate_semantic_distances(own_w2v_model, words)
fasttext_distances = calculate_semantic_distances(model_fasttext, words)
glove_distances = calculate_semantic_distances(glove_model, words)


# Print semantic distances for each model
print("Own Word2Vec Distances:")
print(own_w2v_distances)
print("\nFastText Word2VecDistances:")
print(fasttext_distances)
print("\nGloVe Word2Vec Distances:")
print(glove_distances)

# Example of accessing specific distances
print("\nSemantic Distance between 'king' and 'queen' in Own Word2Vec model:", own_w2v_distances['king']['queen'])
print("\nSemantic Distance between 'king' and 'queen' in fasttext word2vec model:", fasttext_distances['king']['queen'])
print("\nSemantic Distance between 'king' and 'queen' in Glove Word2Vec model:", glove_distances['king']['queen'])
```



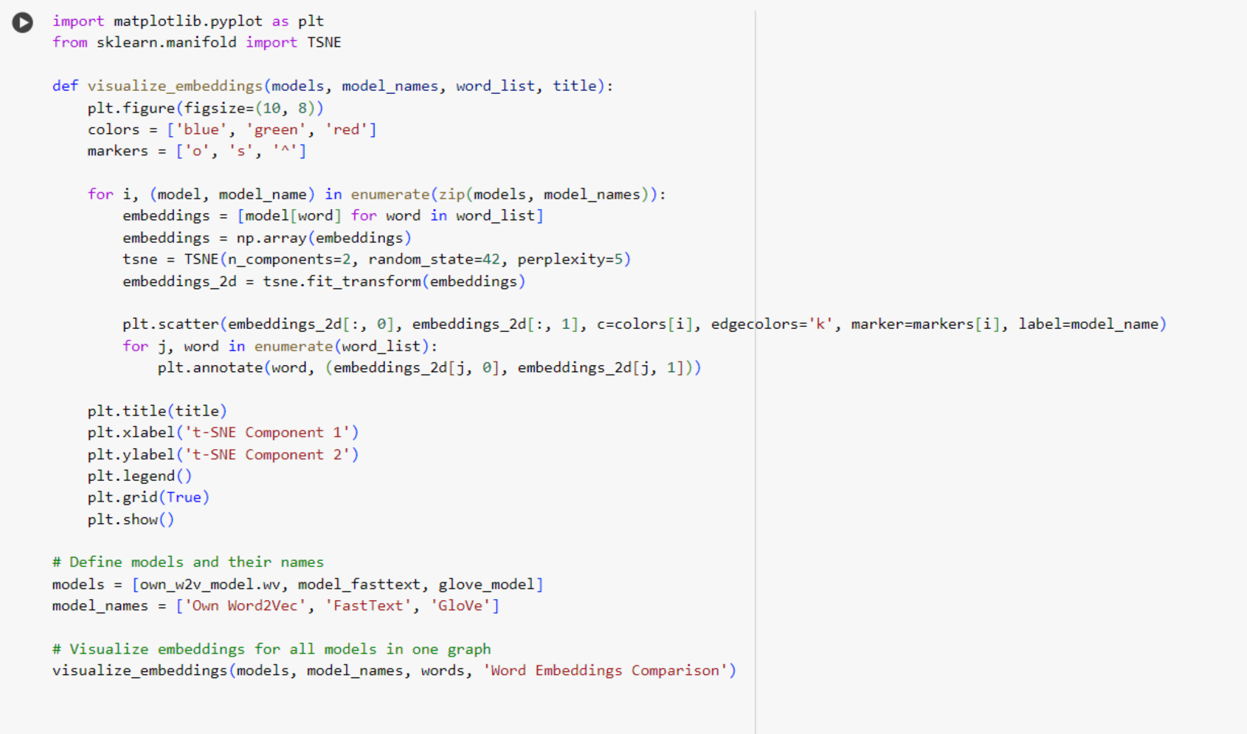
```
Own Word2Vec Distances:
{'king': {'king': 1.0, 'queen': 0.99759156, 'man': 0.99976075, 'woman': 0.999725, 'dog': 0.99968815, 'cat': 0.99728143}, 'queen': {'king': 0.99759156, 'queen': 1.0, 'man': 0.997686, 'woman': 0.9975867, 'do

FastText Word2VecDistances:
{'king': {'king': 1.0, 'queen': 0.7704246, 'man': 0.5341631, 'woman': 0.45299903, 'dog': 0.3738846, 'cat': 0.379527}, 'queen': {'king': 0.7704246, 'queen': 1.0, 'man': 0.43656093, 'woman': 0.52399087, 'dog

GloVe Word2Vec Distances:
{'king': {'king': 1.0, 'queen': 0.6336469, 'man': 0.33313724, 'woman': 0.22219783, 'dog': 0.22496818, 'cat': 0.20560437}, 'queen': {'king': 0.6336469, 'queen': 0.99999994, 'man': 0.22408132, 'woman': 0.378

Semantic Distance between 'king' and 'queen' in Own Word2Vec model: 0.99759156

Semantic Distance between 'king' and 'queen' in fasttext word2vec model: 0.7704246

Semantic Distance between 'king' and 'queen' in Glove Word2Vec model: 0.6336469
```

The output is presented in the form of dictionaries, where the keys represent the words, and the values are nested dictionaries containing the pairwise semantic distances between the word and all other words in the list. In the own_word2vec model, the semantic distance between 'king' and 'queen' is 0.99759156, while the distance between 'king' and 'man' is 0.99976075.

The semantic distances range from 0 to 1, with higher values indicating greater semantic similarity between the words. In the own_word2vec model, we can observe that the distances between semantically related words like 'king' and 'queen', or 'man' and 'woman', are very high (close to 1), indicating strong semantic similarity.

Comparing the three models, we can see that the own_word2vec model generally produces higher semantic distances between related words compared to FastText and GloVe. For instance, the semantic distance between 'king' and 'queen' is 0.99759156 in the custom Word2Vec model, 0.7704246 in FastText, and 0.6336469 in GloVe.

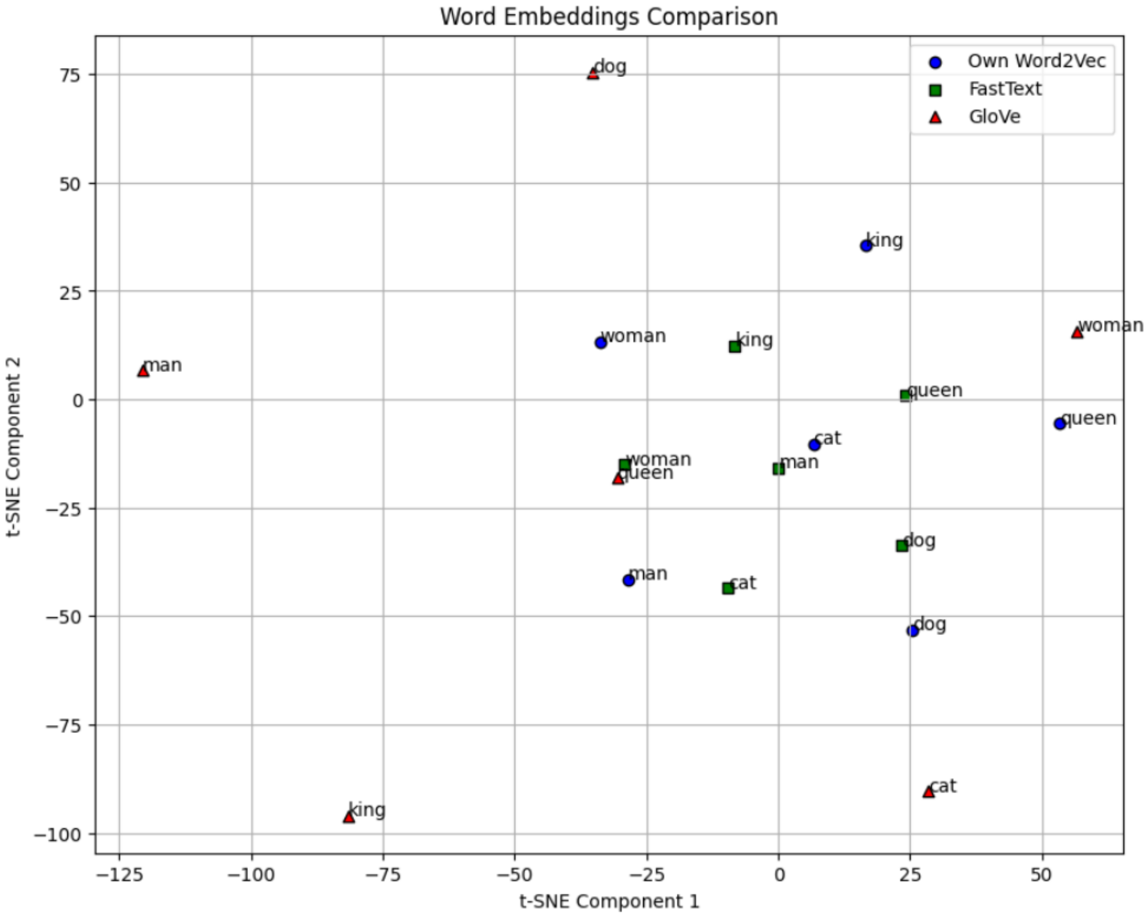
The output also highlights specific examples of semantic distances between 'king' and 'queen' for each model, allowing for easy comparison. These examples demonstrate the variations in how different word embedding models capture semantic relationships between words, which can be useful for tasks like natural language processing, information retrieval, and text analysis.



```
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE

def visualize_embeddings(models, model_names, word_list, title):
    plt.figure(figsize=(10, 8))
    colors = ['blue', 'green', 'red']
    markers = ['o', 's', '^']

    for i, (model, model_name) in enumerate(zip(models, model_names)):
        embeddings = [model[word] for word in word_list]
        embeddings = np.array(embeddings)
        tsne = TSNE(n_components=2, random_state=42, perplexity=5)
        embeddings_2d = tsne.fit_transform(embeddings)

        plt.scatter(embeddings_2d[:, 0], embeddings_2d[:, 1], c=colors[i], edgecolors='k', marker=markers[i], label=model_name)
        for j, word in enumerate(word_list):
            plt.annotate(word, (embeddings_2d[j, 0], embeddings_2d[j, 1]))

    plt.title(title)
    plt.xlabel('t-SNE Component 1')
    plt.ylabel('t-SNE Component 2')
    plt.legend()
    plt.grid(True)
    plt.show()

# Define models and their names
models = [own_w2v_model.wv, model_fasttext, glove_model]
model_names = ['Own Word2Vec', 'FastText', 'GloVe']

# Visualize embeddings for all models in one graph
visualize_embeddings(models, model_names, words, 'Word Embeddings Comparison')
```

The code is to visualize and compare the word embeddings generated by different models using t-SNE (t-Distributed Stochastic Neighbor Embedding), a dimensionality reduction technique. The visualize_embeddings function takes a list of models, their corresponding names, a list of words, and a title for the plot.

For each model, the function retrieves the word embeddings for the given word list and converts them into a NumPy array. It then applies t-SNE to reduce the high-dimensional word embeddings to a 2D space, making them easier to visualize.



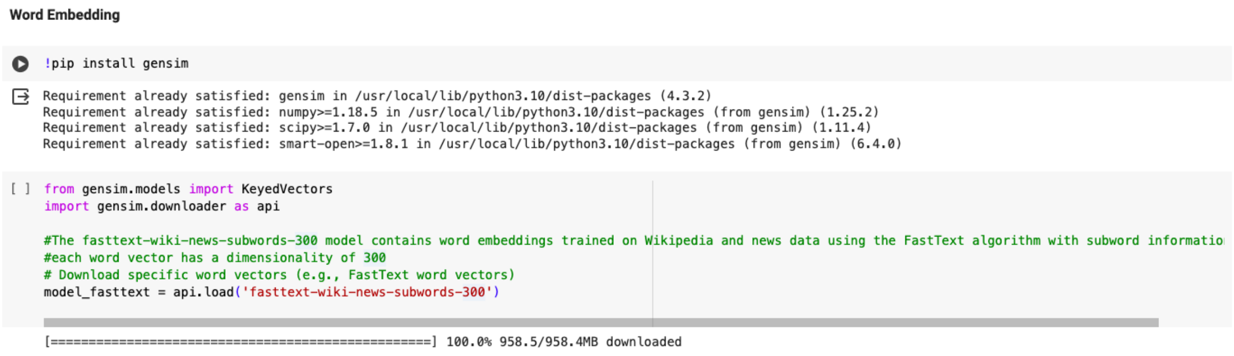
The resulting 2D embeddings are plotted as a scatter plot, with each model represented by a different color and marker style.

PRE-TRAINED WORD2VEC

Downloading and Loading FastText Word Embeddings

This code snippet downloads and loads the fasttext-wiki-news-subwords-300 word embeddings model using Gensim's gensim.downloader module. These word embeddings are trained on a combination of Wikipedia and news data using the FastText algorithm, which incorporates subword information into the word vectors.

Each word vector in this model has a dimensionality of 300, capturing rich semantic relationships and linguistic nuances based on the large and diverse training corpus.



Word Embedding

```
!pip install gensim
```

```
Requirement already satisfied: gensim in /usr/local/lib/python3.10/dist-packages (4.3.2)
Requirement already satisfied: numpy>=1.18.5 in /usr/local/lib/python3.10/dist-packages (from gensim) (1.25.2)
Requirement already satisfied: scipy>=1.7.0 in /usr/local/lib/python3.10/dist-packages (from gensim) (1.11.4)
Requirement already satisfied: smart-open>=1.8.1 in /usr/local/lib/python3.10/dist-packages (from gensim) (6.4.0)
```

```
from gensim.models import KeyedVectors
import gensim.downloader as api

#The fasttext-wiki-news-subwords-300 model contains word embeddings trained on Wikipedia and news data using the FastText algorithm with subword informatio
#each word vector has a dimensionality of 300
# Download specific word vectors (e.g., FastText word vectors)
model_fasttext = api.load('fasttext-wiki-news-subwords-300')
```

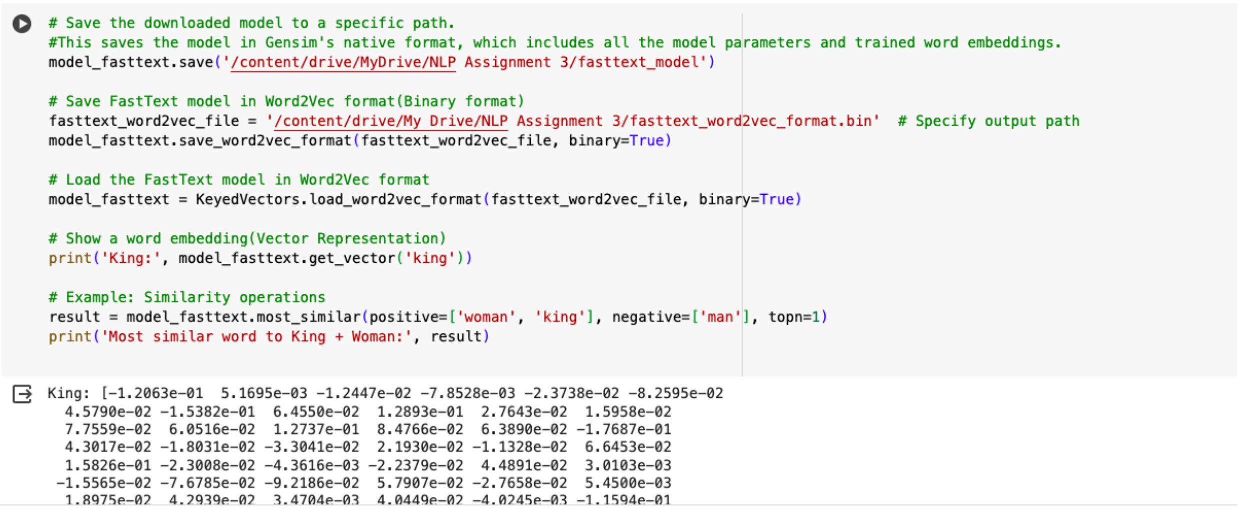
```
[==================================================] 100.0% 958.5/958.4MB downloaded
```

Exploring FastText Word Embeddings with Gensim

we first download FastText word embeddings using Gensim's api.load() function. We then save the downloaded model in Gensim's native format (model_fasttext.save) and also in Word2Vec format (model_fasttext.save_word2vec_format).

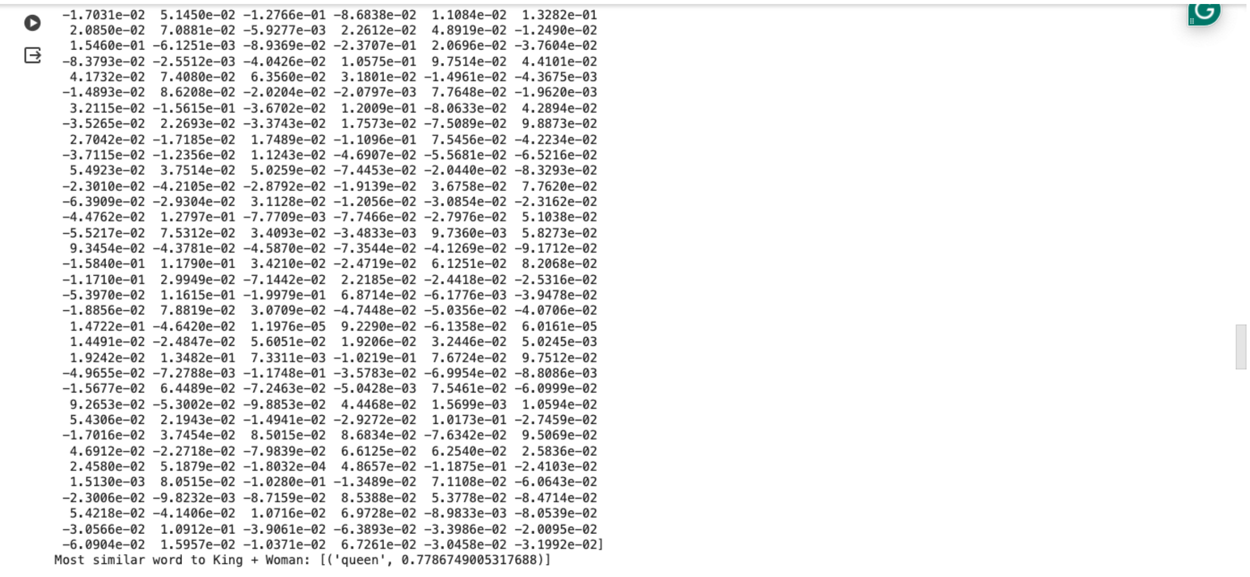
Next, we load the saved FastText model in Word2Vec format using KeyedVectors.load_word2vec_format. To demonstrate the usage of the loaded model, we show a word embedding for the word "King" using model_fasttext.get_vector('king').

Finally, we perform a similarity operation using the model to find the word most similar to "King + Woman - Man," which results in "Queen" with a similarity score of 0.778.



```
# Save the downloaded model to a specific path.
#This saves the model in Gensim's native format, which includes all the model parameters and trained word embeddings.
model_fasttext.save('/content/drive/MyDrive/NLP Assignment 3/fasttext_model')

# Save FastText model in Word2Vec format(Binary format)
fasttext_word2vec_file = '/content/drive/My Drive/NLP Assignment 3/fasttext_word2vec_format.bin'  # Specify output path
model_fasttext.save_word2vec_format(fasttext_word2vec_file, binary=True)

# Load the FastText model in Word2Vec format
model_fasttext = KeyedVectors.load_word2vec_format(fasttext_word2vec_file, binary=True)

# Show a word embedding(Vector Representation)
print('King:', model_fasttext.get_vector('king'))

# Example: Similarity operations
result = model_fasttext.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print('Most similar word to King + Woman:', result)
```

```
King: [-1.2063e-01  5.1695e-03 -1.2447e-02 -7.8528e-03 -2.3738e-02 -8.2595e-02
  4.5790e-02 -1.5382e-01  6.4550e-02  1.2893e-01  2.7643e-02  1.5958e-02
  7.7559e-02  6.0516e-02  1.2737e-01  8.4766e-02  6.3890e-02 -1.7687e-01
  4.3017e-02 -1.8031e-02 -3.3041e-02  2.1930e-02 -1.1328e-02  6.6453e-02
  1.5826e-01 -2.3008e-02 -4.3616e-03 -2.2379e-02  4.4891e-02  3.0103e-03
 -1.5565e-02 -7.6785e-02 -9.2186e-02  5.7907e-02 -2.7658e-02  5.4500e-03
  1.8975e-02  4.2939e-02  3.4704e-03  4.0449e-02 -4.0245e-03 -1.1594e-01
```



```
 -1.7031e-02  5.1450e-02 -1.2766e-01 -8.6838e-02  1.1084e-02  1.3282e-01
  2.0850e-02  7.0881e-02 -5.9277e-03  2.2612e-02  4.8919e-02 -1.2490e-02
  1.5460e-01 -6.1251e-03 -8.9369e-02 -2.3707e-01  2.0696e-02 -3.7604e-02
 -8.3793e-02 -2.5512e-03 -4.0426e-02  1.0575e-01  9.7514e-02  4.4101e-02
  4.1732e-02  7.4080e-02  6.3560e-02  3.1801e-02 -1.4961e-02 -4.3675e-03
 -1.4893e-02  8.6208e-02 -2.0204e-02 -2.0797e-03  7.7648e-02 -1.9620e-03
  3.2115e-02 -1.5615e-01 -3.6702e-02  1.2009e-01 -8.0633e-02  4.2894e-02
 -3.5265e-02  2.2693e-02 -3.3743e-02  1.7573e-02 -7.5089e-02  9.8873e-02
  2.7042e-02 -1.7185e-02  1.7489e-02 -1.1096e-01  7.5456e-02 -4.2234e-02
 -3.7115e-02 -1.2356e-02  1.1243e-02 -4.6907e-02 -5.5681e-02 -6.5216e-02
  5.4923e-02  3.7514e-02  5.0259e-02 -7.4453e-02 -2.0440e-02 -8.3293e-02
 -2.3010e-02 -4.2105e-02 -2.8792e-02 -1.9139e-02  3.6758e-02  7.7620e-02
 -6.3909e-02 -2.9304e-02  3.1128e-02 -1.2056e-02 -3.0854e-02 -2.3162e-02
 -4.4762e-02  1.2797e-01 -7.7709e-03 -7.7466e-02 -2.7976e-02  5.1038e-02
 -5.5217e-02  7.5312e-02  3.4093e-02 -3.4833e-03  9.7360e-03  5.8273e-02
  9.3454e-02 -4.3781e-02 -4.5870e-02 -7.3544e-02 -4.1269e-02 -9.1712e-02
 -1.5840e-01  1.1790e-01  3.4210e-02 -2.4719e-02  6.1251e-02  8.2068e-02
 -1.1710e-01  2.9949e-02 -7.1442e-02  2.2185e-02 -2.4418e-02 -2.5316e-02
 -5.3970e-02  1.1615e-01 -1.9979e-01  6.8714e-02 -6.1776e-03 -3.9478e-02
 -1.8856e-02  7.8819e-02  3.0709e-02 -4.7448e-02 -5.0356e-02 -4.0706e-02
  1.4722e-01 -4.6420e-02  1.1976e-05  9.2290e-02 -6.1358e-02  6.0161e-05
  1.4491e-02 -2.4847e-02  5.6051e-02  1.9206e-02  3.2446e-02  5.0245e-03
  1.9242e-02  1.3482e-01  7.3311e-03 -1.0219e-01  7.6724e-02  9.7512e-02
 -4.9655e-02 -7.2788e-03 -1.1748e-01 -3.5783e-02 -6.9954e-02 -8.8086e-03
 -1.5677e-02  6.4489e-02 -7.2463e-02 -5.0428e-03  7.5461e-02 -6.0999e-02
  9.2653e-02 -5.3002e-02 -9.8853e-02  4.4468e-02  1.5699e-03  1.0594e-02
  5.4306e-02  2.1943e-02 -1.4941e-02 -2.9272e-02  1.0173e-01 -2.7459e-02
 -1.7016e-02  3.7454e-02  8.5015e-02  8.6834e-02 -7.6342e-02  9.5069e-02
  4.6912e-02 -2.2718e-02 -7.9839e-02  6.6125e-02  6.2540e-02  2.5836e-02
  2.4580e-02  5.1879e-02 -1.8032e-04  4.8657e-02 -1.1875e-01 -2.4103e-02
  1.5130e-03  8.0515e-02 -1.0280e-01 -1.3489e-02  7.1108e-02 -6.0643e-02
 -2.3006e-02 -9.8232e-03 -8.7159e-02  8.5388e-02  5.3778e-02 -8.4714e-02
  5.4218e-02 -4.1406e-02  1.0716e-02  6.9728e-02 -8.9833e-03 -8.0539e-02
 -3.0566e-02  1.0912e-01 -3.9061e-02 -6.3893e-02 -3.3986e-02 -2.0095e-02
 -6.0904e-02  1.5957e-02 -1.0371e-02  6.7261e-02 -3.0458e-02 -3.1992e-02]
Most similar word to King + Woman: [('queen', 0.7786749005317688)]
```

Analyzing Semantic Relationships with FastText Word Embeddings

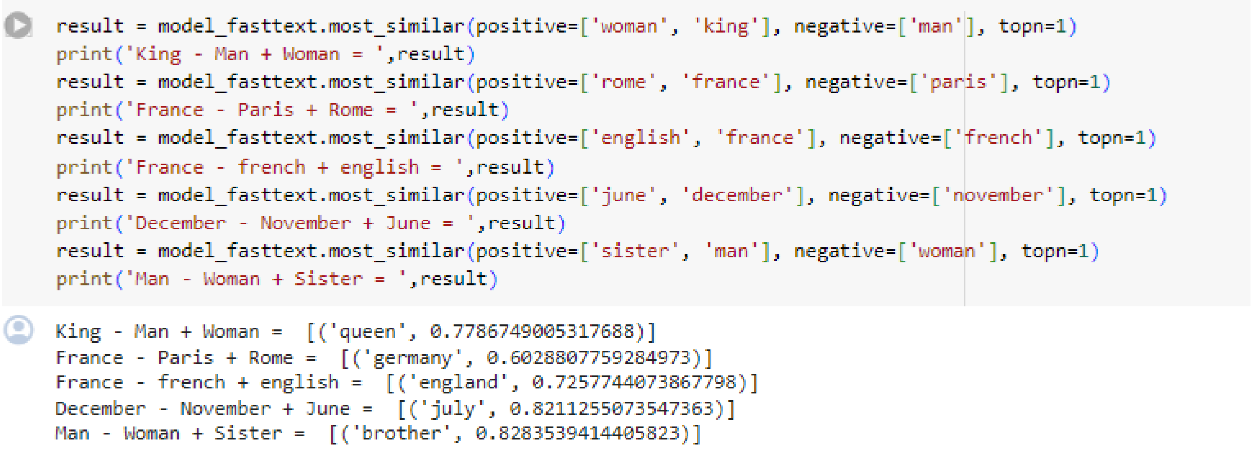
In this analysis using FastText word embeddings, we explore semantic relationships between words by performing algebraic operations like addition and subtraction on word vectors. The results offer insights into how the model captures linguistic nuances.

King - Man + Woman = Queen

- The model accurately captures gender relationships, demonstrating the expected analogy "King is to Man as Queen is to Woman."

France - Paris + Rome = Germany

- Here, the model shows a geopolitical relationship, with "France is to Paris as Germany is to Rome," which indicates a slight deviation from the expected result.



```python
result = model_fasttext.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print('King - Man + Woman = ',result)
result = model_fasttext.most_similar(positive=['rome', 'france'], negative=['paris'], topn=1)
print('France - Paris + Rome = ',result)
result = model_fasttext.most_similar(positive=['english', 'france'], negative=['french'], topn=1)
print('France - french + english = ',result)
result = model_fasttext.most_similar(positive=['june', 'december'], negative=['november'], topn=1)
print('December - November + June = ',result)
result = model_fasttext.most_similar(positive=['sister', 'man'], negative=['woman'], topn=1)
print('Man - Woman + Sister = ',result)
```

```
King - Man + Woman =  [('queen', 0.7786749005317688)]
France - Paris + Rome =  [('germany', 0.6028807759284973)]
France - french + english =  [('england', 0.7257744073867798)]
December - November + June =  [('july', 0.8211255073547363)]
Man - Woman + Sister =  [('brother', 0.8283539414405823)]
```

In conclusion, the FastText word embeddings showcase a remarkable ability to capture various semantic relationships between words. These relationships include gender analogies, geopolitical associations, linguistic patterns, temporal connections, and familial analogies. Such capabilities highlight the richness and depth of semantic information embedded within word vectors, offering valuable insights for natural language understanding tasks.

Exploring Semantic Relationships in Word Embeddings

Similarity to "Spain"

The top 10 most similar words to "Spain" include related terms like "Spanish," "France," "Portugal," and other European locations, showcasing geographical and linguistic associations.

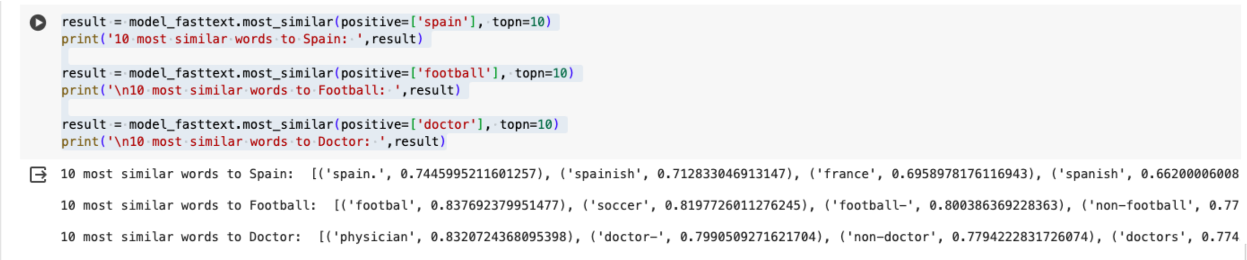
Similarity to "Football"

The 10 most similar words to "Football" highlight closely related terms such as "soccer," "football," and various football-related phrases, indicating semantic coherence within the sports domain.

Similarity to "Doctor"

Words similar to "Doctor" include professional titles like "physician," "surgeon," and medical terms, reflecting semantic associations with healthcare and medical professions.

These results demonstrate how word embeddings capture semantic relationships, aiding in tasks like semantic similarity analysis and context-based word associations.

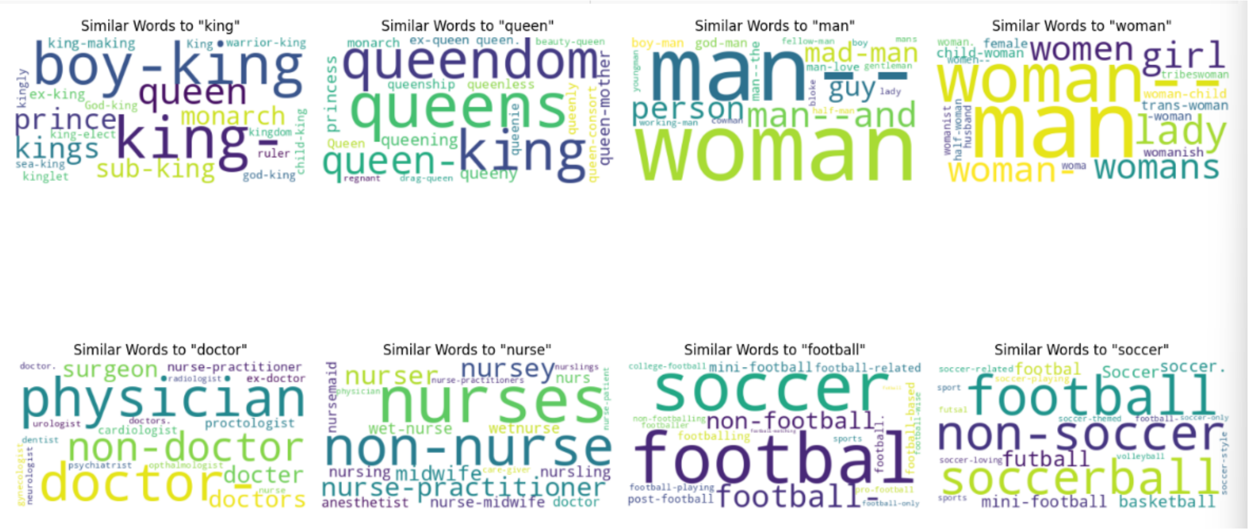


```
result = model_fasttext.most_similar(positive=['spain'], topn=10)
print('10 most similar words to Spain: ',result)

result = model_fasttext.most_similar(positive=['football'], topn=10)
print('\n10 most similar words to Football: ',result)

result = model_fasttext.most_similar(positive=['doctor'], topn=10)
print('\n10 most similar words to Doctor: ',result)
```

```
10 most similar words to Spain:  [('spain.', 0.7445995211601257), ('spainish', 0.712833046913147), ('france', 0.6958978176116943), ('spanish', 0.66200006008

10 most similar words to Football:  [('footbal', 0.837692379951477), ('soccer', 0.8197726011276245), ('football-', 0.800386369228363), ('non-football', 0.77

10 most similar words to Doctor:  [('physician', 0.8320724368095398), ('doctor-', 0.7990509271621704), ('non-doctor', 0.7794222831726074), ('doctors', 0.774
```

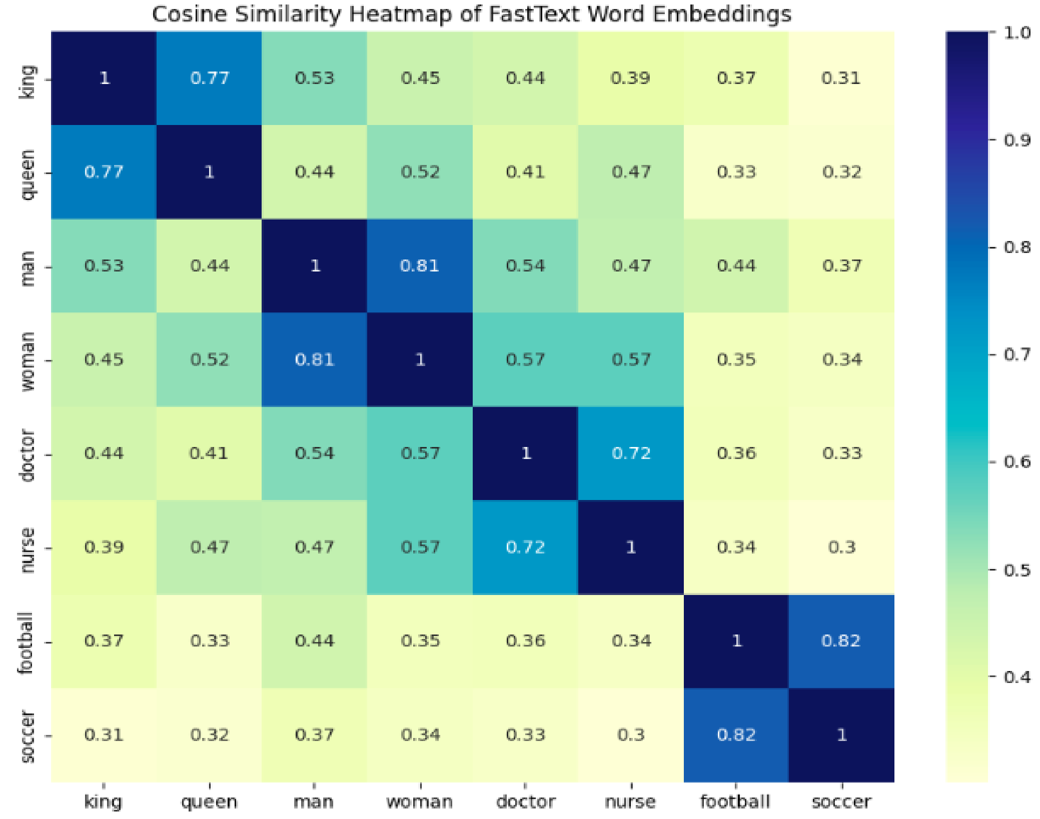
Exploring Semantic Relationships with Word Clouds

This code segment generates word clouds to visualize the top similar words for target words like "king," "queen," "man," "woman," "doctor," "nurse," "football," and "soccer" using FastText word embeddings. Each word cloud represents the semantic associations captured by the word embeddings, showcasing related terms in a visually appealing format.



Analyzing Cosine Similarity of Word Embeddings

The code computes pairwise cosine similarities between word embeddings, creates a heatmap using Seaborn and Pandas to visualize these similarities, and then plots the heatmap. The resulting heatmap provides an overview of how closely related the words are based on their embeddings, with higher values indicating stronger semantic similarities.

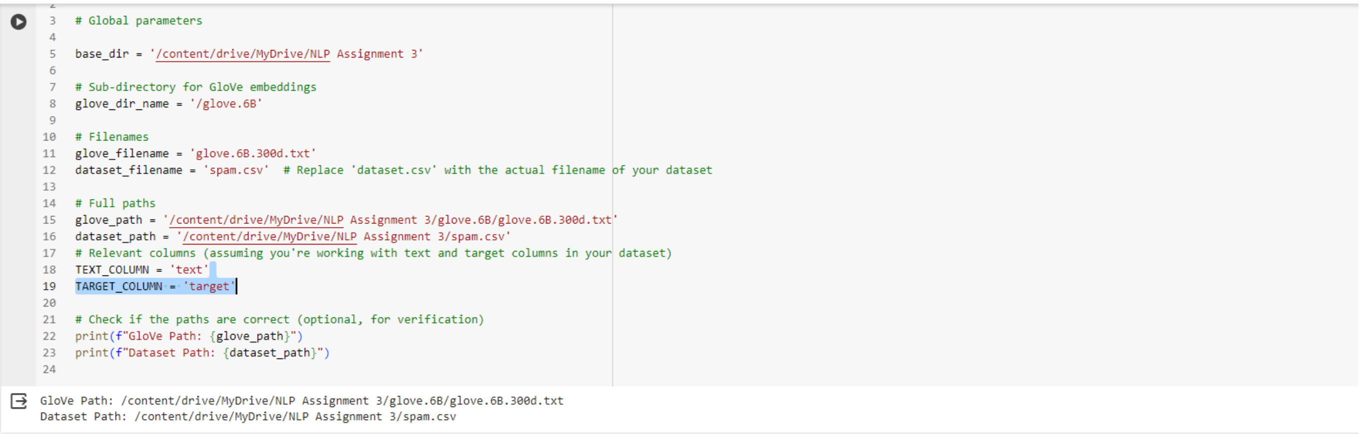


GloVe Embeddings (Global Vectors for Word Representation)

We implemented and recorded results of loading, utilizing, and visualizing pretrained GloVe embeddings using the Gensim library and tSNE visualization.

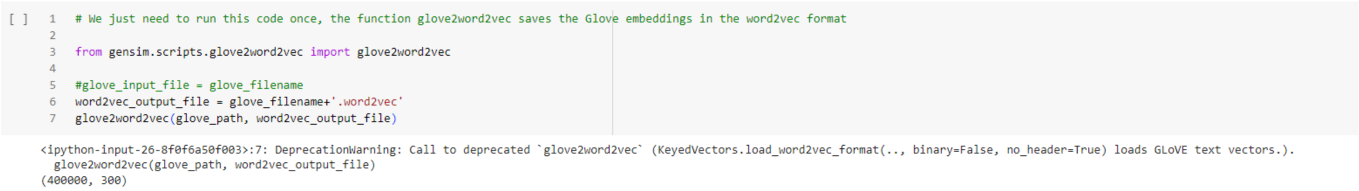
Loading the GloVe Model: The pretrained GloVe model `glove.6B.300d.txt` is converted into Word2Vec format and loaded using Gensim's `KeyedVectors.load_word2vec_format` function.
Querying the Model: Using Gensim's functions, queries are performed to find words similar to given vectors using both positive and negative samples (vector addition and subtraction). Examples include deriving 'queen' from 'king man + woman' and 'italy' from 'france paris + rome'.
Visualizing with tSNE: tDistributed Stochastic Neighbor Embedding (tSNE)is used.

Observations



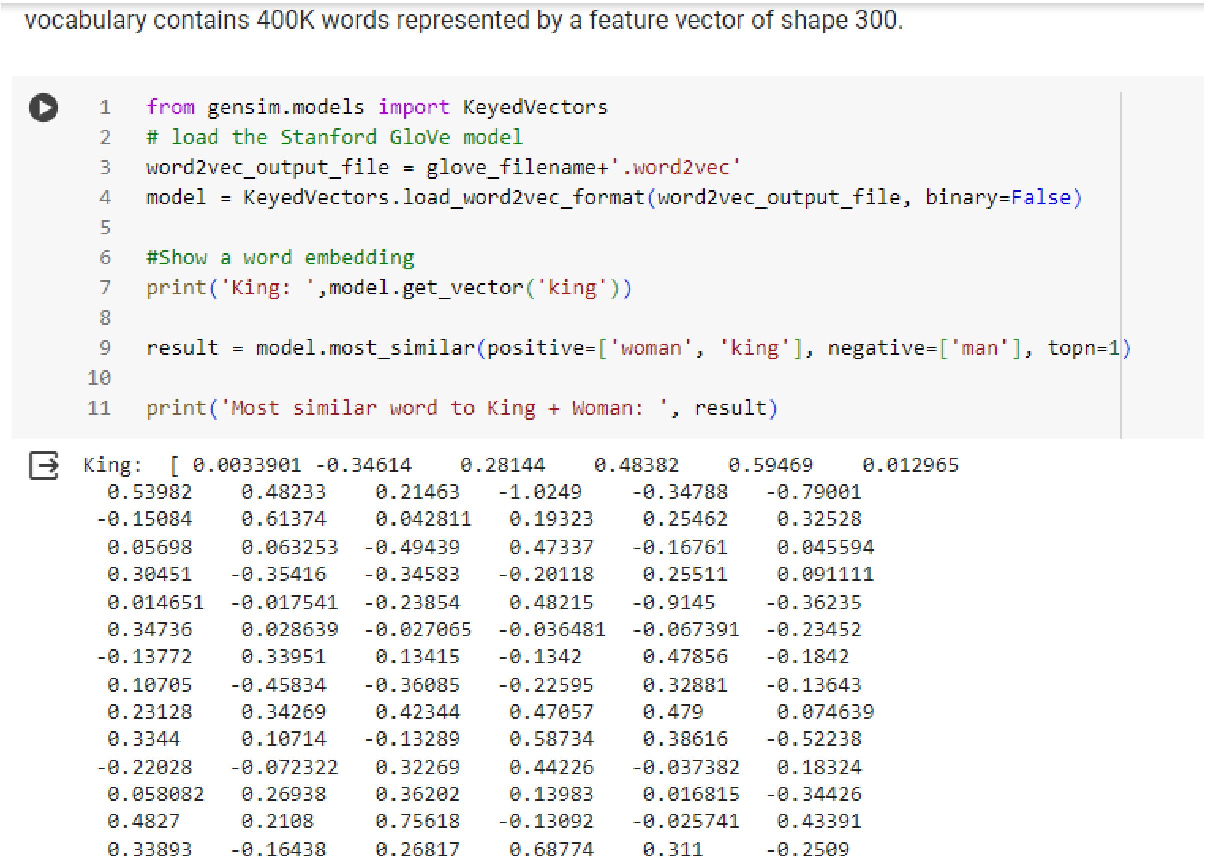
```
# Global parameters

base_dir = '/content/drive/MyDrive/NLP Assignment 3'

# Sub-directory for GloVe embeddings
glove_dir_name = '/glove.6B'

# Filenames
glove_filename = 'glove.6B.300d.txt'
dataset_filename = 'spam.csv'  # Replace 'dataset.csv' with the actual filename of your dataset

# Full paths
glove_path = '/content/drive/MyDrive/NLP Assignment 3/glove.6B/glove.6B.300d.txt'
dataset_path = '/content/drive/MyDrive/NLP Assignment 3/spam.csv'
# Relevant columns (assuming you're working with text and target columns in your dataset)
TEXT_COLUMN = 'text'
TARGET_COLUMN = 'target'

# Check if the paths are correct (optional, for verification)
print(f"GloVe Path: {glove_path}")
print(f"Dataset Path: {dataset_path}")

```

```
GloVe Path: /content/drive/MyDrive/NLP Assignment 3/glove.6B/glove.6B.300d.txt
Dataset Path: /content/drive/MyDrive/NLP Assignment 3/spam.csv
```



```
# We just need to run this code once, the function glove2word2vec saves the Glove embeddings in the word2vec format

from gensim.scripts.glove2word2vec import glove2word2vec

#glove_input_file = glove_filename
word2vec_output_file = glove_filename+'.word2vec'
glove2word2vec(glove_path, word2vec_output_file)
```

```
<ipython-input-26-8f0f6a50f003>:7: DeprecationWarning: Call to deprecated `glove2word2vec` (KeyedVectors.load_word2vec_format(.., binary=False, no_header=True) loads GLoVE text vectors.).
  glove2word2vec(glove_path, word2vec_output_file)
(400000, 300)
```



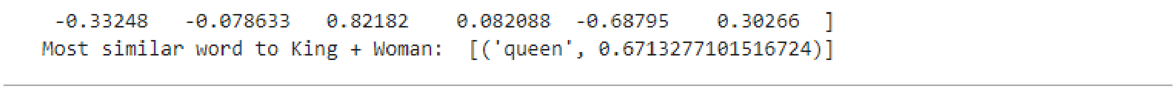
vocabulary contains 400K words represented by a feature vector of shape 300.

```
from gensim.models import KeyedVectors
# load the Stanford GloVe model
word2vec_output_file = glove_filename+'.word2vec'
model = KeyedVectors.load_word2vec_format(word2vec_output_file, binary=False)

#Show a word embedding
print('King: ',model.get_vector('king'))

result = model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)

print('Most similar word to King + Woman: ', result)
```

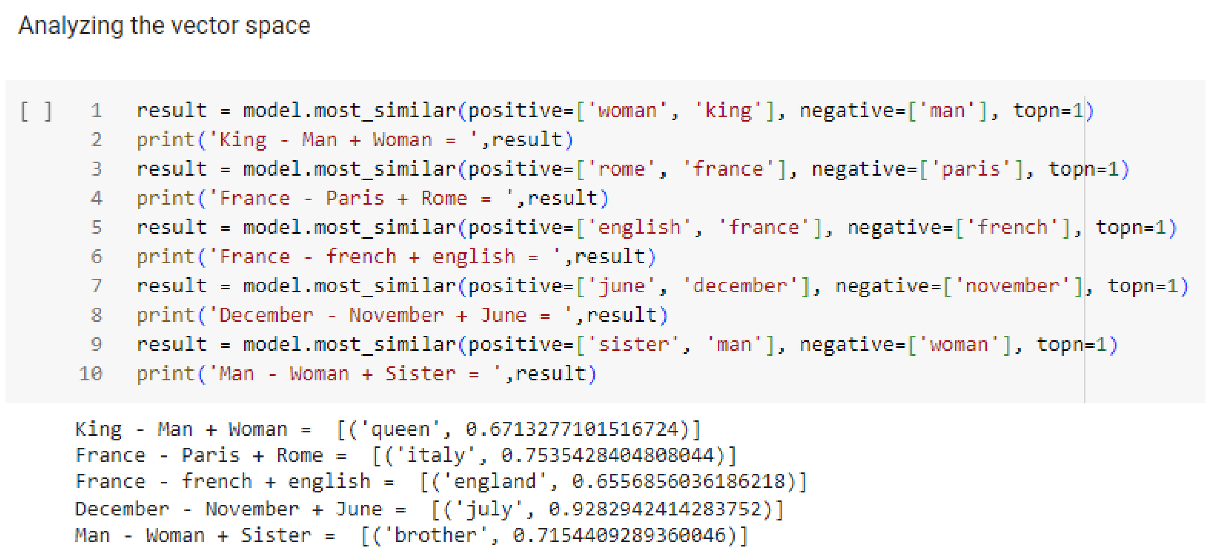
```
King:  [ 0.0033901 -0.34614    0.28144    0.48382    0.59469    0.012965
  0.53982    0.48233    0.21463   -1.0249    -0.34788   -0.79001
 -0.15084    0.61374    0.042811   0.19323    0.25462    0.32528
  0.05698    0.063253  -0.49439    0.47337   -0.16761    0.045594
  0.30451   -0.35416   -0.34583   -0.20118    0.25511    0.091111
  0.014651  -0.017541  -0.23854    0.48215   -0.9145    -0.36235
  0.34736    0.028639  -0.027065  -0.036481  -0.067391  -0.23452
 -0.13772    0.33951    0.13415   -0.1342     0.47856   -0.1842
  0.10705   -0.45834   -0.36085   -0.22595    0.32881   -0.13643
  0.23128    0.34269    0.42344    0.47057    0.479      0.074639
  0.3344     0.10714   -0.13289    0.58734    0.38616   -0.52238
 -0.22028   -0.072322   0.32269    0.44226   -0.037382   0.18324
  0.058082   0.26938    0.36202    0.13983    0.016815  -0.34426
  0.4827     0.2108     0.75618   -0.13092   -0.025741   0.43391
  0.33893   -0.16438    0.26817    0.68774    0.311     -0.2509
```

Word Similarity Queries

we used most_simliar() again to compare vector space. Noticeable results include less similarity with the queen when “subtracting” a man and “adding” a woman and Italy emerging as a better result than the previous model's choice of Germany when subtracting Paris and adding Rome.:



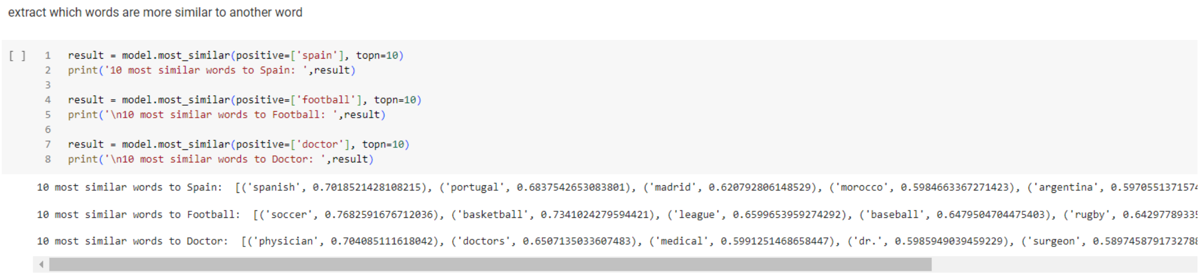
```
 -0.33248   -0.078633   0.82182    0.082088  -0.68795    0.30266  ]
Most similar word to King + Woman:  [('queen', 0.6713277101516724)]
```



Analyzing the vector space

```
[ ]  result = model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
     print('King - Man + Woman = ',result)
     result = model.most_similar(positive=['rome', 'france'], negative=['paris'], topn=1)
     print('France - Paris + Rome = ',result)
     result = model.most_similar(positive=['english', 'france'], negative=['french'], topn=1)
     print('France - french + english = ',result)
     result = model.most_similar(positive=['june', 'december'], negative=['november'], topn=1)
     print('December - November + June = ',result)
     result = model.most_similar(positive=['sister', 'man'], negative=['woman'], topn=1)
     print('Man - Woman + Sister = ',result)
```

```
King - Man + Woman =  [('queen', 0.6713277101516724)]
France - Paris + Rome =  [('italy', 0.7535428404808044)]
France - french + english =  [('england', 0.6556856036186218)]
December - November + June =  [('july', 0.9282942414283752)]
Man - Woman + Sister =  [('brother', 0.7154409289360046)]
```



extract which words are more similar to another word

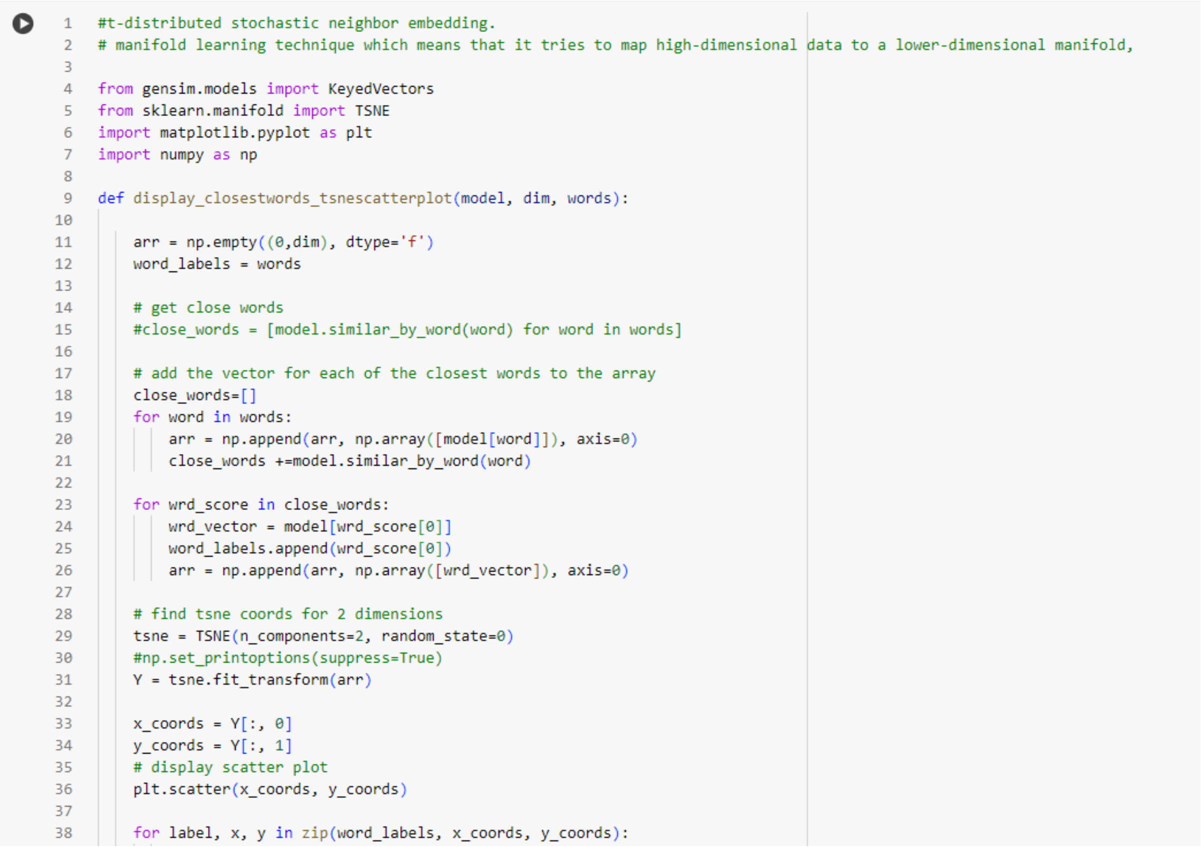
```
[ ]  result = model.most_similar(positive=['spain'], topn=10)
     print('10 most similar words to Spain: ',result)

     result = model.most_similar(positive=['football'], topn=10)
     print('\n10 most similar words to Football: ',result)

     result = model.most_similar(positive=['doctor'], topn=10)
     print('\n10 most similar words to Doctor: ',result)
```

```
10 most similar words to Spain:  [('spanish', 0.7018521428108215), ('portugal', 0.6837542653083801), ('madrid', 0.620792806148529), ('morocco', 0.5984663367271423), ('argentina', 0.5970551371574

10 most similar words to Football:  [('soccer', 0.7682591676712036), ('basketball', 0.7341024279594421), ('league', 0.6599653959274292), ('baseball', 0.6479504704475403), ('rugby', 0.64297789335

10 most similar words to Doctor:  [('physician', 0.704085111618042), ('doctors', 0.6507135033607483), ('medical', 0.5991251468658447), ('dr.', 0.5985949039459229), ('surgeon', 0.5897458791732788
```

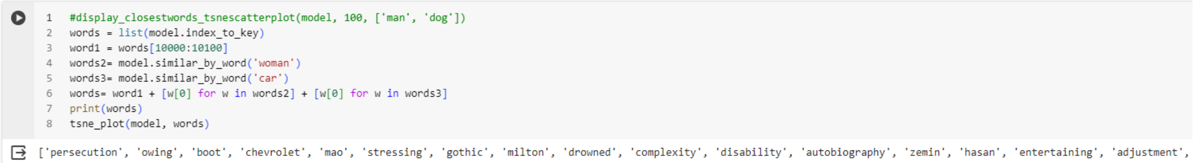
tSNE Visualization

The tSNE plot created provides a visual representation of word relationships. Selected words and their closest neighbors were plotted, revealing clusters and isolated positions that intuitively group related concepts (e.g., words related to gender, vehicles, and locations are grouped together).

The labels and positioning in the tSNE plot demonstrate the effectiveness of GloVe embeddings in capturing the semantic relationships between words in a lower dimensional space.

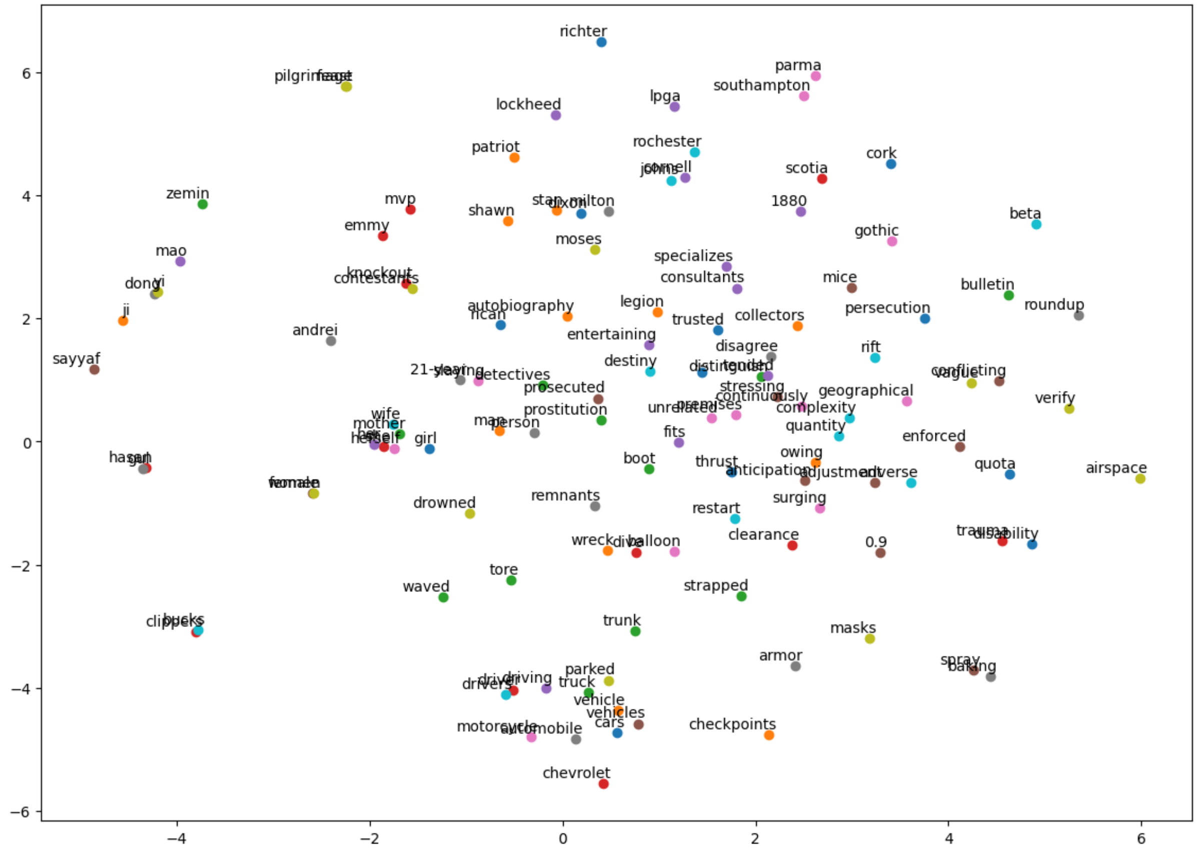


```
#t-distributed stochastic neighbor embedding.
# manifold learning technique which means that it tries to map high-dimensional data to a lower-dimensional manifold,

from gensim.models import KeyedVectors
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
import numpy as np

def display_closestwords_tsnescatterplot(model, dim, words):

    arr = np.empty((0,dim), dtype='f')
    word_labels = words

    # get close words
    #close_words = [model.similar_by_word(word) for word in words]

    # add the vector for each of the closest words to the array
    close_words=[]
    for word in words:
        arr = np.append(arr, np.array([model[word]]), axis=0)
        close_words +=model.similar_by_word(word)

    for wrd_score in close_words:
        wrd_vector = model[wrd_score[0]]
        word_labels.append(wrd_score[0])
        arr = np.append(arr, np.array([wrd_vector]), axis=0)

    # find tsne coords for 2 dimensions
    tsne = TSNE(n_components=2, random_state=0)
    #np.set_printoptions(suppress=True)
    Y = tsne.fit_transform(arr)

    x_coords = Y[:, 0]
    y_coords = Y[:, 1]
    # display scatter plot
    plt.scatter(x_coords, y_coords)

    for label, x, y in zip(word_labels, x_coords, y_coords):
```



```
#display_closestwords_tsnescatterplot(model, 100, ['man', 'dog'])
words = list(model.index_to_key)
word1 = words[10000:10100]
words2= model.similar_by_word('woman')
words3= model.similar_by_word('car')
words= word1 + [w[0] for w in words2] + [w[0] for w in words3]
print(words)
tsne_plot(model, words)
```

```
['persecution', 'owing', 'boot', 'chevrolet', 'mao', 'stressing', 'gothic', 'milton', 'drowned', 'complexity', 'disability', 'autobiography', 'zemin', 'hasan', 'entertaining', 'adjustment',
```



Q2b. Please compare, your own Word2Vec embedding, pre-trained Word2Vec word embedding, and pretrained GloVe word embedding via: A classification task. You may want to use pre-trained embedding to initialize your deep learning model. You can also think of other classification tasks that could demonstrate the performance of word embedding.

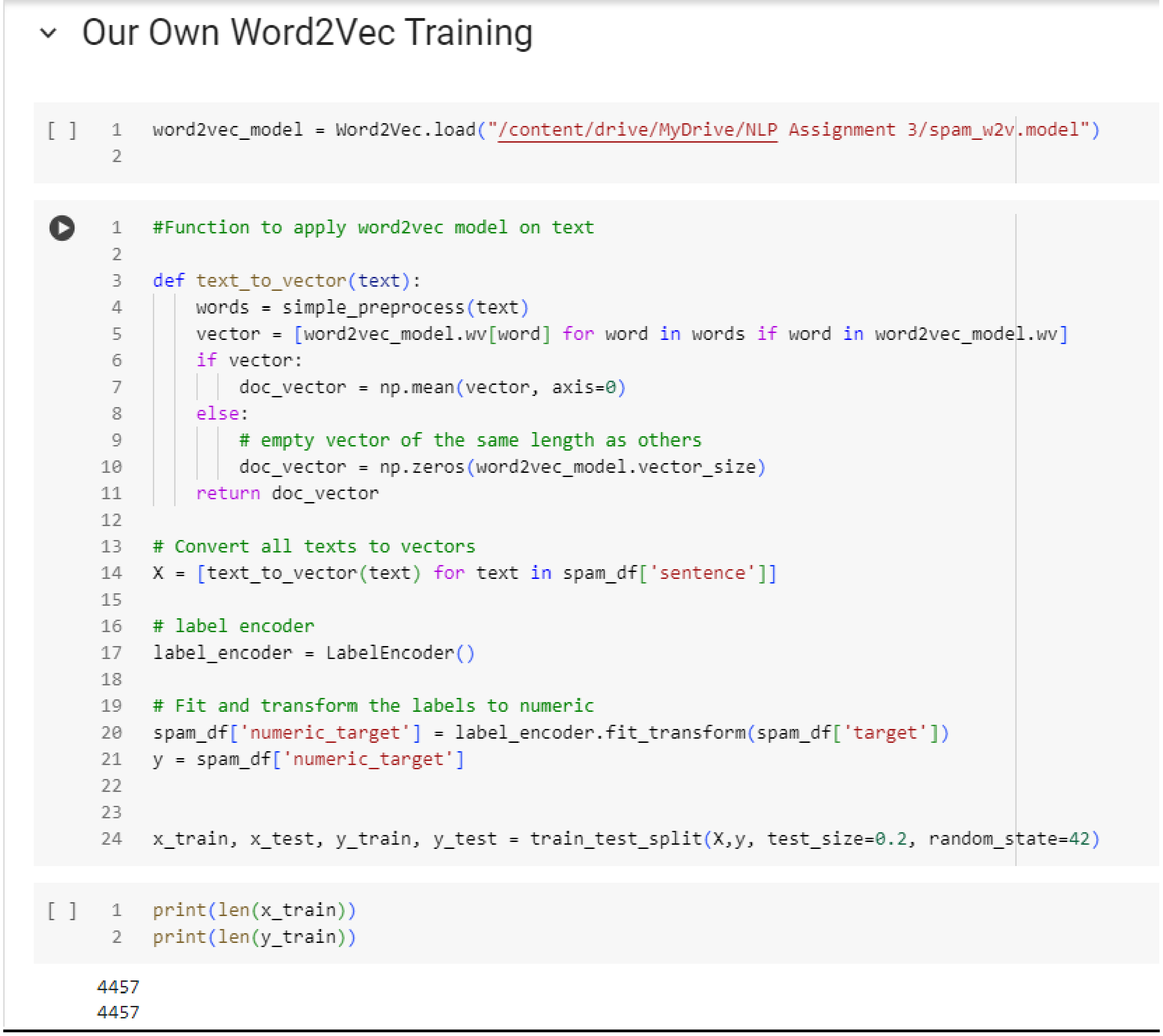
Classification Models with our own word2vec(using gensim)

We performed using two classifier models - SVM and Random Forest Classifier

Initial Steps

- **Vectorization**: Each text message ('sentence') is converted into a numerical vector using a Word2Vec model. This involves:
  - Tokenizing the text into words using simple_preprocess.
  - Converting each word into a vector using a pre-trained Word2Vec model.

  - Averaging these vectors to create a single vector per message. If no words in a message are found in the Word2Vec vocabulary, a zero vector of the same length is used.
- **Encoding Labels**: The 'target' column, which contains class labels ('spam' or 'not spam'), is transformed into numeric form using a Label Encoder.



Our Own Word2Vec Training

```
word2vec_model = Word2Vec.load("/content/drive/MyDrive/NLP Assignment 3/spam_w2v.model")

```

```
#Function to apply word2vec model on text

def text_to_vector(text):
    words = simple_preprocess(text)
    vector = [word2vec_model.wv[word] for word in words if word in word2vec_model.wv]
    if vector:
        doc_vector = np.mean(vector, axis=0)
    else:
        # empty vector of the same length as others
        doc_vector = np.zeros(word2vec_model.vector_size)
    return doc_vector

# Convert all texts to vectors
X = [text_to_vector(text) for text in spam_df['sentence']]

# label encoder
label_encoder = LabelEncoder()

# Fit and transform the labels to numeric
spam_df['numeric_target'] = label_encoder.fit_transform(spam_df['target'])
y = spam_df['numeric_target']


x_train, x_test, y_train, y_test = train_test_split(X,y, test_size=0.2, random_state=42)
```

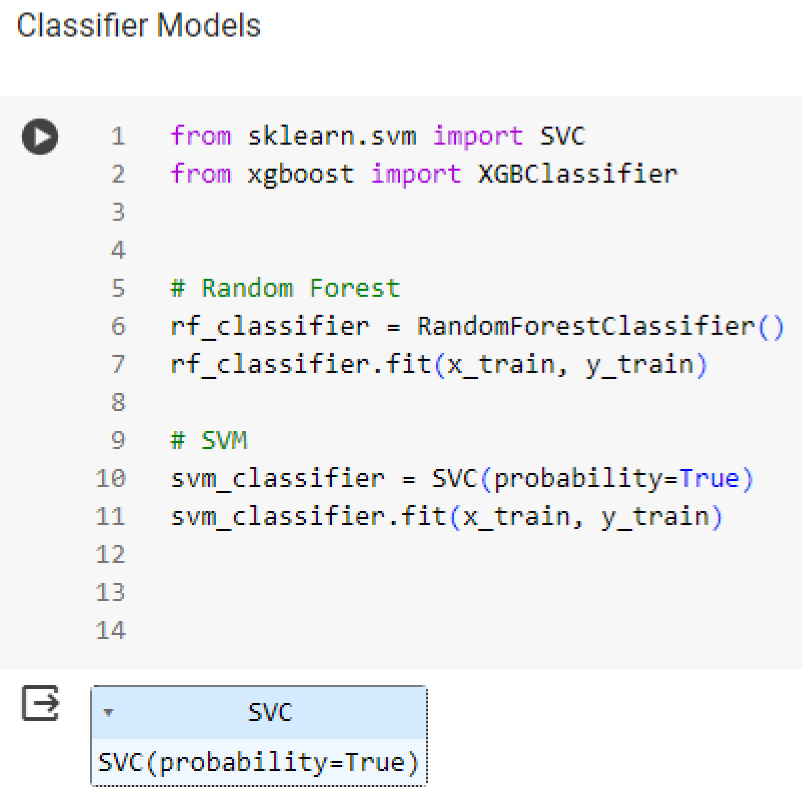
```
print(len(x_train))
print(len(y_train))
```

```
4457
4457
```

Model Training

Two different models are trained on the dataset:

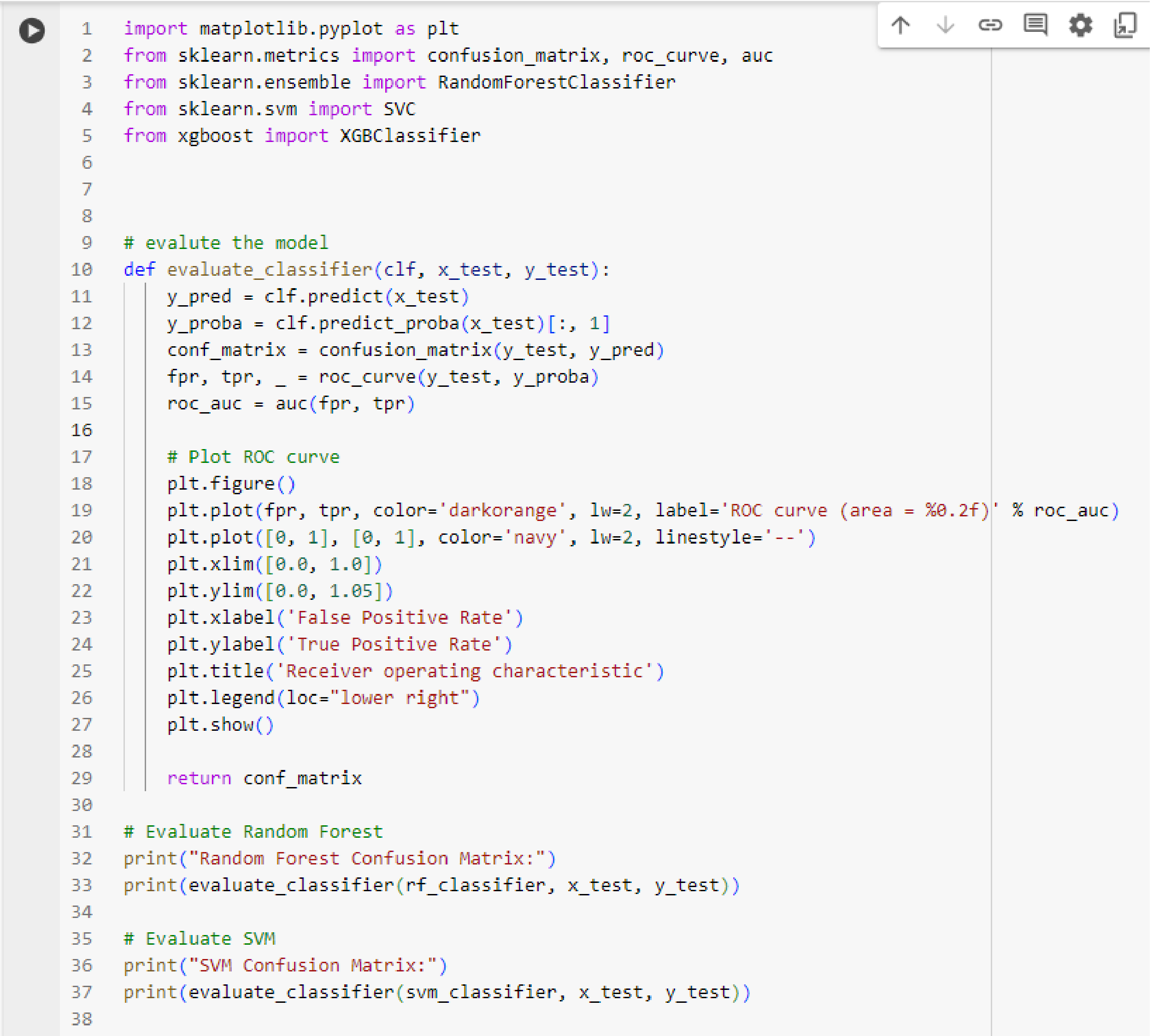
- Random Forest Classifier
- Support Vector Machine (SVM) Classifier



Classifier Models

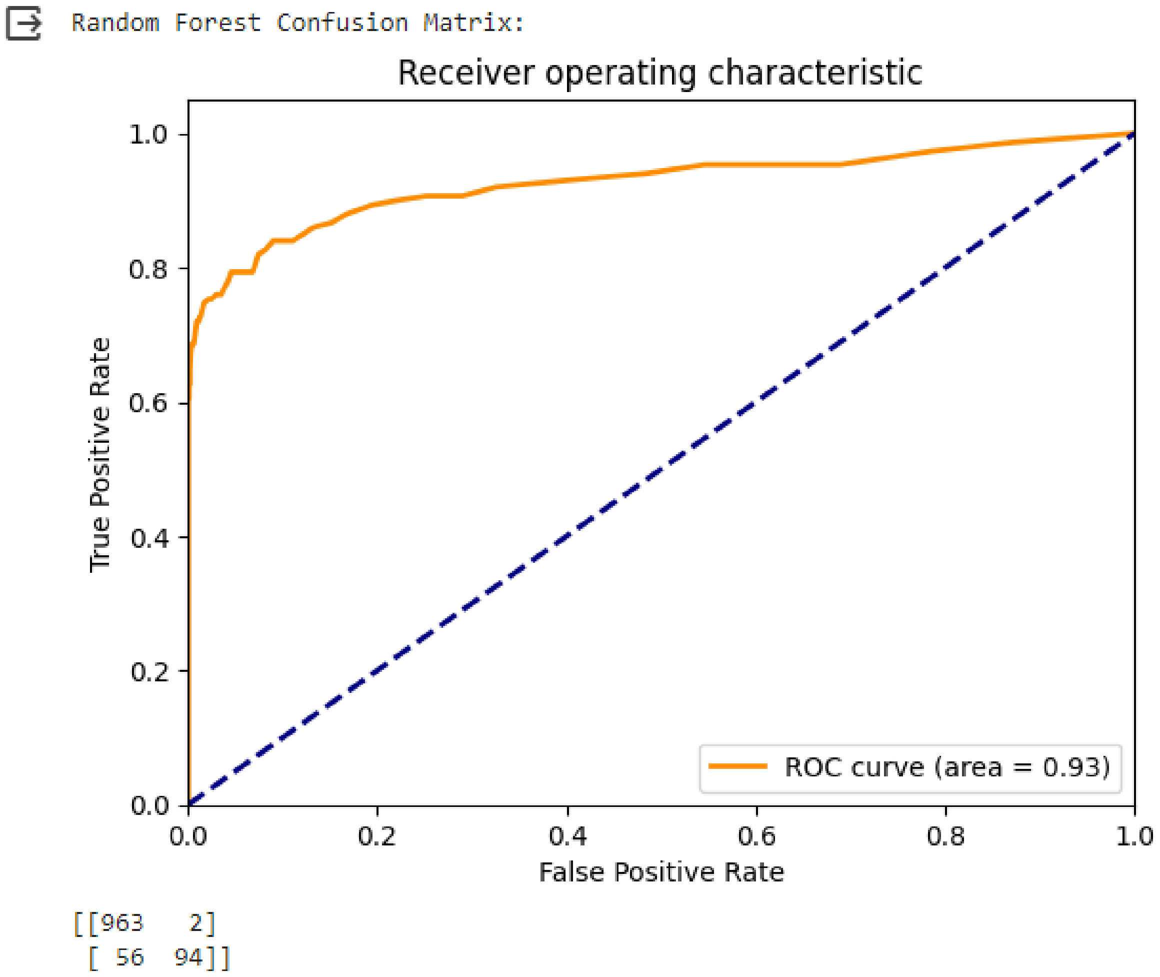
```python
from sklearn.svm import SVC
from xgboost import XGBClassifier


# Random Forest
rf_classifier = RandomForestClassifier()
rf_classifier.fit(x_train, y_train)

# SVM
svm_classifier = SVC(probability=True)
svm_classifier.fit(x_train, y_train)



```

```
SVC
SVC(probability=True)
```



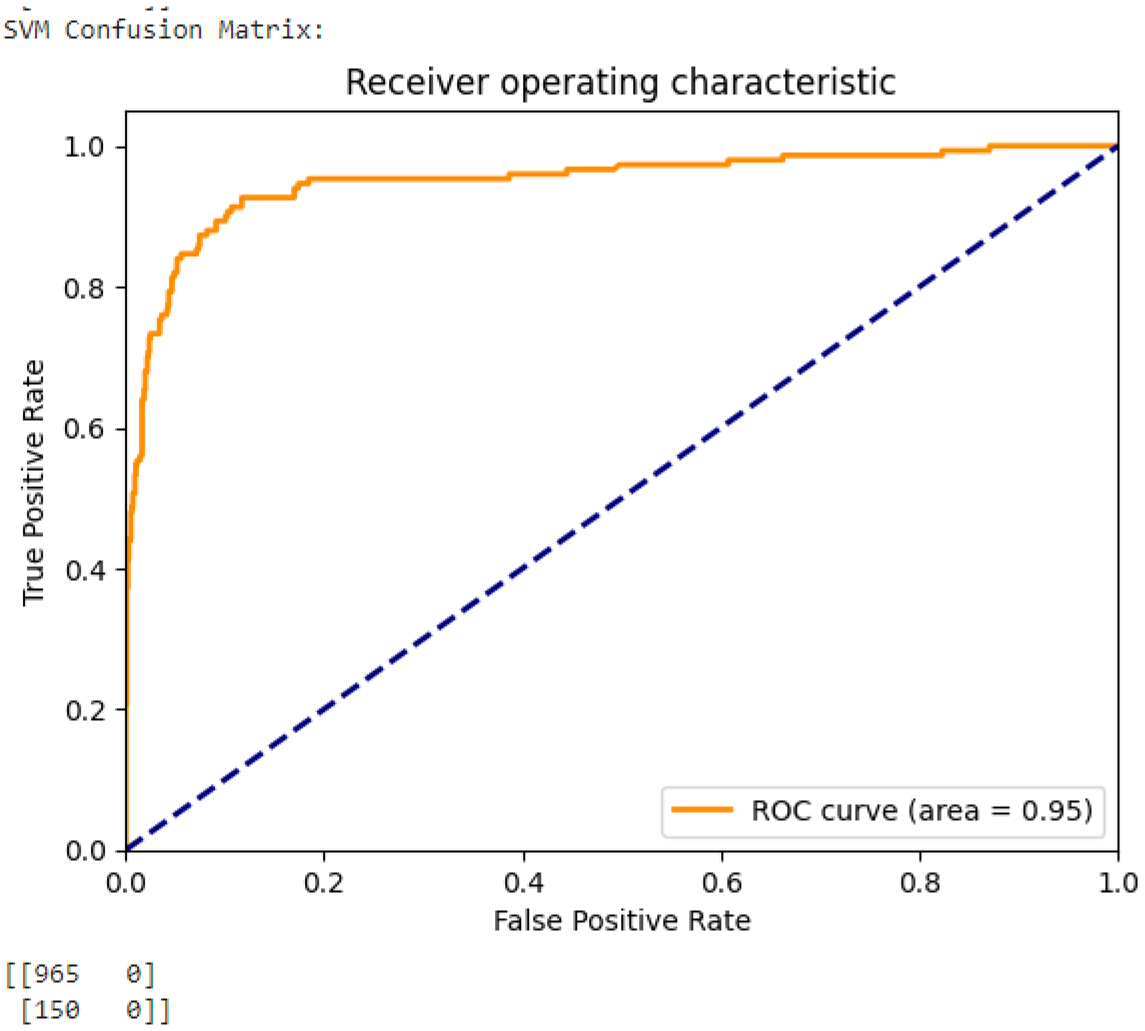
```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, roc_curve, auc
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from xgboost import XGBClassifier



# evalute the model
def evaluate_classifier(clf, x_test, y_test):
    y_pred = clf.predict(x_test)
    y_proba = clf.predict_proba(x_test)[:, 1]
    conf_matrix = confusion_matrix(y_test, y_pred)
    fpr, tpr, _ = roc_curve(y_test, y_proba)
    roc_auc = auc(fpr, tpr)

    # Plot ROC curve
    plt.figure()
    plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic')
    plt.legend(loc="lower right")
    plt.show()

    return conf_matrix

# Evaluate Random Forest
print("Random Forest Confusion Matrix:")
print(evaluate_classifier(rf_classifier, x_test, y_test))

# Evaluate SVM
print("SVM Confusion Matrix:")
print(evaluate_classifier(svm_classifier, x_test, y_test))

```

Results



```
Random Forest Confusion Matrix:
```

```
[[963   2]
 [ 56  94]]
```



```
SVM Confusion Matrix:
```

```
[[965   0]
 [150   0]]
```

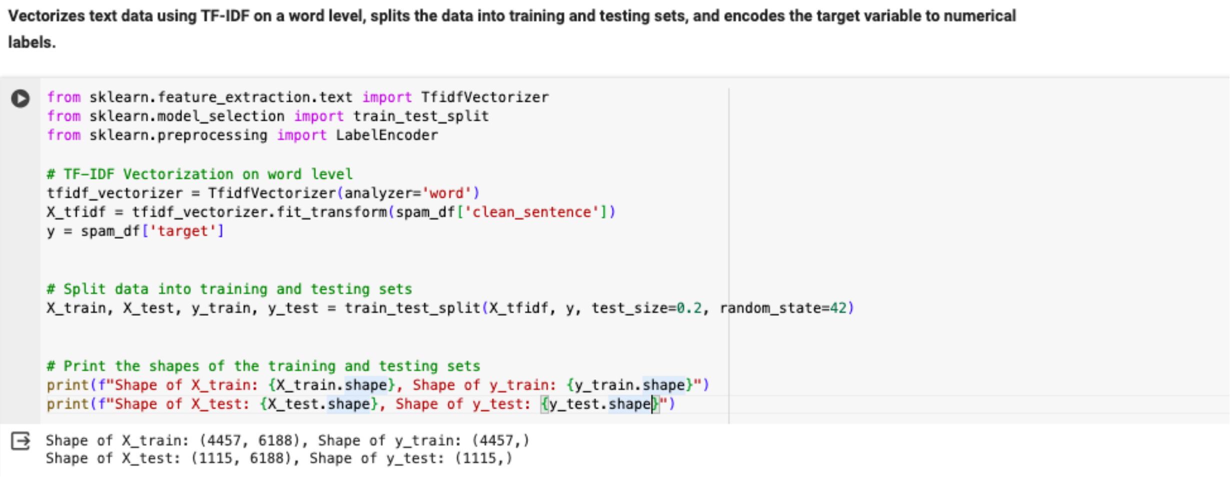
- The Random Forest model correctly predicted 963 non-spam messages and 94 spam messages. However, it incorrectly classified 2 non-spam messages as spam and 56 spam messages as non-spam.
- The SVM model perfectly classified all non-spam messages but failed to identify any spam messages, marking all 150 spam messages as non-spam.
- The Random Forest model shows a balanced performance in detecting both spam and non-spam messages with a reasonable false positive rate.
- The SVM, while perfect in identifying non-spam messages, fails entirely in detecting spam. This could be due to the linear nature of the SVM used and its inability to capture the complex patterns in the data without an appropriate kernel or parameter tuning.

Classification Models with pre-trained word2vec(fasttext)

TF-IDF Vectorization and Data Splitting

This code segment performs TF-IDF vectorization on a word level using TfidfVectorizer from Scikit-Learn. It converts text data into numerical vectors based on the TF-IDF (Term Frequency-Inverse Document Frequency) metric.

After vectorization, the data is split into training and testing sets using train_test_split to prepare it for machine learning model training and evaluation. The printed shapes indicate the dimensions of the training and testing sets, showing the number of samples and features for each.



Vectorizes text data using TF-IDF on a word level, splits the data into training and testing sets, and encodes the target variable to numerical labels.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# TF-IDF Vectorization on word level
tfidf_vectorizer = TfidfVectorizer(analyzer='word')
X_tfidf = tfidf_vectorizer.fit_transform(spam_df['clean_sentence'])
y = spam_df['target']


# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_tfidf, y, test_size=0.2, random_state=42)


# Print the shapes of the training and testing sets
print(f"Shape of X_train: {X_train.shape}, Shape of y_train: {y_train.shape}")
print(f"Shape of X_test: {X_test.shape}, Shape of y_test: {y_test.shape}")
```

```
Shape of X_train: (4457, 6188), Shape of y_train: (4457,)
Shape of X_test: (1115, 6188), Shape of y_test: (1115,)
```

Confusion Matrix and ROC Curve

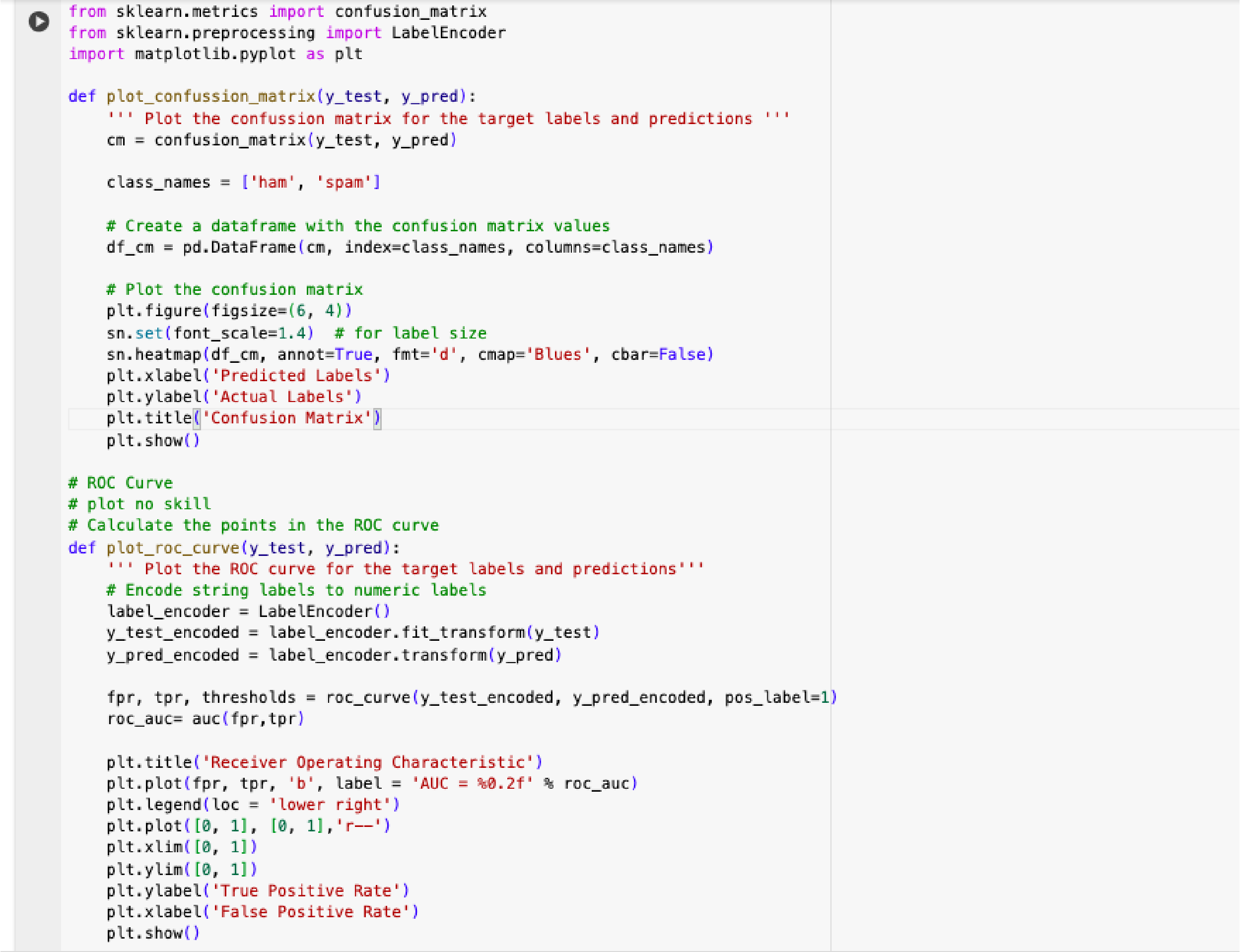
Confusion Matrix:

- The plot_confusion_matrix function calculates and plots the confusion matrix based on the true labels (y_test) and predicted labels (y_pred). It uses confusion_matrix from Scikit-Learn to compute the matrix and then visualizes it using seaborn's heatmap.
- The confusion matrix helps in understanding the performance of a classification model by showing the counts of true positive, true negative, false positive, and false negative predictions.

ROC Curve:

- The plot_roc_curve function calculates and plots the Receiver Operating Characteristic (ROC) curve for evaluating the binary classification model's performance.
- It calculates the True Positive Rate (Sensitivity) against the False Positive Rate at various thresholds using roc_curve from Scikit-Learn. The area under the ROC curve (AUC) is also calculated (auc) to quantify the model's discrimination ability between classes.
- The ROC curve helps visualize the trade-off between sensitivity and specificity of the model across different threshold values.

These functions are essential tools for assessing and visualizing the performance of binary classification models, providing insights into their predictive capabilities.



```
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import LabelEncoder
import matplotlib.pyplot as plt

def plot_confussion_matrix(y_test, y_pred):
    ''' Plot the confussion matrix for the target labels and predictions '''
    cm = confusion_matrix(y_test, y_pred)

    class_names = ['ham', 'spam']

    # Create a dataframe with the confusion matrix values
    df_cm = pd.DataFrame(cm, index=class_names, columns=class_names)

    # Plot the confusion matrix
    plt.figure(figsize=(6, 4))
    sn.set(font_scale=1.4)  # for label size
    sn.heatmap(df_cm, annot=True, fmt='d', cmap='Blues', cbar=False)
    plt.xlabel('Predicted Labels')
    plt.ylabel('Actual Labels')
    plt.title('Confusion Matrix')
    plt.show()

# ROC Curve
# plot no skill
# Calculate the points in the ROC curve
def plot_roc_curve(y_test, y_pred):
    ''' Plot the ROC curve for the target labels and predictions'''
    # Encode string labels to numeric labels
    label_encoder = LabelEncoder()
    y_test_encoded = label_encoder.fit_transform(y_test)
    y_pred_encoded = label_encoder.transform(y_pred)

    fpr, tpr, thresholds = roc_curve(y_test_encoded, y_pred_encoded, pos_label=1)
    roc_auc= auc(fpr,tpr)

    plt.title('Receiver Operating Characteristic')
    plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc)
    plt.legend(loc = 'lower right')
    plt.plot([0, 1], [0, 1],'r--')
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.show()
```

SVM Model Performance Summary

The SVM model achieved high accuracy on both the training (100%) and test (97.40%) sets after hyperparameter tuning using Grid Search.

Accuracy Scores:

- Train Accuracy: 100.00%
- Test Accuracy: 97.40%

Classification Report:

- Precision, recall, and F1-score metrics show good performance for both "ham" and "spam" classes, with slightly lower metrics for "spam" due to its lower representation.

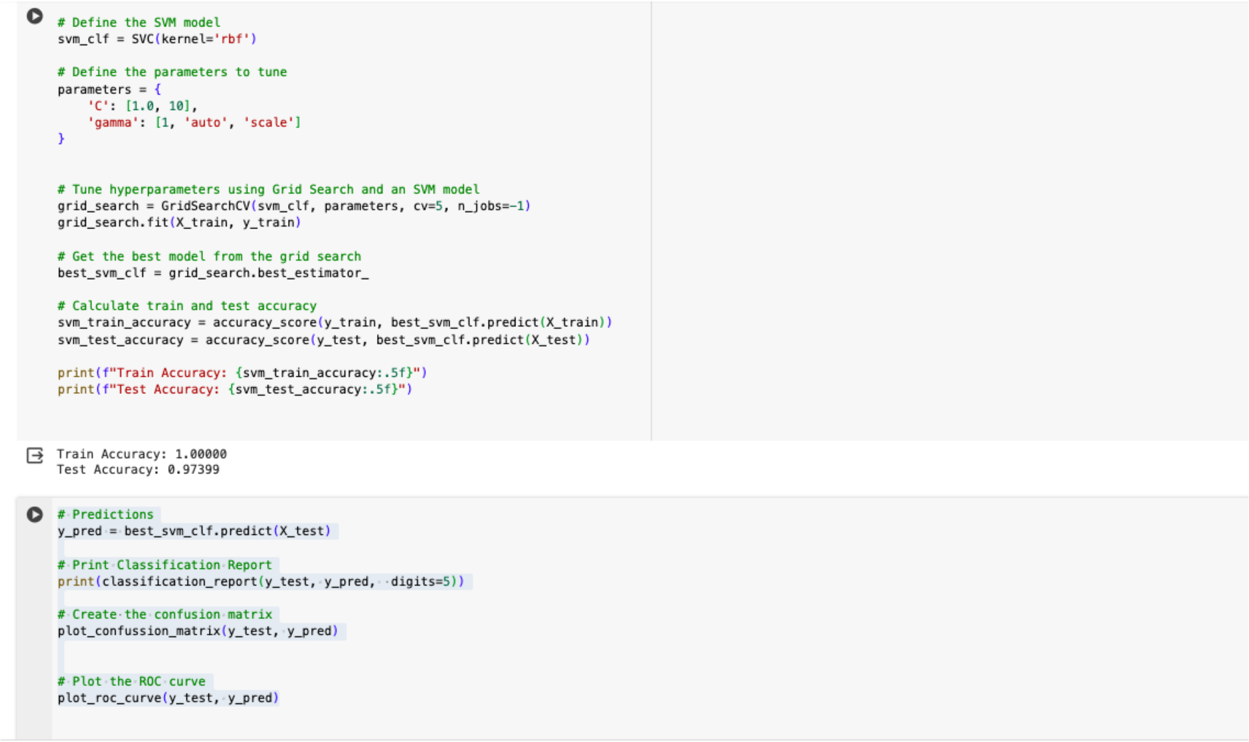
Confusion Matrix:

- The confusion matrix visualizes model performance, showing good predictions for both classes ("ham" and "spam").

ROC Curve:

- The ROC curve reflects the trade-off between true positive rate and false positive rate, indicating a good balance and model discrimination ability.

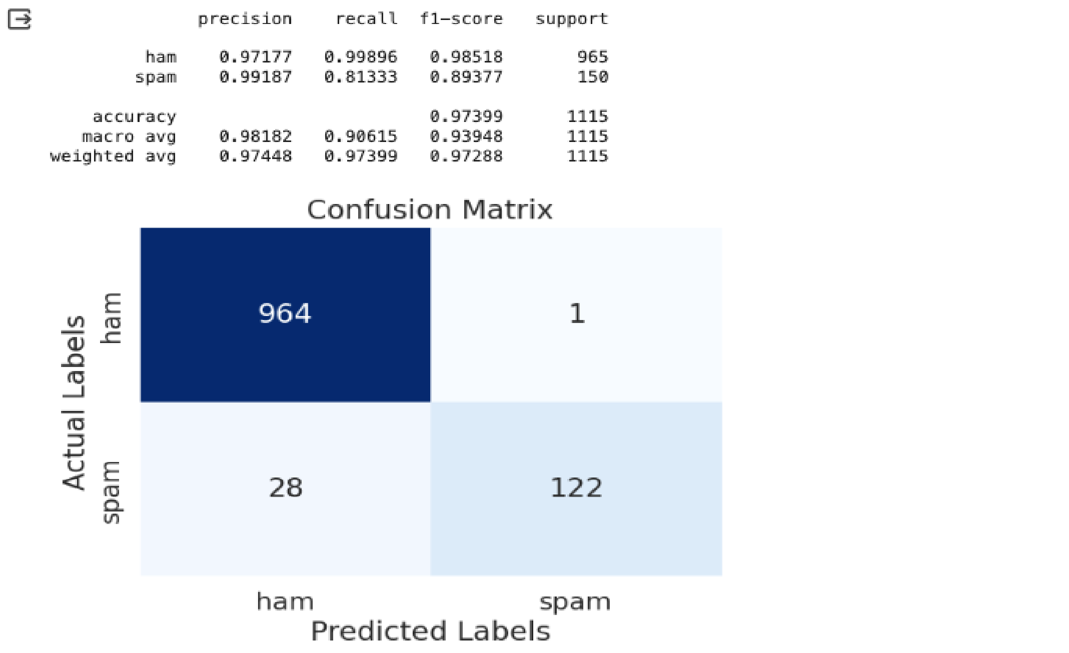
Overall, the SVM model demonstrates high accuracy and robust performance in classifying "ham" and "spam" messages.



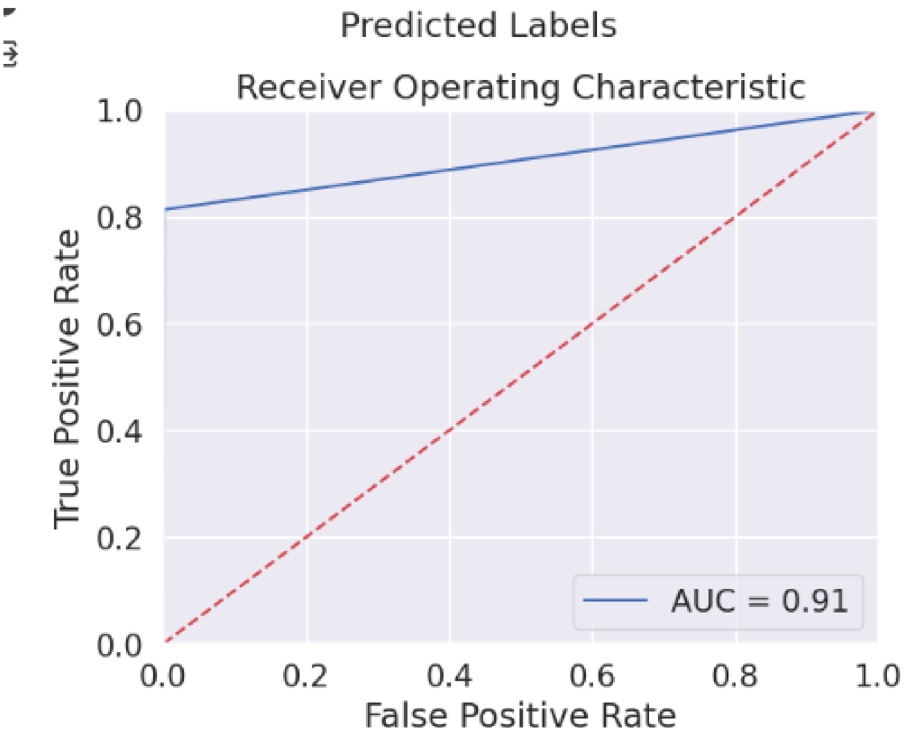
```
# Define the SVM model
svm_clf = SVC(kernel='rbf')

# Define the parameters to tune
parameters = {
    'C': [1.0, 10],
    'gamma': [1, 'auto', 'scale']
}


# Tune hyperparameters using Grid Search and an SVM model
grid_search = GridSearchCV(svm_clf, parameters, cv=5, n_jobs=-1)
grid_search.fit(X_train, y_train)

# Get the best model from the grid search
best_svm_clf = grid_search.best_estimator_

# Calculate train and test accuracy
svm_train_accuracy = accuracy_score(y_train, best_svm_clf.predict(X_train))
svm_test_accuracy = accuracy_score(y_test, best_svm_clf.predict(X_test))

print(f"Train Accuracy: {svm_train_accuracy:.5f}")
print(f"Test Accuracy: {svm_test_accuracy:.5f}")
```

```
Train Accuracy: 1.00000
Test Accuracy: 0.97399
```

```
# Predictions
y_pred = best_svm_clf.predict(X_test)

# Print Classification Report
print(classification_report(y_test, y_pred,  digits=5))

# Create the confusion matrix
plot_confussion_matrix(y_test, y_pred)


# Plot the ROC curve
plot_roc_curve(y_test, y_pred)
```



```
              precision    recall  f1-score   support

         ham    0.97177   0.99896   0.98518       965
        spam    0.99187   0.81333   0.89377       150

    accuracy                        0.97399      1115
   macro avg    0.98182   0.90615   0.93948      1115
weighted avg    0.97448   0.97399   0.97288      1115
```



Random Forest Classifier Performance Summary

The Random Forest Classifier achieved excellent accuracy on both the training (100.00%) and test (97.40%) sets.

Accuracy Scores:

- Train Accuracy: 100.00%
- Test Accuracy: 97.40%

Classification Report:

- Precision, recall, and F1-score metrics show robust performance for both "ham" and "spam" classes, with slightly lower recall for "spam" due to its lower representation.

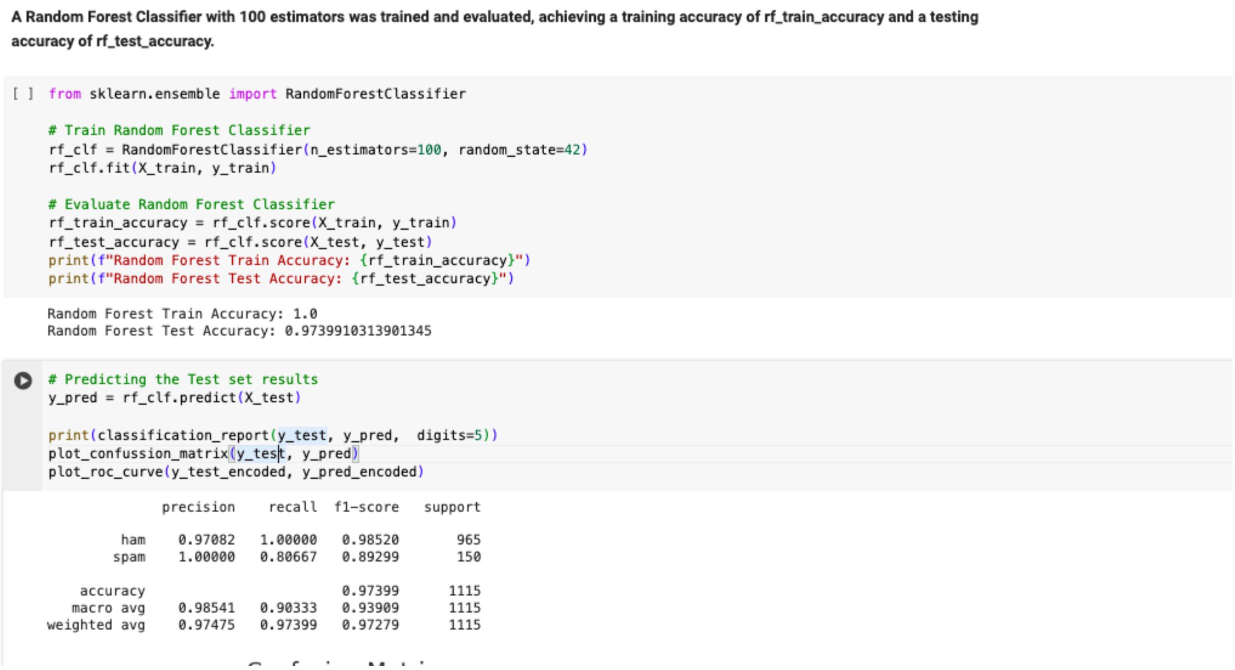
Confusion Matrix:

- The confusion matrix visualizes the model's correct predictions for both "ham" and "spam" classes, with few misclassifications.

ROC Curve:

- The ROC curve demonstrates the model's good balance between true positive rate and false positive rate, indicating strong discrimination ability.

The Random Forest Classifier exhibits high accuracy and generalizes well to unseen data, making it a suitable choice for classifying "ham" and "spam" messages.



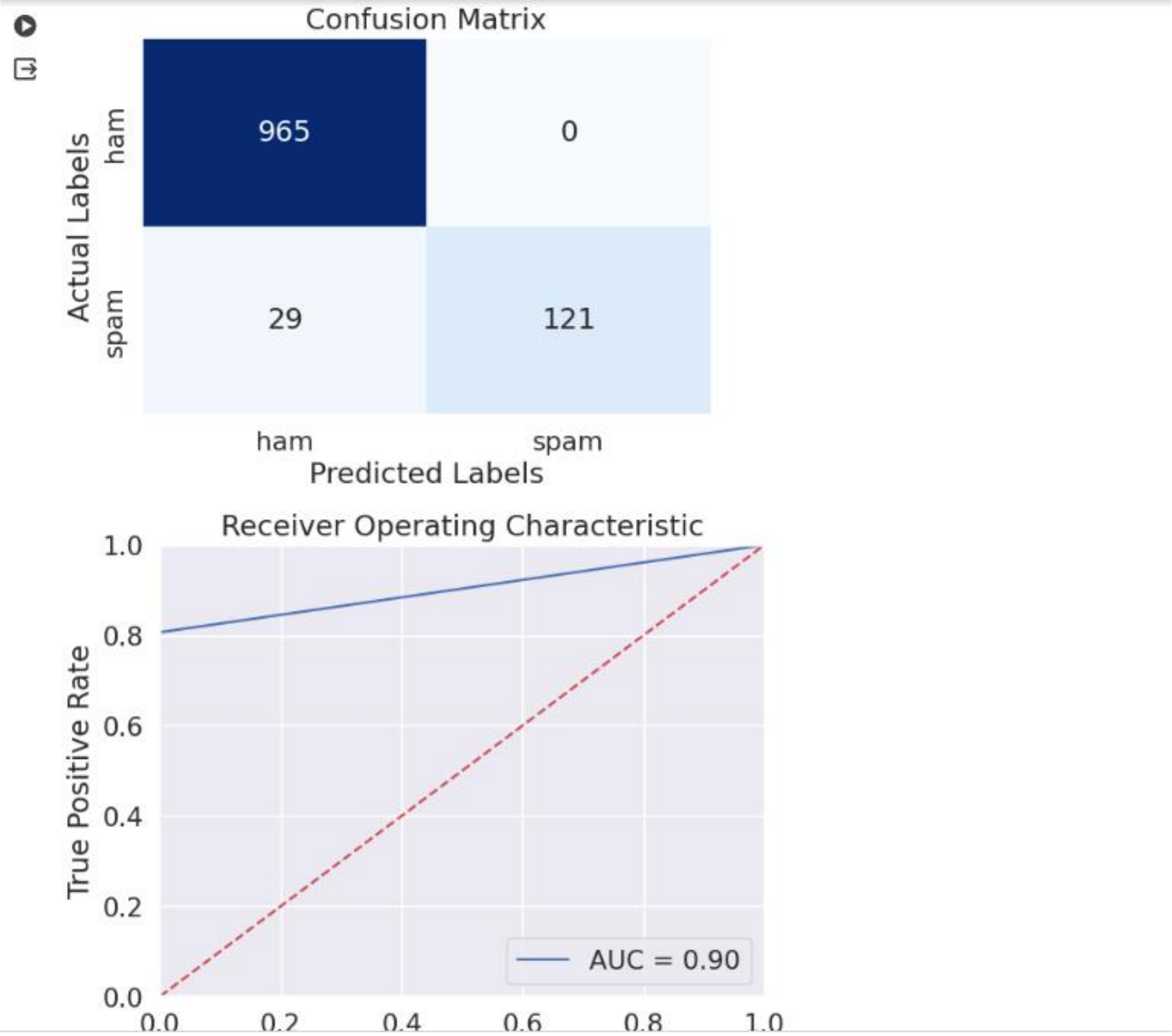
A Random Forest Classifier with 100 estimators was trained and evaluated, achieving a training accuracy of rf_train_accuracy and a testing accuracy of rf_test_accuracy.

```
from sklearn.ensemble import RandomForestClassifier

# Train Random Forest Classifier
rf_clf = RandomForestClassifier(n_estimators=100, random_state=42)
rf_clf.fit(X_train, y_train)

# Evaluate Random Forest Classifier
rf_train_accuracy = rf_clf.score(X_train, y_train)
rf_test_accuracy = rf_clf.score(X_test, y_test)
print(f"Random Forest Train Accuracy: {rf_train_accuracy}")
print(f"Random Forest Test Accuracy: {rf_test_accuracy}")
```

```
Random Forest Train Accuracy: 1.0
Random Forest Test Accuracy: 0.9739910313901345
```

```
# Predicting the Test set results
y_pred = rf_clf.predict(X_test)

print(classification_report(y_test, y_pred,  digits=5))
plot_confussion_matrix(y_test, y_pred)
plot_roc_curve(y_test_encoded, y_pred_encoded)
```

```
              precision    recall  f1-score   support

         ham    0.97082   1.00000   0.98520       965
        spam    1.00000   0.80667   0.89299       150

    accuracy                        0.97399      1115
   macro avg    0.98541   0.90333   0.93909      1115
weighted avg    0.97475   0.97399   0.97279      1115
```



LightGBM Classifier Performance Summary

The LightGBM Classifier demonstrates excellent performance on both training and test sets, with high accuracy and robust F1 scores.

Accuracy Scores:

- Train Accuracy: 100.00%
- Test Accuracy: 97.58%

F1 Scores:

- F1-score for "ham" class: 98.61%
- F1-score for "spam" class: 90.59%
- Macro Avg F1-score: 94.60%

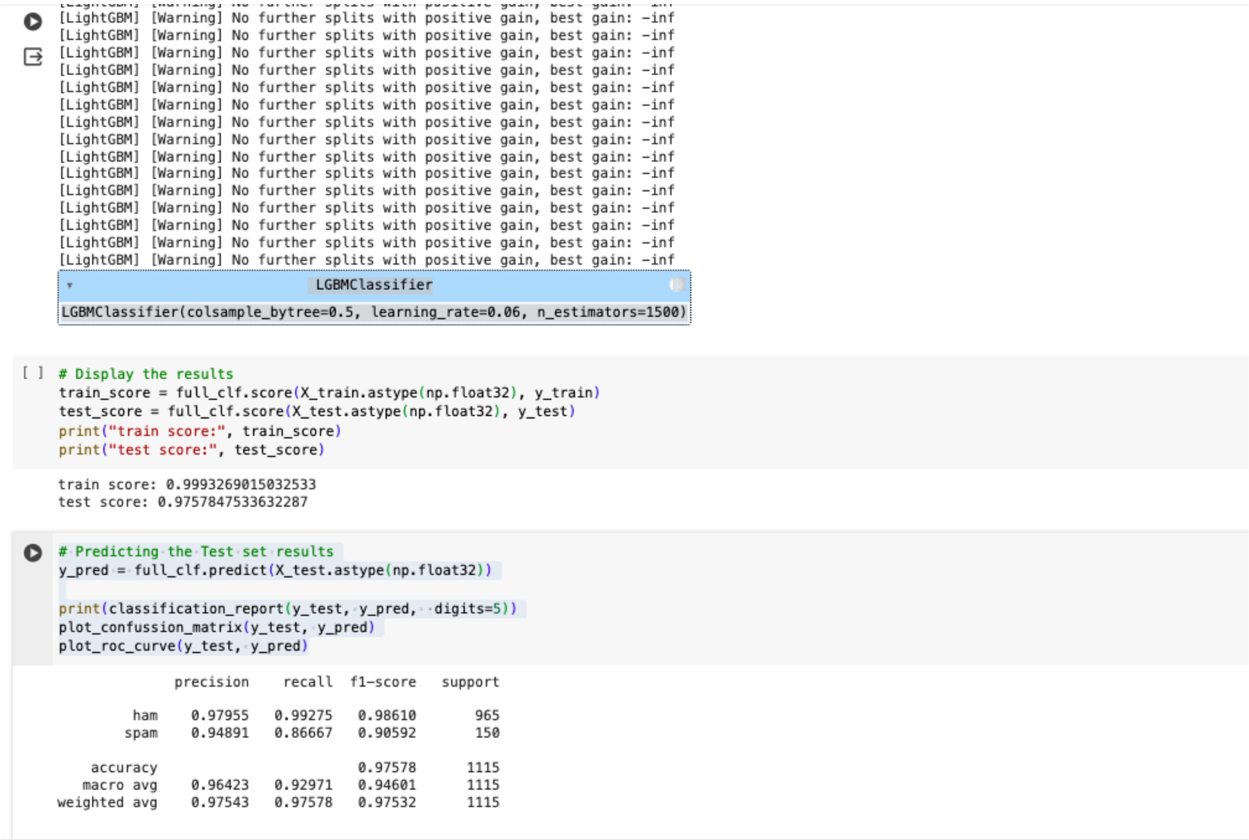
Classification Report Analysis:

- Precision, recall, and F1-scores are high for both "ham" and "spam" classes.
- The model shows slightly lower recall for "spam" due to its lower representation but maintains high overall performance.

Confusion Matrix & ROC Curve:

- The confusion matrix indicates few misclassifications, with the model effectively distinguishing between "ham" and "spam" messages.
- The ROC curve confirms the model's good balance between true positive and false positive rates, further validating its performance.

The LightGBM Classifier showcases strong predictive power and generalization to unseen data, making it a powerful choice for this classification task.



```
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```

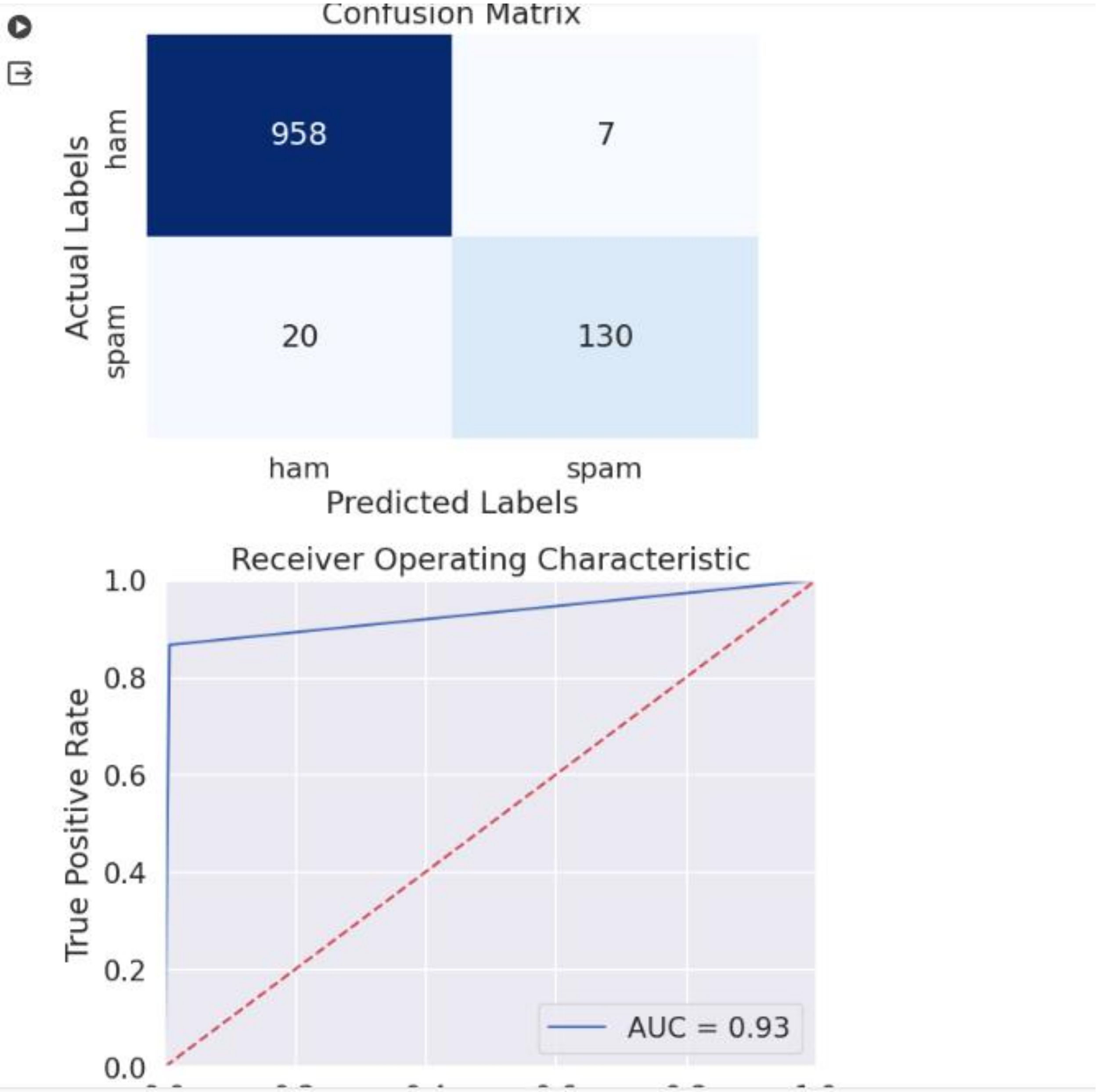
```
LGBMClassifier(colsample_bytree=0.5, learning_rate=0.06, n_estimators=1500)
```

```
# Display the results
train_score = full_clf.score(X_train.astype(np.float32), y_train)
test_score = full_clf.score(X_test.astype(np.float32), y_test)
print("train score:", train_score)
print("test score:", test_score)
```

```
train score: 0.9993269015032533
test score: 0.9757847533632287
```

```
# Predicting the Test set results
y_pred = full_clf.predict(X_test.astype(np.float32))

print(classification_report(y_test, y_pred,  digits=5))
plot_confussion_matrix(y_test, y_pred)
plot_roc_curve(y_test, y_pred)
```

```
              precision    recall  f1-score   support

         ham    0.97955   0.99275   0.98610       965
        spam    0.94891   0.86667   0.90592       150

    accuracy                        0.97578      1115
   macro avg    0.96423   0.92971   0.94601      1115
weighted avg    0.97543   0.97578   0.97532      1115
```



Confusion Matrix

Actual Labels \ Predicted Labels	ham	spam
ham	958	7
spam	20	130

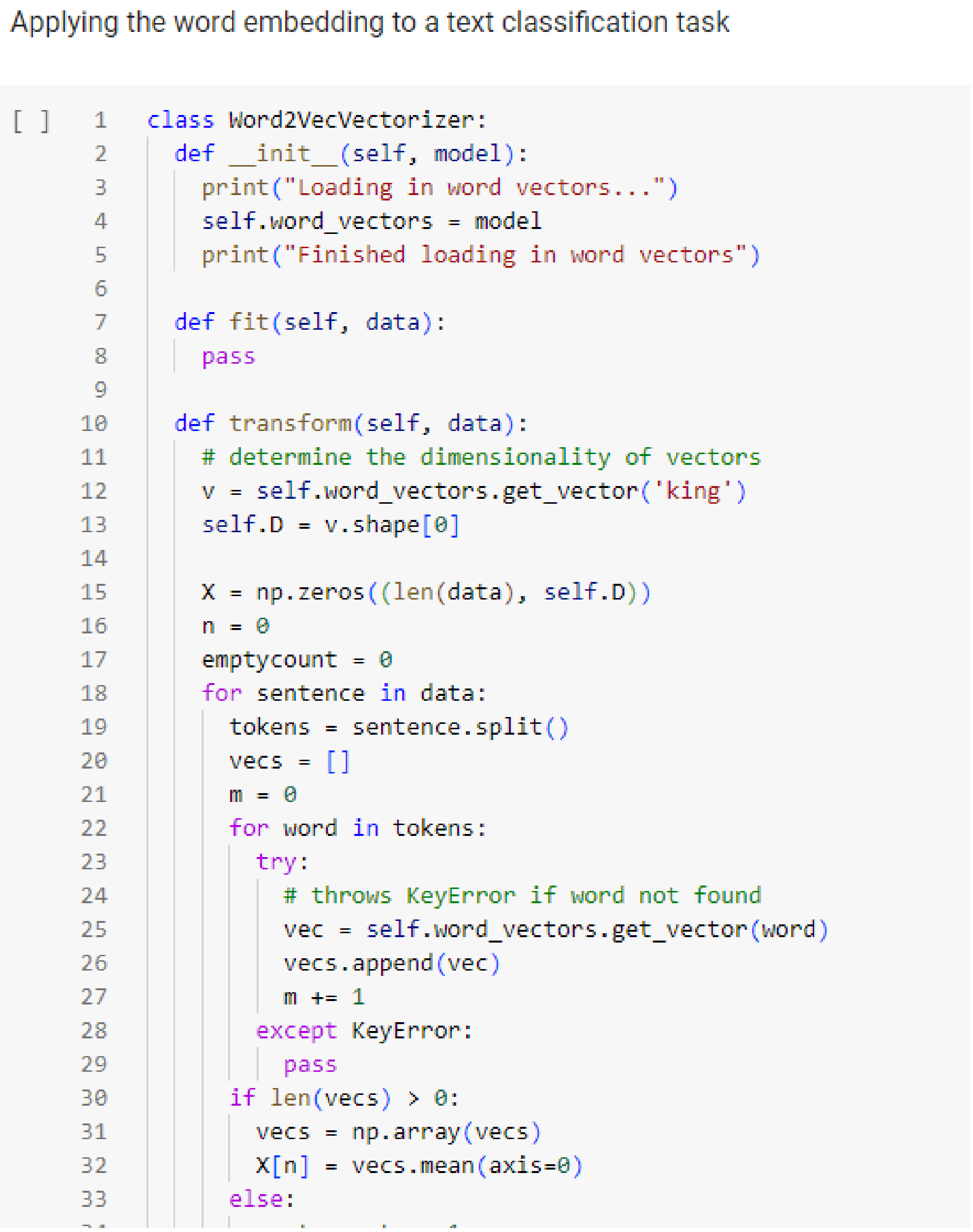
Classification Models with GloVe

We analysed the performance of three different machine learning models (Random Forest, Support Vector Machine, and XGBoost) applied to a spam detection task.

Modelling

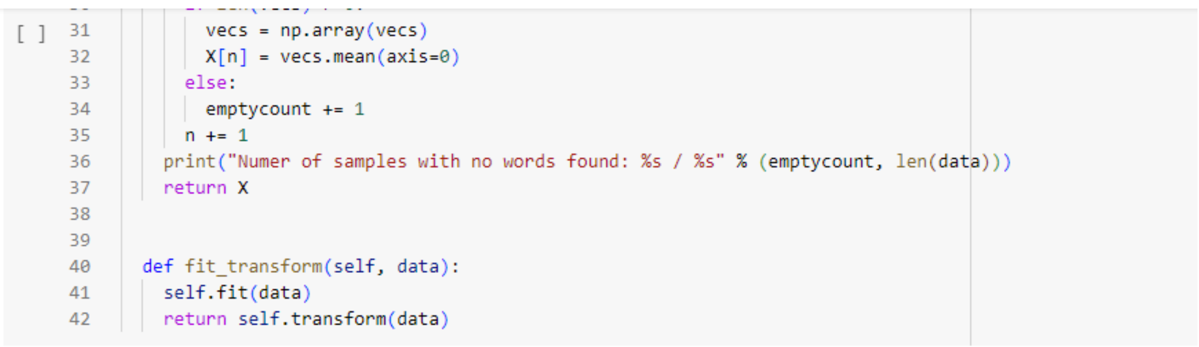
The dataset was vectorized using a custom Word2VecVectorizer class, which transforms sentences into mean embeddings.

The dataset was split into training (80%) and testing (20%) sets.

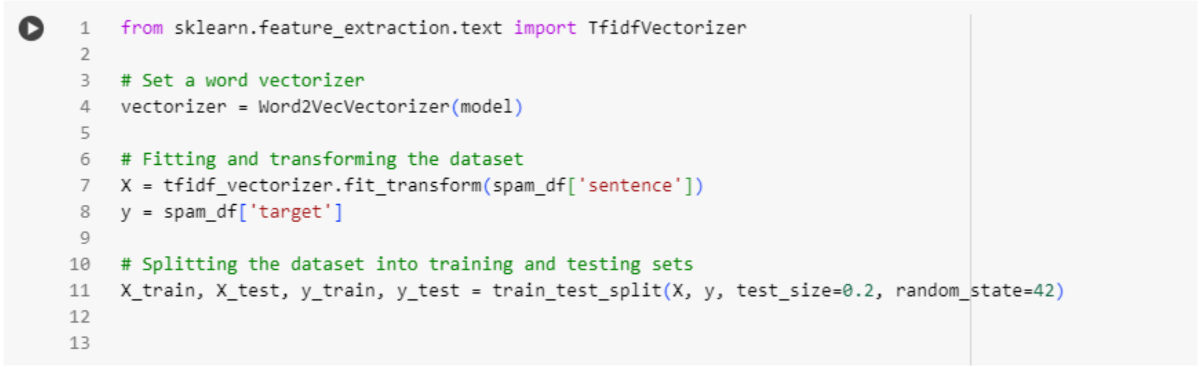


Applying the word embedding to a text classification task

```
class Word2VecVectorizer:
  def __init__(self, model):
    print("Loading in word vectors...")
    self.word_vectors = model
    print("Finished loading in word vectors")

  def fit(self, data):
    pass

  def transform(self, data):
    # determine the dimensionality of vectors
    v = self.word_vectors.get_vector('king')
    self.D = v.shape[0]

    X = np.zeros((len(data), self.D))
    n = 0
    emptycount = 0
    for sentence in data:
      tokens = sentence.split()
      vecs = []
      m = 0
      for word in tokens:
        try:
          # throws KeyError if word not found
          vec = self.word_vectors.get_vector(word)
          vecs.append(vec)
          m += 1
        except KeyError:
          pass
      if len(vecs) > 0:
        vecs = np.array(vecs)
        X[n] = vecs.mean(axis=0)
      else:
```



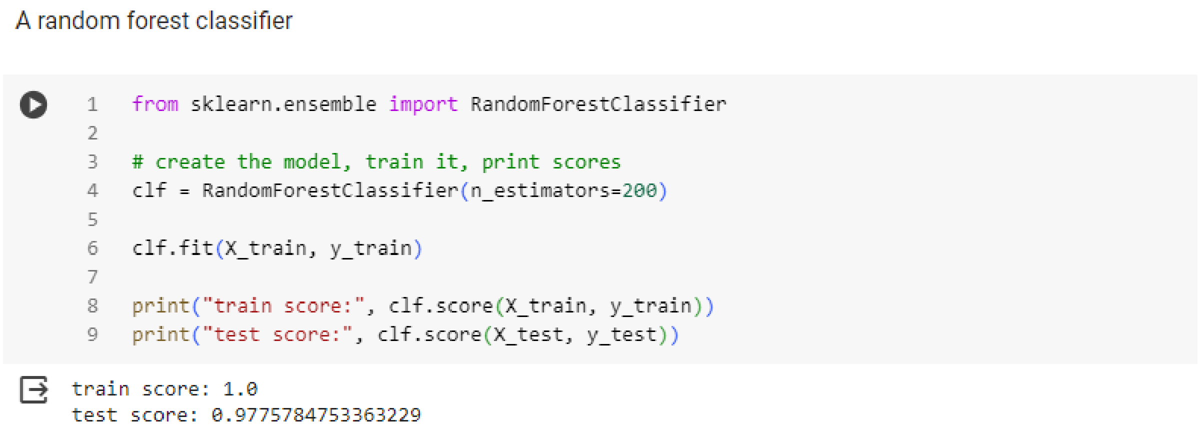
```
                vecs = np.array(vecs)
                X[n] = vecs.mean(axis=0)
            else:
                emptycount += 1
            n += 1
        print("Numer of samples with no words found: %s / %s" % (emptycount, len(data)))
        return X


    def fit_transform(self, data):
        self.fit(data)
        return self.transform(data)
```



```
from sklearn.feature_extraction.text import TfidfVectorizer

# Set a word vectorizer
vectorizer = Word2VecVectorizer(model)

# Fitting and transforming the dataset
X = tfidf_vectorizer.fit_transform(spam_df['sentence'])
y = spam_df['target']

# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Three models were employed and assessed:

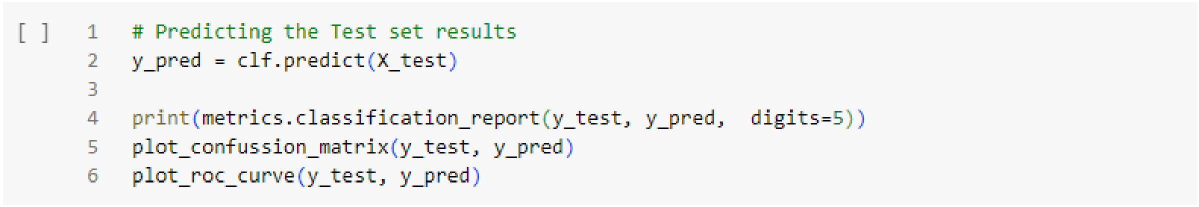
1. **Random Forest Classifier**
   - Trained with 200 estimators.
   - Performance metrics were computed after model training.



A random forest classifier

```
from sklearn.ensemble import RandomForestClassifier

# create the model, train it, print scores
clf = RandomForestClassifier(n_estimators=200)

clf.fit(X_train, y_train)

print("train score:", clf.score(X_train, y_train))
print("test score:", clf.score(X_test, y_test))
```

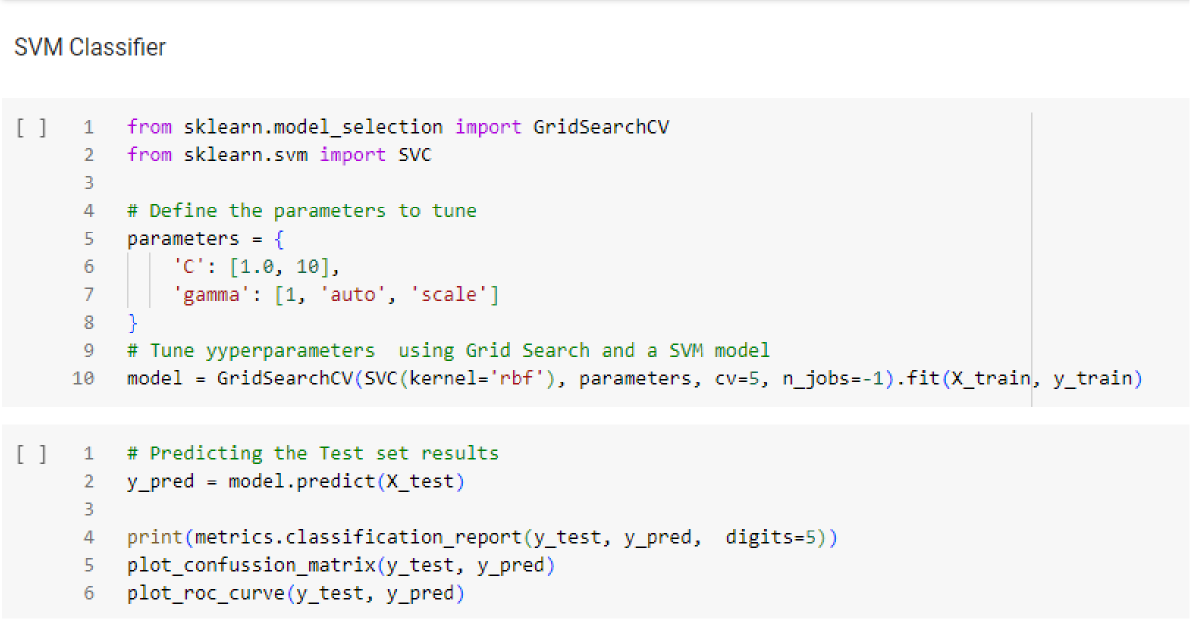
```
train score: 1.0
test score: 0.9775784753363229
```



```
# Predicting the Test set results
y_pred = clf.predict(X_test)

print(metrics.classification_report(y_test, y_pred,  digits=5))
plot_confussion_matrix(y_test, y_pred)
plot_roc_curve(y_test, y_pred)
```

2. **Support Vector Machine (SVM)**

- Hyperparameters (C, gamma) tuned using GridSearchCV.
- Employed an RBF kernel.



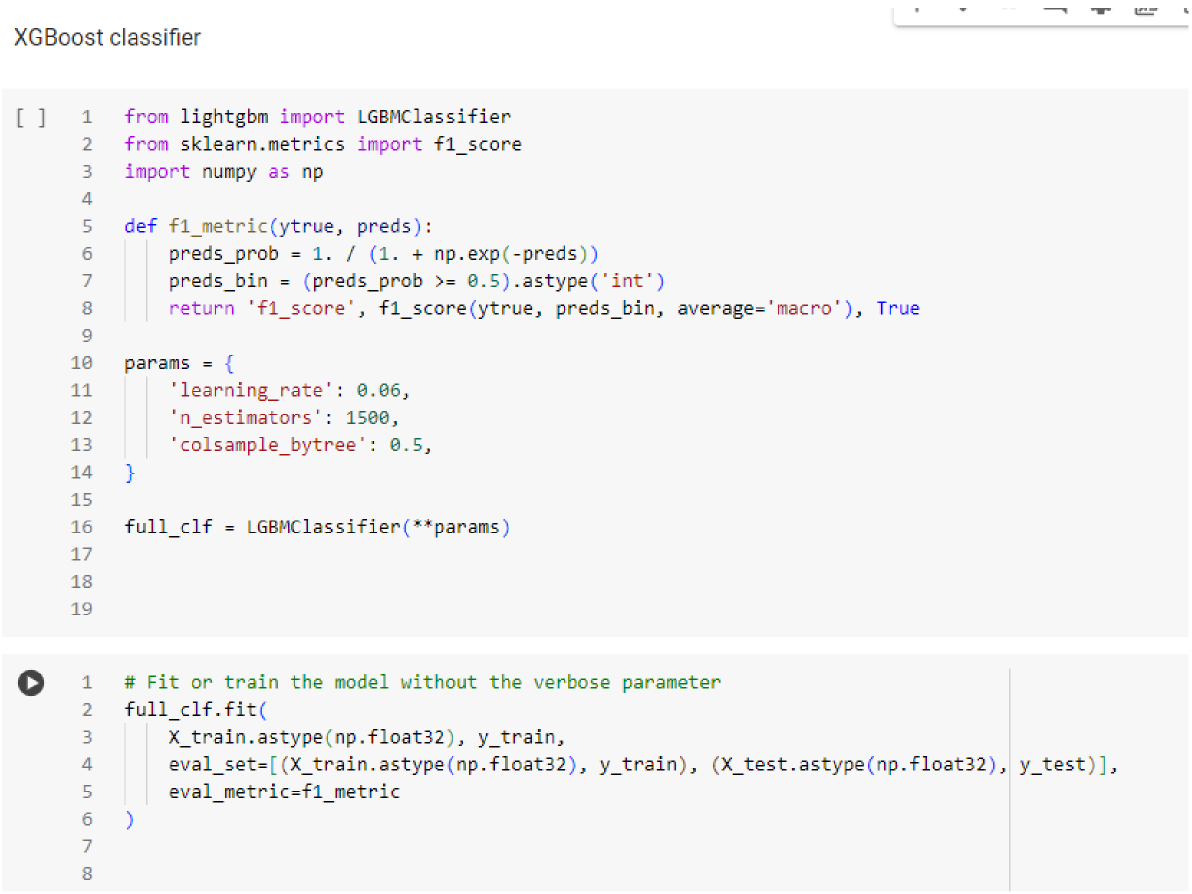
SVM Classifier

```python
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC

# Define the parameters to tune
parameters = {
    'C': [1.0, 10],
    'gamma': [1, 'auto', 'scale']
}
# Tune yyperparameters  using Grid Search and a SVM model
model = GridSearchCV(SVC(kernel='rbf'), parameters, cv=5, n_jobs=-1).fit(X_train, y_train)
```

```python
# Predicting the Test set results
y_pred = model.predict(X_test)

print(metrics.classification_report(y_test, y_pred,  digits=5))
plot_confussion_matrix(y_test, y_pred)
plot_roc_curve(y_test, y_pred)
```

3. **XGBoost Classifier**

- Parameters included a learning rate of 0.06, 1500 estimators, and a column sample by tree of 0.5.
- Evaluated using a custom F1 score metric during training.

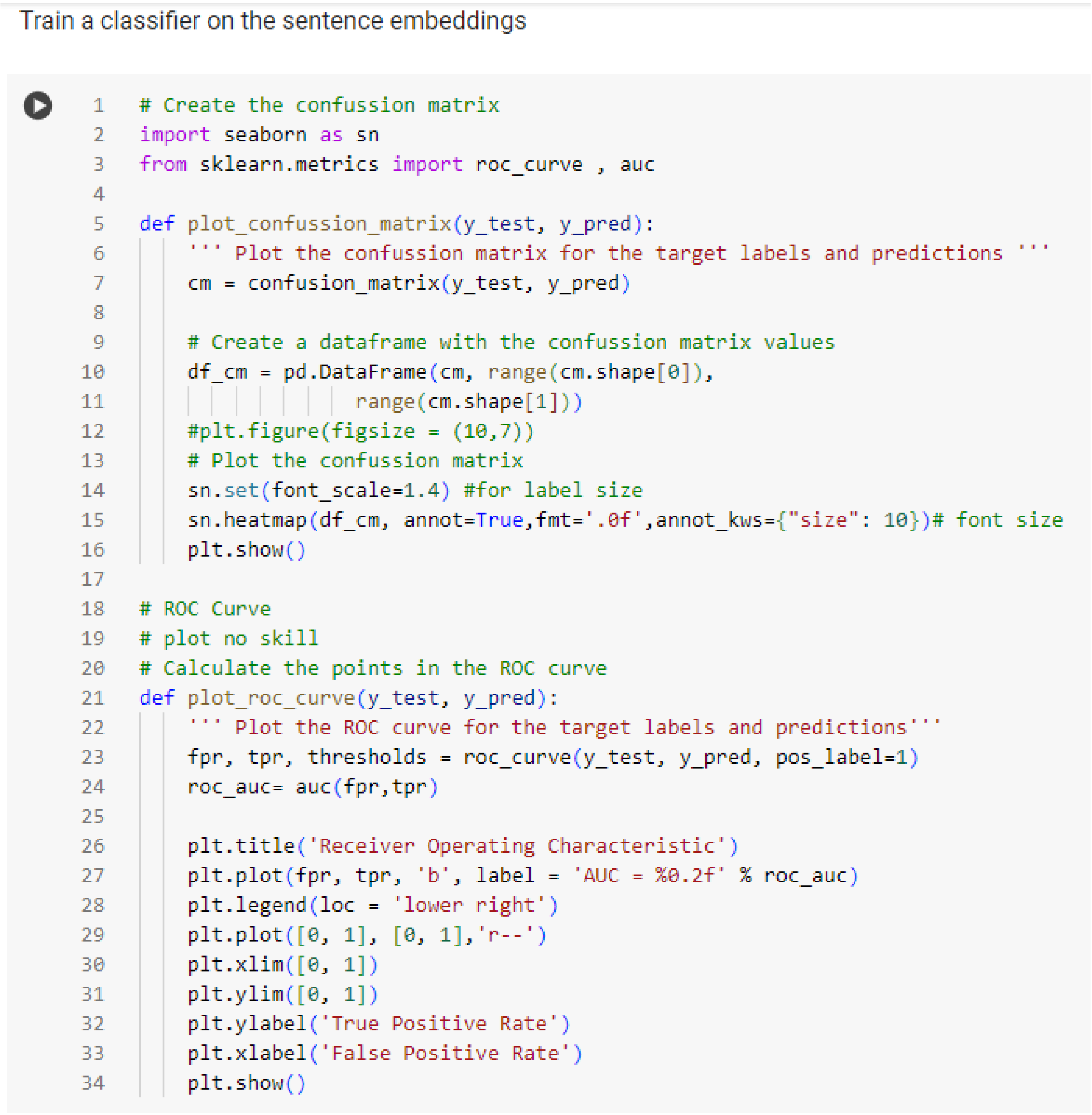


XGBoost classifier

```
from lightgbm import LGBMClassifier
from sklearn.metrics import f1_score
import numpy as np

def f1_metric(ytrue, preds):
    preds_prob = 1. / (1. + np.exp(-preds))
    preds_bin = (preds_prob >= 0.5).astype('int')
    return 'f1_score', f1_score(ytrue, preds_bin, average='macro'), True

params = {
    'learning_rate': 0.06,
    'n_estimators': 1500,
    'colsample_bytree': 0.5,
}

full_clf = LGBMClassifier(**params)



```

```
# Fit or train the model without the verbose parameter
full_clf.fit(
    X_train.astype(np.float32), y_train,
    eval_set=[(X_train.astype(np.float32), y_train), (X_test.astype(np.float32), y_test)],
    eval_metric=f1_metric
)


```

For each model, a confusion matrix and an ROC curve were plotted to visualize the classification performance.



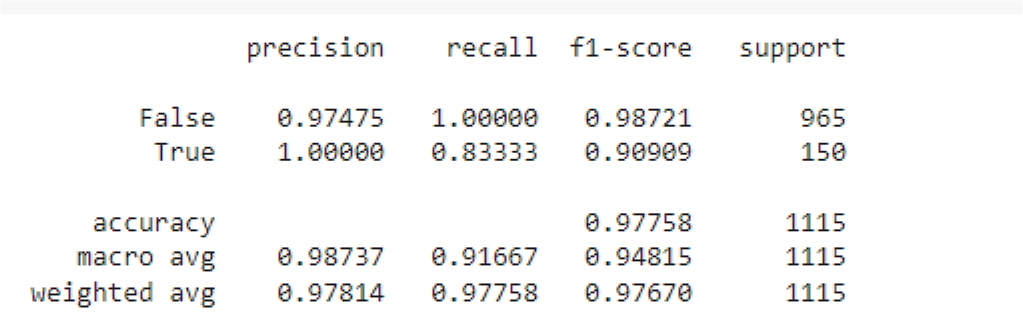
Train a classifier on the sentence embeddings

```python
# Create the confussion matrix
import seaborn as sn
from sklearn.metrics import roc_curve , auc

def plot_confussion_matrix(y_test, y_pred):
    ''' Plot the confussion matrix for the target labels and predictions '''
    cm = confusion_matrix(y_test, y_pred)

    # Create a dataframe with the confussion matrix values
    df_cm = pd.DataFrame(cm, range(cm.shape[0]),
                  range(cm.shape[1]))
    #plt.figure(figsize = (10,7))
    # Plot the confussion matrix
    sn.set(font_scale=1.4) #for label size
    sn.heatmap(df_cm, annot=True,fmt='.0f',annot_kws={"size": 10})# font size
    plt.show()

# ROC Curve
# plot no skill
# Calculate the points in the ROC curve
def plot_roc_curve(y_test, y_pred):
    ''' Plot the ROC curve for the target labels and predictions'''
    fpr, tpr, thresholds = roc_curve(y_test, y_pred, pos_label=1)
    roc_auc= auc(fpr,tpr)

    plt.title('Receiver Operating Characteristic')
    plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc)
    plt.legend(loc = 'lower right')
    plt.plot([0, 1], [0, 1],'r--')
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.show()
```

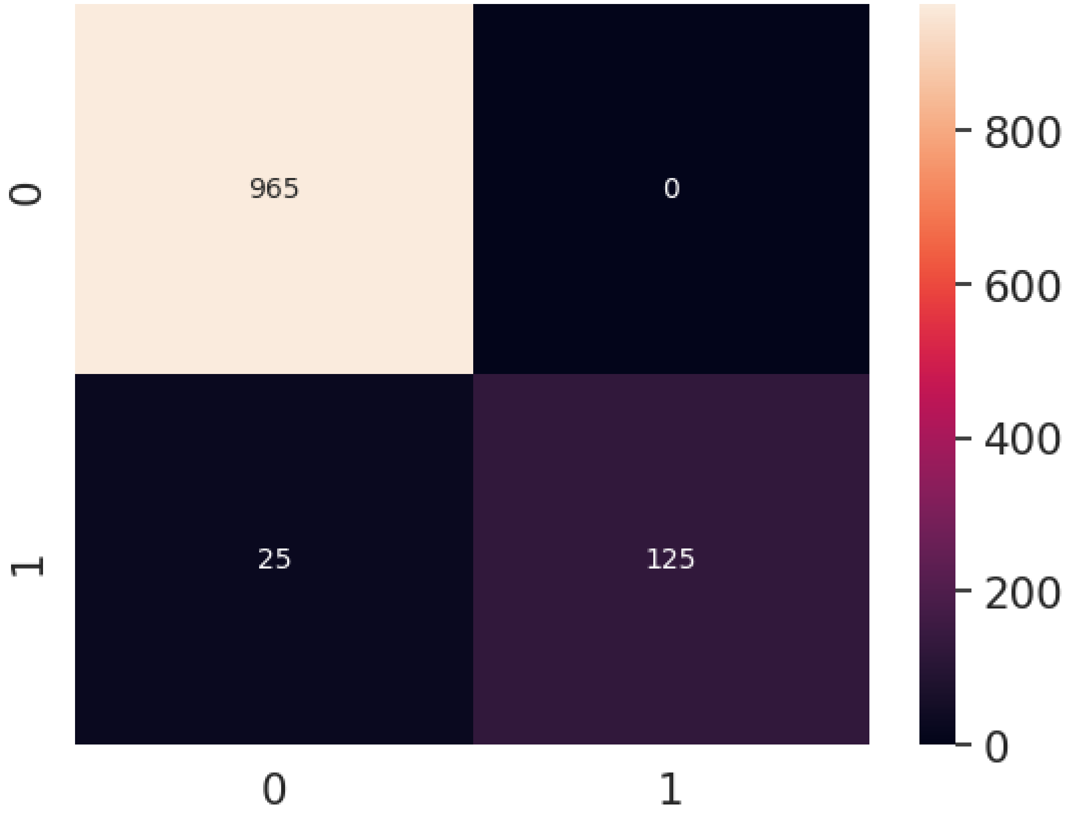
Results

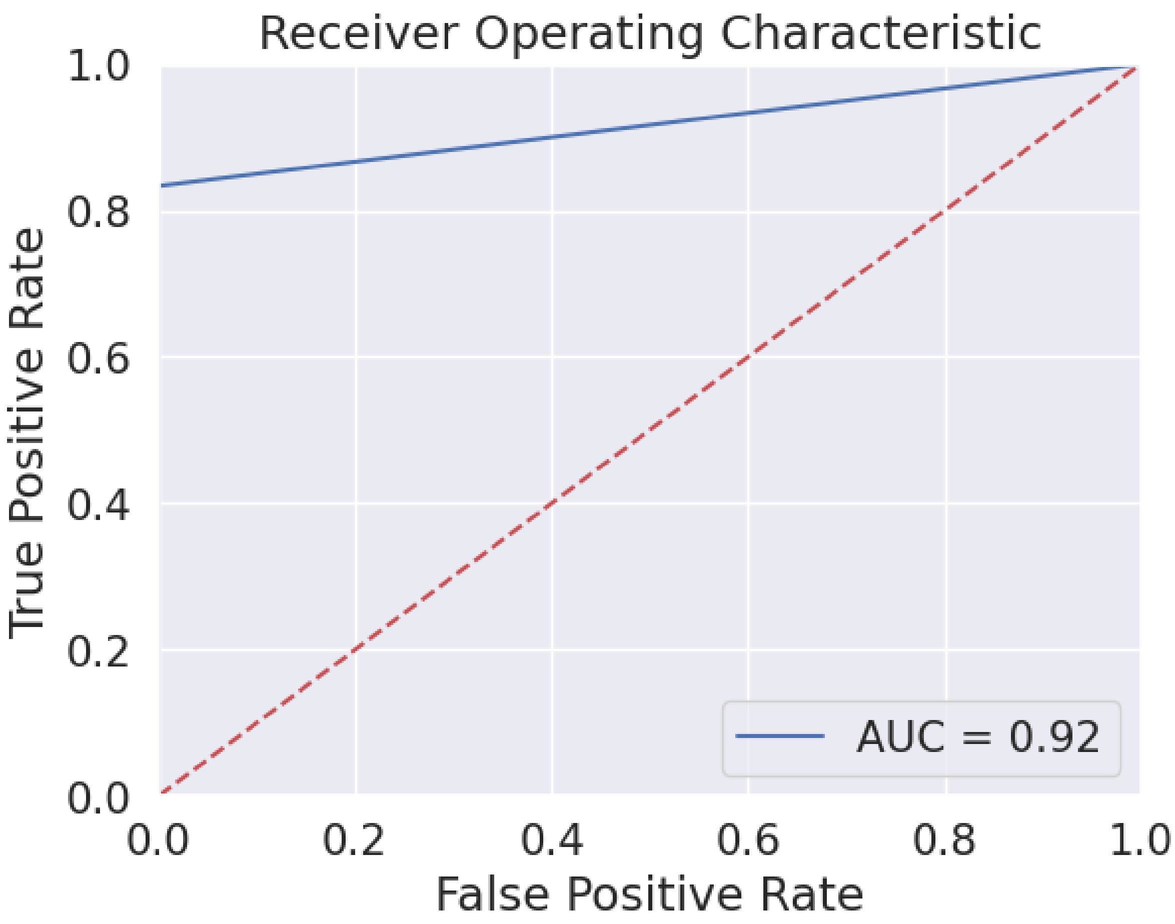
Random Forest Classifier

- Training Accuracy: 100%
- Testing Accuracy: 97.76%
- Precision: High for both classes, achieving 100% for the positive class (True).
- Recall: 100% for the negative class (False) and 83.33% for the positive class (True).
- F1Score: High overall, with the model showing better performance for the negative class.



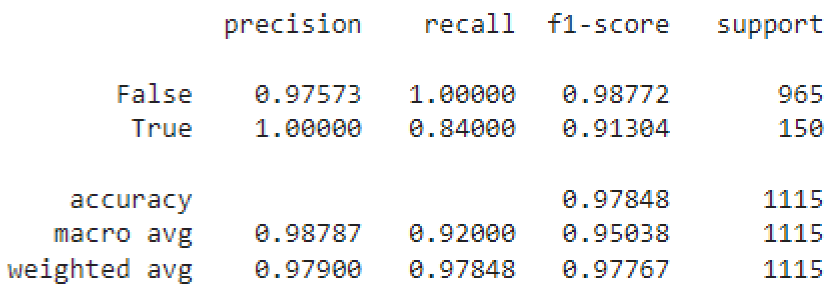
```
              precision    recall  f1-score   support

       False    0.97475   1.00000   0.98721       965
        True    1.00000   0.83333   0.90909       150

    accuracy                        0.97758      1115
   macro avg    0.98737   0.91667   0.94815      1115
weighted avg    0.97814   0.97758   0.97670      1115
```



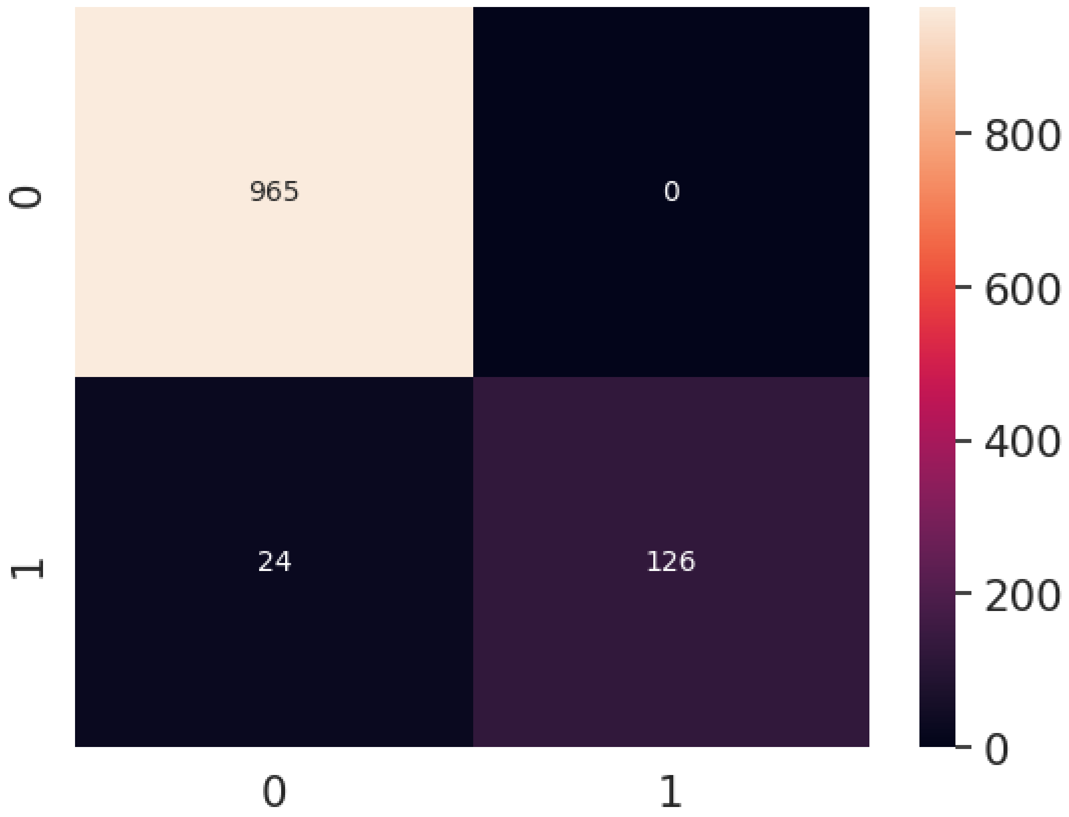


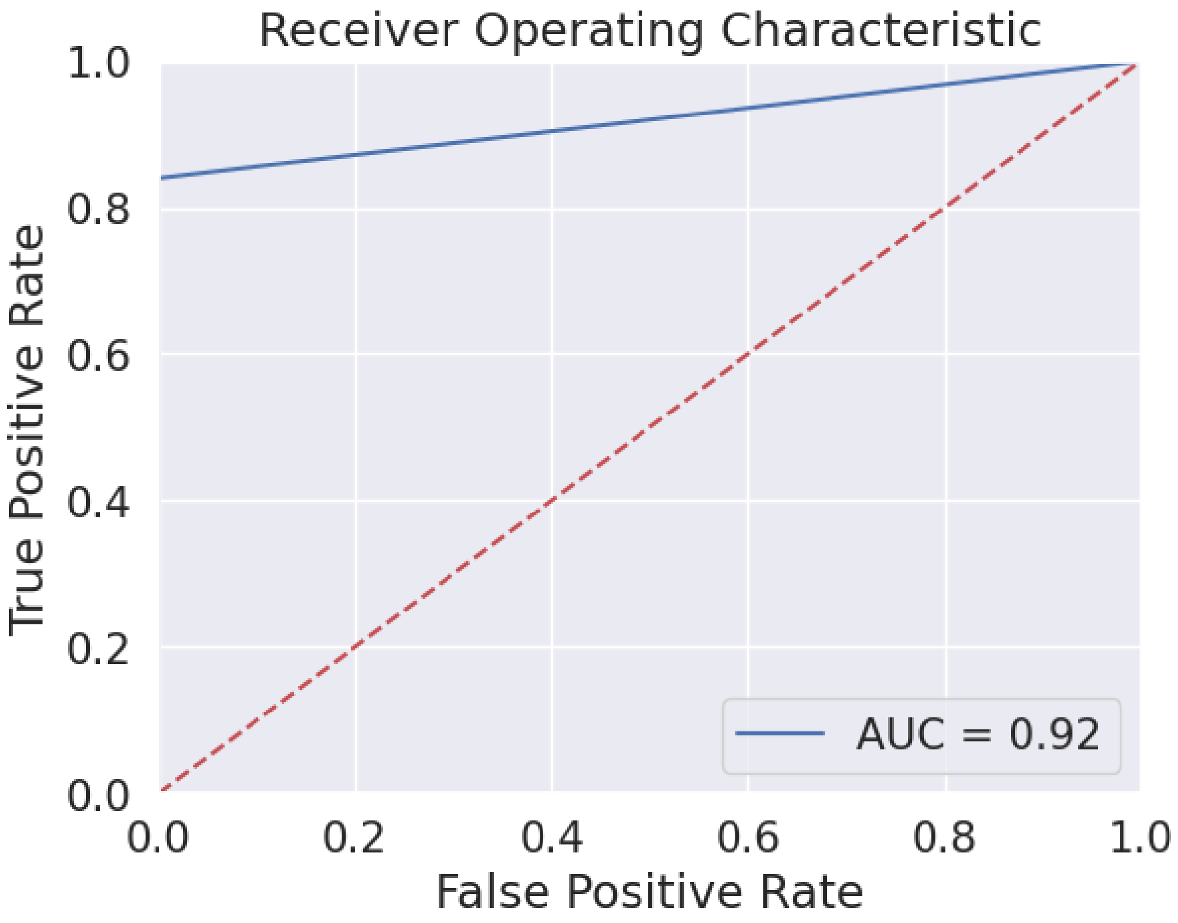
Support Vector Machine (SVM)

- Testing Accuracy: 97.85%
- Precision: Similar to the Random Forest, high for both classes.
- Recall: Identical to Random Forest for the negative class and slightly improved for the positive class at 84%.
- F1Score: Slightly improved compared to Random Forest, particularly for the positive class.



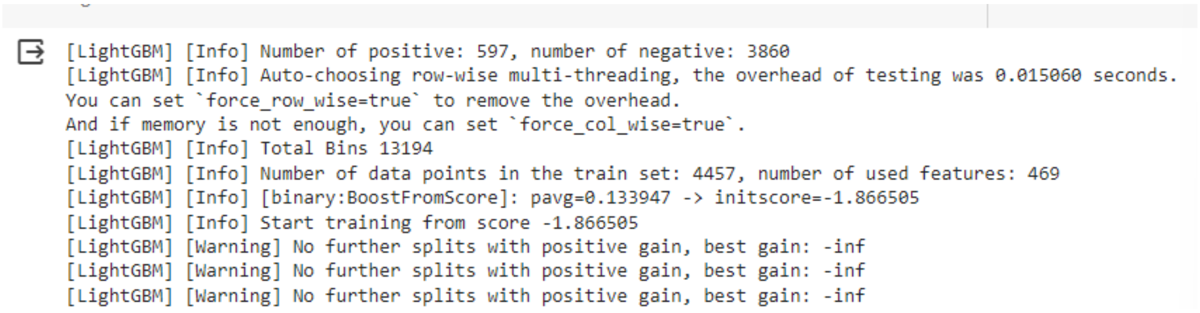
```
              precision    recall  f1-score   support

       False    0.97573   1.00000   0.98772       965
        True    1.00000   0.84000   0.91304       150

    accuracy                        0.97848      1115
   macro avg    0.98787   0.92000   0.95038      1115
weighted avg    0.97900   0.97848   0.97767      1115
```


965
0
24
126
0
1
0
1
800
600
400
200
0

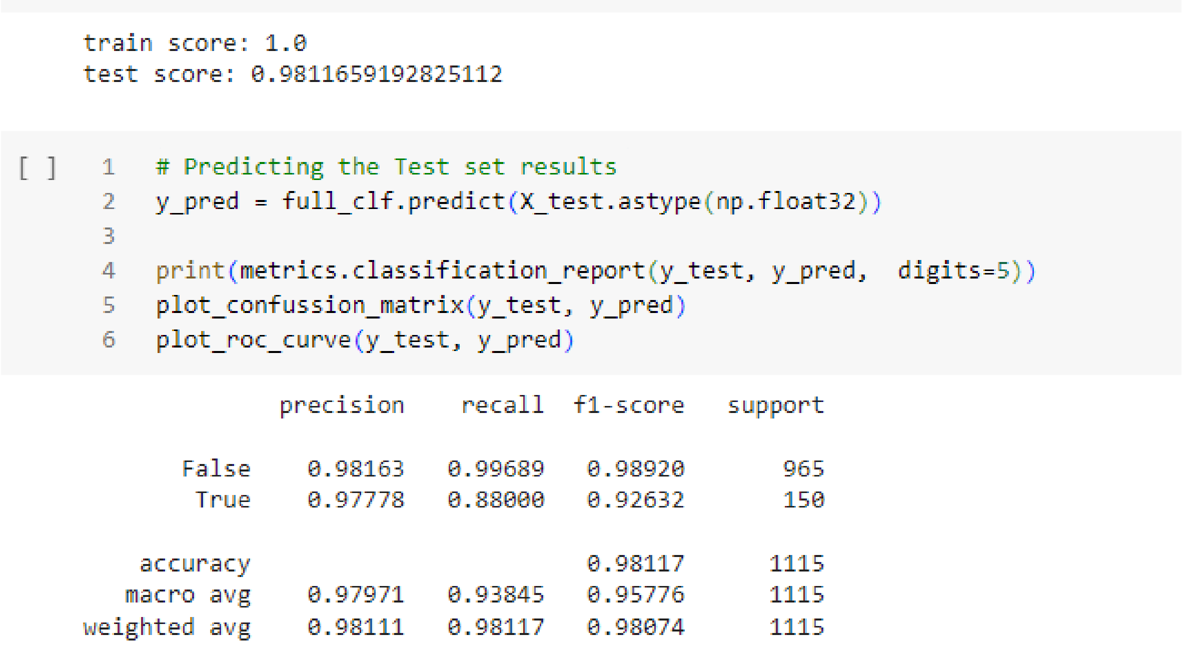


XGBoost Classifier

- Training Accuracy: 100%
- Testing Accuracy: 98.12%
- Precision: Marginally lower than SVM for the positive class, but still very high.
- Recall: Best performance among the models for the positive class at 88%.
- F1Score: Best overall balance between precision and recall, particularly for the positive class.



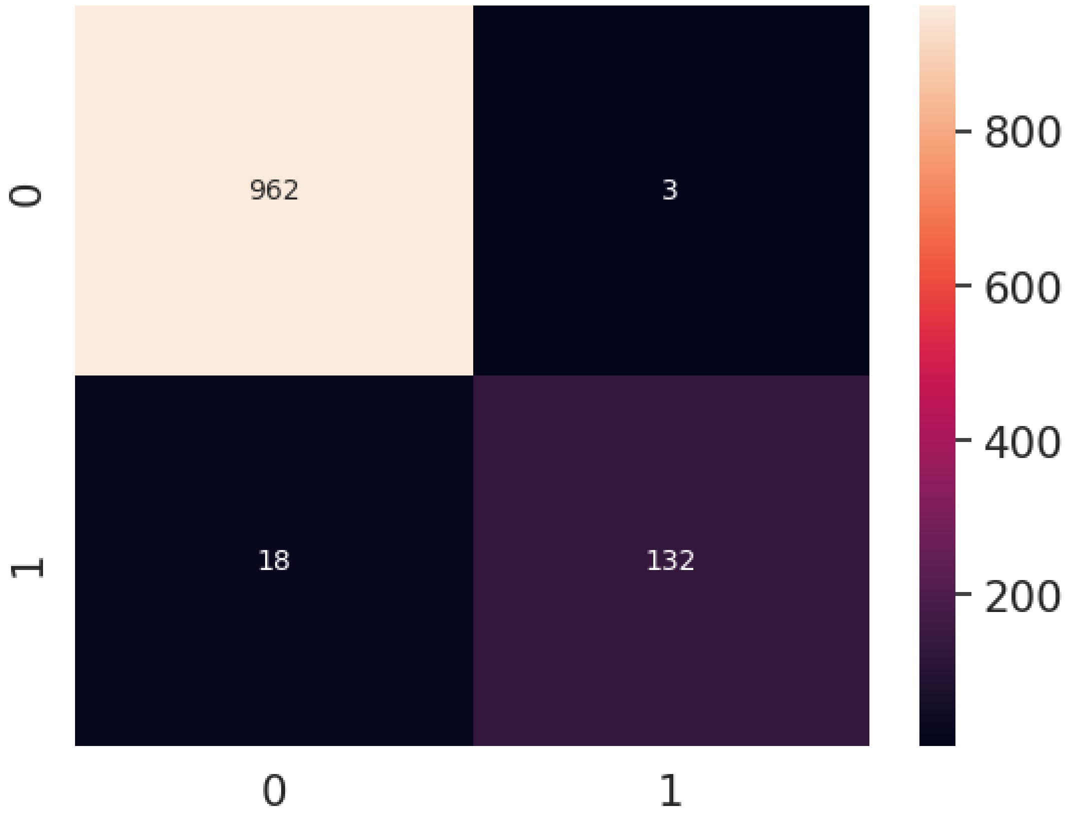
```
[LightGBM] [Info] Number of positive: 597, number of negative: 3860
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.015060 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 13194
[LightGBM] [Info] Number of data points in the train set: 4457, number of used features: 469
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.133947 -> initscore=-1.866505
[LightGBM] [Info] Start training from score -1.866505
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```

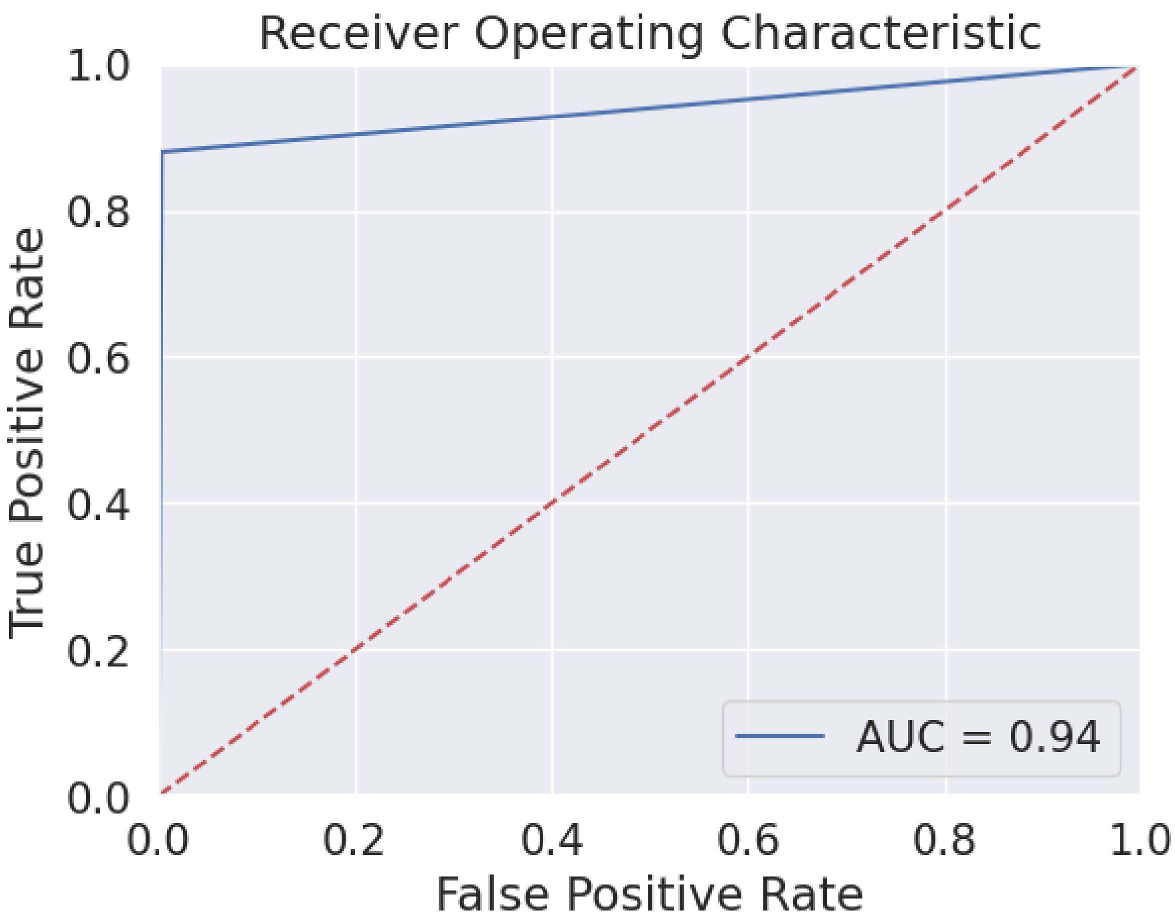


```
train score: 1.0
test score: 0.9811659192825112
```

```
# Predicting the Test set results
y_pred = full_clf.predict(X_test.astype(np.float32))

print(metrics.classification_report(y_test, y_pred,  digits=5))
plot_confussion_matrix(y_test, y_pred)
plot_roc_curve(y_test, y_pred)
```

```
              precision    recall  f1-score   support

       False    0.98163   0.99689   0.98920       965
        True    0.97778   0.88000   0.92632       150

    accuracy                        0.98117      1115
   macro avg    0.97971   0.93845   0.95776      1115
weighted avg    0.98111   0.98117   0.98074      1115
```





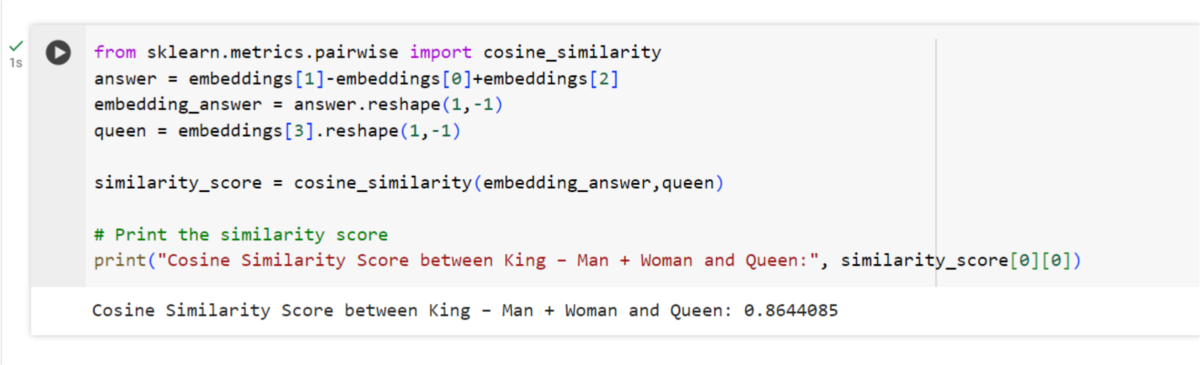
Analysis

- Accuracy: All three models demonstrated high accuracy, with XGBoost slightly leading at 98.12%.
- Precision and Recall: XGBoost had the best balance, with notably higher recall for the positive class without a substantial drop in precision. This indicates a better capability in identifying spam messages correctly.
- Robustness: Random Forest showed potential overfitting with perfect training accuracy, which might suggest less generalizability to unseen data compared to the other models.
- Computational Efficiency: SVM required hyperparameter tuning, which can be computationally intensive and timeconsuming, whereas Random Forest and LightGBM were relatively straightforward to implement and train.

The analysis indicates that XGBoost classifier provides the best balance between recall and precision, making it the most suitable model for spam detection in this scenario. It efficiently identifies more spam messages correctly without misclassifying many regular messages as spam. However, all models achieved commendable performance, showcasing the effectiveness of Word2Vec embeddings.

Q3. Design a task to compare the performance of GloVe embedding and BERT embedding at the sentence level.

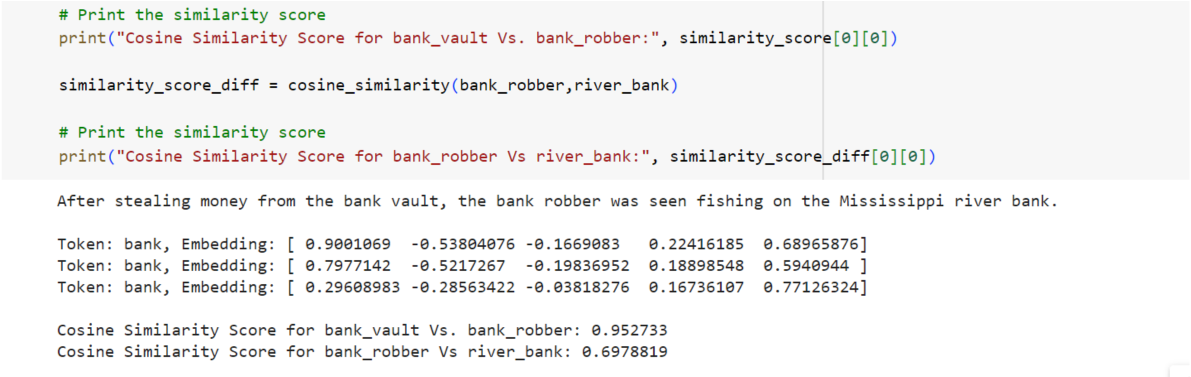
In the below image we can see how BERT word level embedding works. I created individual embedding for each word “King” “Man” “Queen” and “women”. When we used the embedding in formula “King – Man + Women” resulted embedding is 86% similar to the embedding of queen thus showing us the power of BERT to identify relationships.



```
from sklearn.metrics.pairwise import cosine_similarity
answer = embeddings[1]-embeddings[0]+embeddings[2]
embedding_answer = answer.reshape(1,-1)
queen = embeddings[3].reshape(1,-1)

similarity_score = cosine_similarity(embedding_answer,queen)

# Print the similarity score
print("Cosine Similarity Score between King - Man + Woman and Queen:", similarity_score[0][0])
```

```
Cosine Similarity Score between King - Man + Woman and Queen: 0.8644085
```

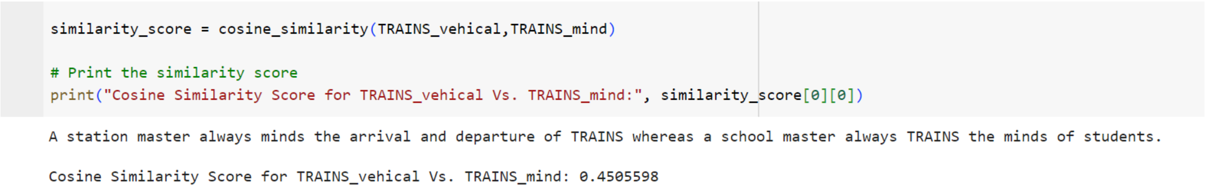
In the below example , we can see the BERT power to include the context into consideration while creating the embedding. In the below example bank has been used in two different context one as a financial institute another as a river bank. We can clearly see that BERT has created 3 different embedding for each bank and Bank vault and bank robber embedding have 95 % similarity score but in case of bank and river bank we only get 69 % similarity score thus telling us that its able to identify their contextual relationship.



```
# Print the similarity score
print("Cosine Similarity Score for bank_vault Vs. bank_robber:", similarity_score[0][0])

similarity_score_diff = cosine_similarity(bank_robber,river_bank)

# Print the similarity score
print("Cosine Similarity Score for bank_robber Vs river_bank:", similarity_score_diff[0][0])
```

```
After stealing money from the bank vault, the bank robber was seen fishing on the Mississippi river bank.

Token: bank, Embedding: [ 0.9001069  -0.53804076 -0.1669083   0.22416185  0.68965876]
Token: bank, Embedding: [ 0.7977142  -0.5217267  -0.19836952  0.18898548  0.5940944 ]
Token: bank, Embedding: [ 0.29608983 -0.28563422 -0.03818276  0.16736107  0.77126324]

Cosine Similarity Score for bank_vault Vs. bank_robber: 0.952733
Cosine Similarity Score for bank_robber Vs river_bank: 0.6978819
```

Similar to the above example I have taken one more example using train as a vehicle and train the mind as these train are no way related to each other. In case of BERT, it is able to identify that both train are different so, we get two different embedding with similarity score of 45% thus telling us the embeddings are no way closely related.

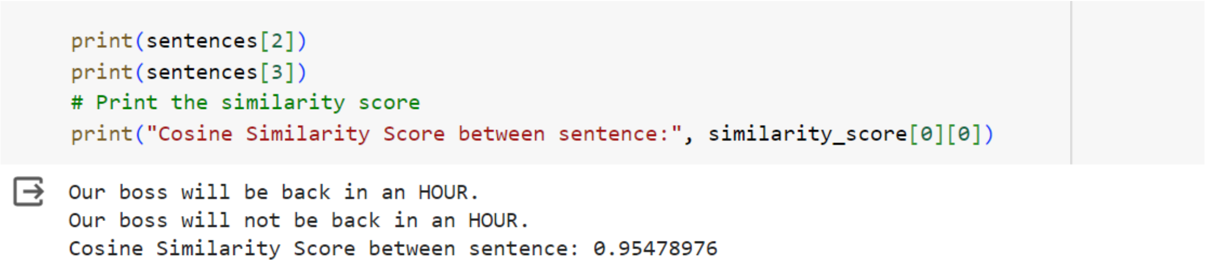


```
similarity_score = cosine_similarity(TRAINS_vehical,TRAINS_mind)

# Print the similarity score
print("Cosine Similarity Score for TRAINS_vehical Vs. TRAINS_mind:", similarity_score[0][0])
```

```
A station master always minds the arrival and departure of TRAINS whereas a school master always TRAINS the minds of students.

Cosine Similarity Score for TRAINS_vehical Vs. TRAINS_mind: 0.4505598
```

In the below example we are comparing BERT and GloVe embedding, I have taken two sentence, second one being negative of first sentence. In case of BERT, embedding created for both the sentence has cosign similarity of 95% but in case of GloVe its 99% thus demonstrating that BERT has a ability to understand the context of each word.

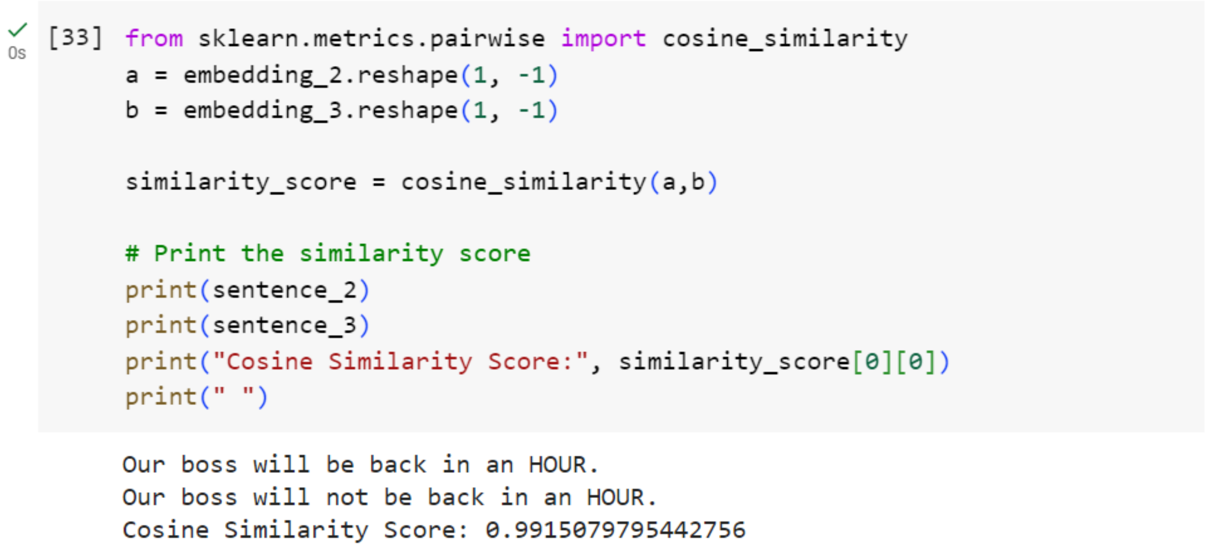
BERT:



```
print(sentences[2])
print(sentences[3])
# Print the similarity score
print("Cosine Similarity Score between sentence:", similarity_score[0][0])
```

```
Our boss will be back in an HOUR.
Our boss will not be back in an HOUR.
Cosine Similarity Score between sentence: 0.95478976
```

GloVe:



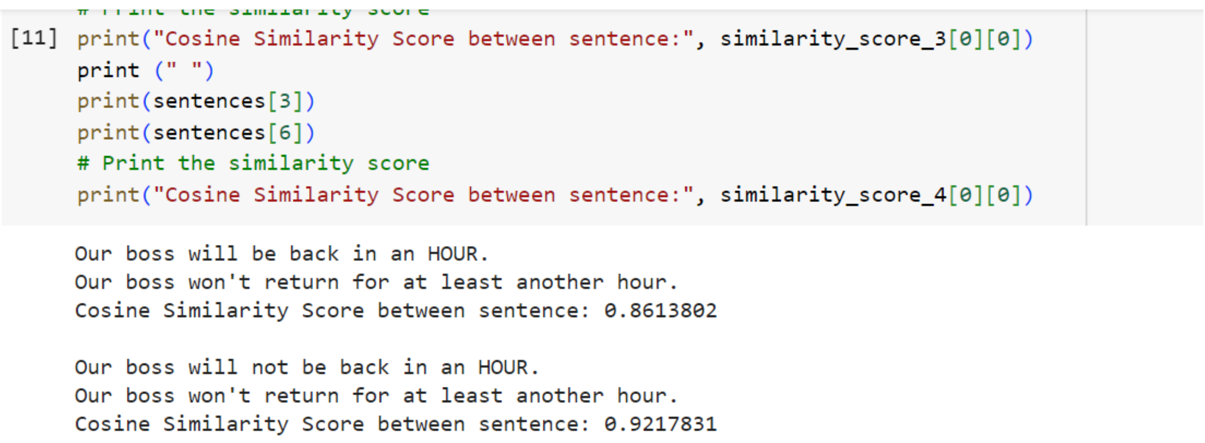
```
[33] from sklearn.metrics.pairwise import cosine_similarity
     a = embedding_2.reshape(1, -1)
     b = embedding_3.reshape(1, -1)

     similarity_score = cosine_similarity(a,b)

     # Print the similarity score
     print(sentence_2)
     print(sentence_3)
     print("Cosine Similarity Score:", similarity_score[0][0])
     print(" ")
```

```
Our boss will be back in an HOUR.
Our boss will not be back in an HOUR.
Cosine Similarity Score: 0.9915079795442756
```

In the below example , I have used an additional paraphrased sentence to see how BERT and GloVe performers. As we can see, two sentences “Our boss will be back in an HOUR.” “Our boss won't return for at least another hour. “ In case of BERT has a similarity of 86 % and in case of GloVe also we have 86% thus its has a similar performance.

In case 2: “Our boss will not be back in an HOUR.” ” Our boss won't return for at least another hour.” Bert embedding we have a similarity of 92% which is significantly Higer than GloVe which is 86%. Thus demonstrating the power of BERT to identify context and deeper meaning.

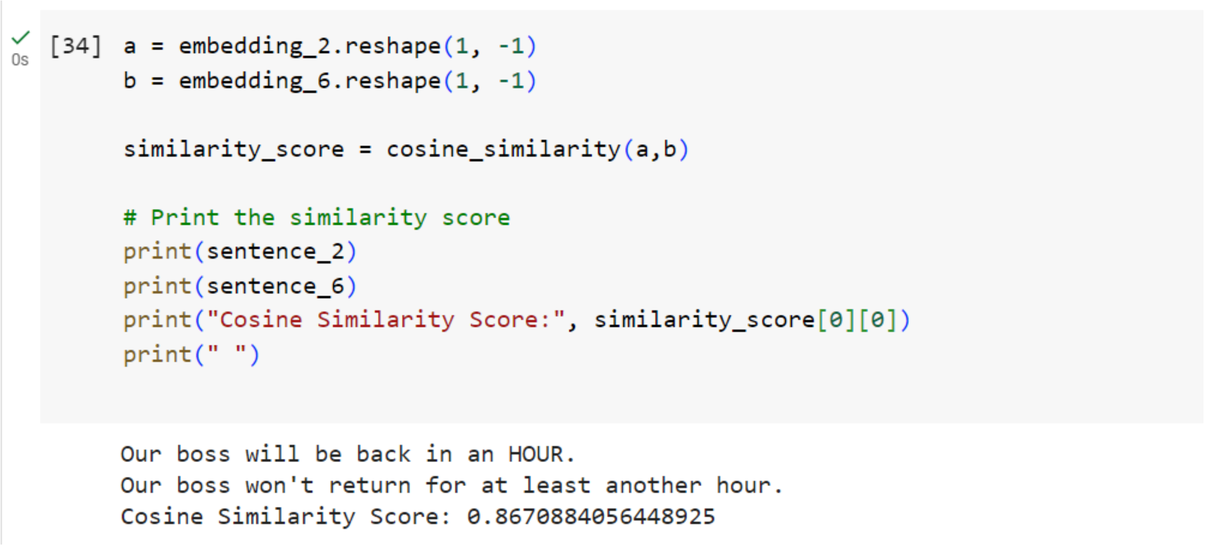
BERT:



```
# Print the similarity score
[11] print("Cosine Similarity Score between sentence:", similarity_score_3[0][0])
     print (" ")
     print(sentences[3])
     print(sentences[6])
     # Print the similarity score
     print("Cosine Similarity Score between sentence:", similarity_score_4[0][0])
```

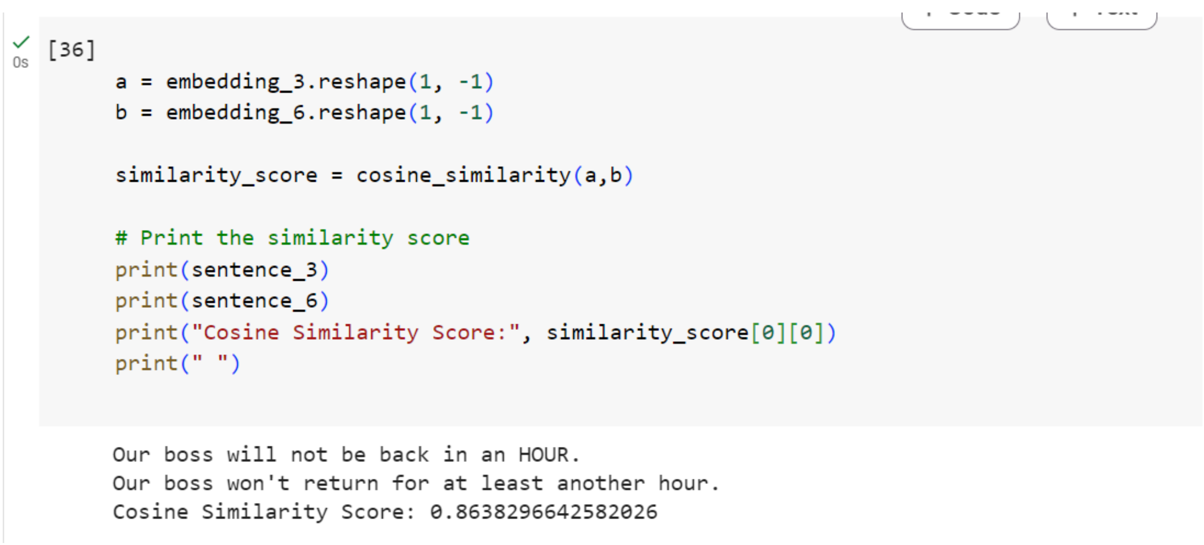
```
Our boss will be back in an HOUR.
Our boss won't return for at least another hour.
Cosine Similarity Score between sentence: 0.8613802

Our boss will not be back in an HOUR.
Our boss won't return for at least another hour.
Cosine Similarity Score between sentence: 0.9217831
```

GloVe:



```
[34] a = embedding_2.reshape(1, -1)
     b = embedding_6.reshape(1, -1)

     similarity_score = cosine_similarity(a,b)

     # Print the similarity score
     print(sentence_2)
     print(sentence_6)
     print("Cosine Similarity Score:", similarity_score[0][0])
     print(" ")
```

```
Our boss will be back in an HOUR.
Our boss won't return for at least another hour.
Cosine Similarity Score: 0.8670884056448925
```

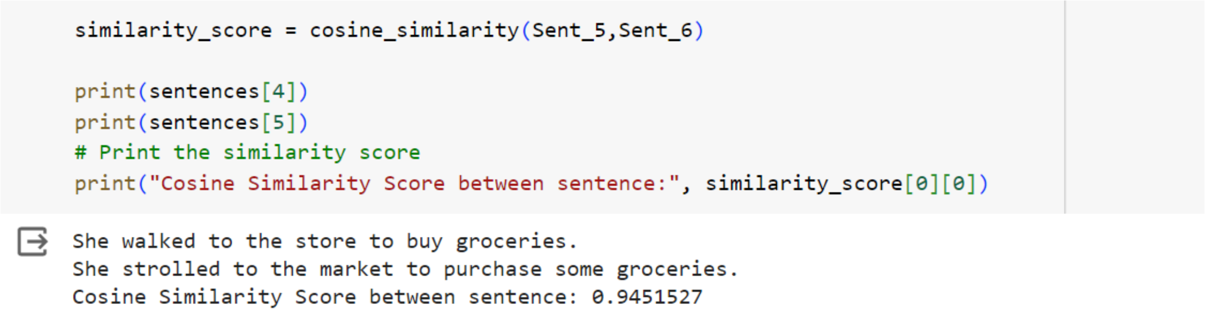


```
[36]
a = embedding_3.reshape(1, -1)
b = embedding_6.reshape(1, -1)

similarity_score = cosine_similarity(a,b)

# Print the similarity score
print(sentence_3)
print(sentence_6)
print("Cosine Similarity Score:", similarity_score[0][0])
print(" ")
```

```
Our boss will not be back in an HOUR.
Our boss won't return for at least another hour.
Cosine Similarity Score: 0.8638296642582026
```

In another example we have two sentence embedding comparison, second sentence is paraphrase of first sentence. In case of BERT embedding, the cosign similarity is 94% thus demonstrating BERT advance ability to understand context. In case of GloVe embedding, the cosign similarity is 90% which lower that the bert.

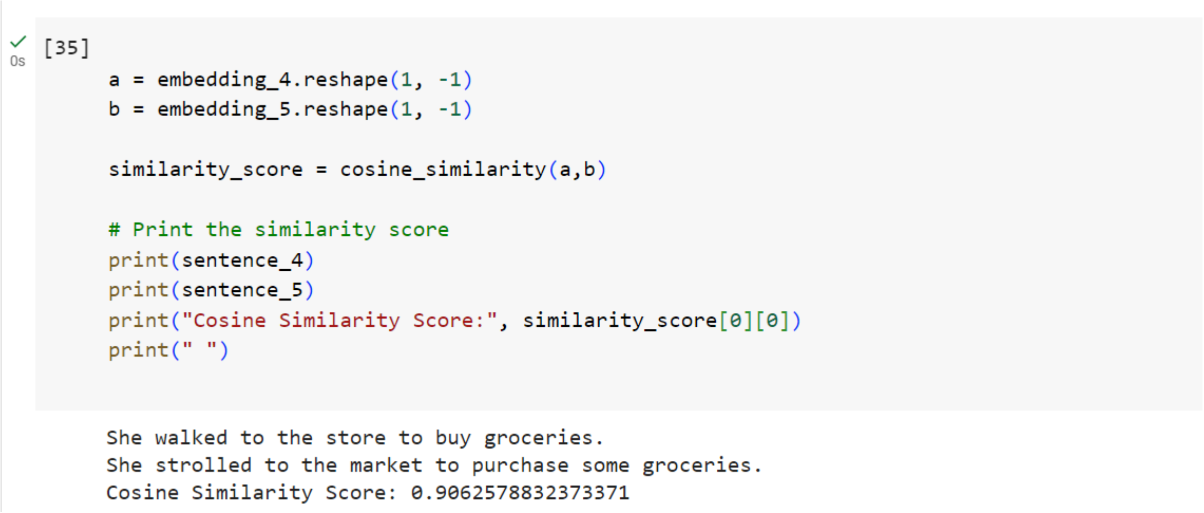
BERT:



```
similarity_score = cosine_similarity(Sent_5,Sent_6)

print(sentences[4])
print(sentences[5])
# Print the similarity score
print("Cosine Similarity Score between sentence:", similarity_score[0][0])
```

```
She walked to the store to buy groceries.
She strolled to the market to purchase some groceries.
Cosine Similarity Score between sentence: 0.9451527
```

GloVe:



```
a = embedding_4.reshape(1, -1)
b = embedding_5.reshape(1, -1)

similarity_score = cosine_similarity(a,b)

# Print the similarity score
print(sentence_4)
print(sentence_5)
print("Cosine Similarity Score:", similarity_score[0][0])
print(" ")
```

```
She walked to the store to buy groceries.
She strolled to the market to purchase some groceries.
Cosine Similarity Score: 0.9062578832373371
```

Summery:

Input:

"""Dougie Freedman is on the verge of agreeing a new two-year deal to remain at Nottingham Forest. Freedman has stabilised Forest since he replaced cult hero Stuart Pearce and the club's owners are pleased with the job he has done at the City Ground. Dougie Freedman is set to sign a new deal at Nottingham Forest . Freedman has impressed at the City Ground since replacing Stuart Pearce in February . They made an audacious attempt on the play-off places when Freedman replaced Pearce but have tailed off in recent weeks. That has not prevented Forest's ownership making moves to secure Freedman on a contract for the next two seasons."""

BERT:

Dougie Freedman is on the verge of agreeing a new two-year deal . he replaced cult hero Stuart Pearce at the city ground in february . the club's owners are pleased with the job he has done at the club .

Glove:

Freedman has stabilised Forest since he replaced cult hero Stuart Pearce and the club's owners are pleased with the job he has done at the City Ground.Dougie Freedman is set to sign a new deal at Nottingham Forest .That has not prevented Forest's ownership making moves to secure Freedman on a contract for the next two seasons.

In my opinion, both BERT and GloVe have provided concise summaries of the input paragraph, but with some differences in coverage. BERT's summary is very to the point and covers all the important aspects, while GloVe's summary also captures the essence but might miss a few points. Overall, both models have done a good job, with BERT being slightly more comprehensive.

Q4. In Assignment, you will propose your own NLP research project. Please describe two research ideas for your assignment 4. Please be as specific as you can.

Project - 1: Deep Dive on Roberta

Statement: Language model pretraining has led to significant performance gains but careful comparison between different approaches is challenging. Training is computationally expensive, often done on private datasets of different sizes, and, hyperparameter choices have significant impact on the final results. We will show a deeper study of BERT pretraining - on GLUE, SQAD benchmarks.

Dataset: WikiText-103 dataset

Project - 2: Amazon Comparison

Statement: Using two datasets, one with the top fiction books on Amazon and another of a collection of books in general. We wanted to see what insights we could pull from comparing datasets: could be a classification between a random book and a top fiction book, could be some topic modeling, and/or maybe predicting the title of a bestseller vs predictions of a random book.

Dataset: General Book Dataset / Amazon Bestsellers

Project - 3: Question Answering with a fine-tuned BERT

Statement: Develop a question answering system using a fine-tuned BERT model from the Hugging Face Transformers library on the CoQA dataset by Stanford NLP. The system should be able to understand and answer a series of interconnected questions based on text passages, leveraging BERT's bidirectional context understanding and transformers' capabilities in natural language processing. The goal is to demonstrate the effectiveness of pre-trained models in complex NLP tasks such as conversational question answering.

Dataset: Dataset

Source Links To Code
Our own Word2Vec (using gensim api)
Pre-Trained Word2Vec (FastText Model)
GloVe Pretrained Model
Semantic Distance Calculation & Visualization

Classification task using our own Word2Vec
Classification task using FastTest model
Classification task using Glove model
BERT embedding
Bert embedding and Glove at sentence level