```
# Numerical computing library for handling arrays and mathematical operations
import numpy as np

# Library for data manipulation and analysis (used for loading and processing datasets)
import pandas as pd

# Regular expressions library for text cleaning and preprocessing
import re

# OS module to interact with the operating system (file paths, directories, etc.)
import os

# Used to extract and manage compressed ZIP files
import zipfile

# Used to download datasets or files from the internet
import requests

# Library for plotting graphs and visualizations
import matplotlib.pyplot as plt

# Advanced visualization library built on matplotlib (used for confusion matrix heatmaps, etc.)
import seaborn as sns


# Used to split dataset into training and testing sets
from sklearn.model_selection import train_test_split

# Used to evaluate classification model performance
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix


# Converts text into integer sequences (tokenization)
from tensorflow.keras.preprocessing.text import Tokenizer

# Pads sequences to ensure equal input length
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Sequential model API to build neural networks layer by layer
from tensorflow.keras.models import Sequential

# Embedding → converts words to dense vectors
# Conv1D → 1D convolution layer for extracting local patterns
# GlobalMaxPooling1D → reduces feature maps to single value per filter
# Dense → fully connected layer
# Dropout → prevents overfitting
from tensorflow.keras.layers import Embedding, Conv1D, GlobalMaxPooling1D, Dense, Dropout

# Adam optimizer for training the neural network
from tensorflow.keras.optimizers import Adam
```

```
data = pd.read_csv("/content/SMSSpamCollection", sep='\t', names=['sentiment', 'review'])

print(data.head())
print(data.info())
print(data['sentiment'].value_counts())

# Check review length
data['review_length'] = data['review'].apply(len)
print(data['review_length'].describe())
```

```
  sentiment                                             review
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...
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5572 entries, 0 to 5571
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype 
---  ------     --------------  ----- 
 0   sentiment  5572 non-null   object
 1   review     5572 non-null   object
dtypes: object(2)
memory usage: 87.2+ KB
None
sentiment
ham     4825
spam     747
Name: count, dtype: int64
count    5572.000000
mean       80.489950
std        59.942907
min         2.000000
25%        36.000000
50%        62.000000
75%       122.000000
max       910.000000
Name: review_length, dtype: float64
```

```
def clean_text(text):
    """Converts text to lowercase and removes non-alphabetic characters."""
    text = text.lower()
    # Remove non-alphabetic characters and replace them with a space
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    return text

data['clean_review'] = data['review'].apply(clean_text)
```

```
max_words = 20000  # Maximum number of words to keep, based on word frequency
max_len = 200      # Maximum length of each review sequence

# Initialize tokenizer with the specified maximum number of words
tokenizer = Tokenizer(num_words=max_words)
# Fit the tokenizer on the clean reviews to build the vocabulary
tokenizer.fit_on_texts(data['clean_review'])

# Convert text reviews to sequences of integers
sequences = tokenizer.texts_to_sequences(data['clean_review'])
# Pad sequences to ensure uniform length for all reviews
X = pad_sequences(sequences, maxlen=max_len)

# Map sentiment labels to numerical values (ham: 0, spam: 1)
y = data['sentiment'].map({'positive':1, 'negative':0})
```

```
glove_url = "http://nlp.stanford.edu/data/glove.6B.zip"
zip_file = "glove.6B.zip"

# Download
if not os.path.exists(zip_file):
    print("Downloading GloVe embeddings...")
    response = requests.get(glove_url)
    with open(zip_file, "wb") as f:
        f.write(response.content)
    print("Download complete!")

# Extract
if not os.path.exists("glove.6B.100d.txt"):
    print("Extracting files...")
    with zipfile.ZipFile(zip_file, 'r') as zip_ref:
        zip_ref.extractall(".")
    print("Extraction complete!")
```

```
Downloading GloVe embeddings...
Download complete!
Extracting files...
Extraction complete!
```

```
embeddings_index = {}
with open('glove.6B.100d.txt', encoding='utf8') as f:
    for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs

print(f"Found {len(embeddings_index)} word vectors in glove.6B.100d.txt.")
```

```
Found 400000 word vectors in glove.6B.100d.txt.
```

```
word_index = tokenizer.word_index # Get the word-to-index mapping from the tokenizer
embedding_dim = 100               # Dimension of the GloVe embeddings

# Determine the number of words to consider for the embedding matrix
num_words = min(max_words, len(word_index) + 1)
# Initialize the embedding matrix with zeros
embedding_matrix = np.zeros((num_words, embedding_dim))

# Populate the embedding matrix with GloVe vectors for words present in our vocabulary
for word, i in word_index.items():
    if i < num_words:
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            # Words not found in embedding index will be all-zeros.
            embedding_matrix[i] = embedding_vector

print(f"Shape of embedding matrix: {embedding_matrix.shape}")
```

```
Shape of embedding matrix: (8630, 100)
```

```
y = data['sentiment'].map({'ham': 0, 'spam': 1})

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}")
```

```
X_train shape: (4457, 200)
X_test shape: (1115, 200)
y_train shape: (4457,)
y_test shape: (1115,)
```

```
model = Sequential()
# Embedding layer using pre-trained GloVe weights, set to non-trainable
model.add(Embedding(num_words, embedding_dim, weights=[embedding_matrix], trainable=False))
# 1D Convolutional layer for feature extraction
model.add(Conv1D(128, 5, activation='relu'))
# Global Max Pooling layer to reduce dimensionality
model.add(GlobalMaxPooling1D())
# Dense layer with ReLU activation
model.add(Dense(64, activation='relu'))
# Dropout layer for regularization to prevent overfitting
model.add(Dropout(0.5))
# Output Dense layer with sigmoid activation for binary classification
model.add(Dense(1, activation='sigmoid'))

model.summary()
```

```
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ embedding (Embedding)            │ ?                      │       863,000 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ conv1d (Conv1D)                  │ ?                      │   0 (unbuilt) │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ global_max_pooling1d             │ ?                      │             0 │
│ (GlobalMaxPooling1D)             │                        │               │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                    │ ?                      │   0 (unbuilt) │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)                │ ?                      │             0 │
├──────────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)                  │ ?                      │   0 (unbuilt) │
└──────────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 863,000 (3.29 MB)
 Trainable params: 0 (0.00 B)
 Non-trainable params: 863,000 (3.29 MB)
```

```python
# Define the Adam optimizer with a specified learning rate
optimizer = Adam(learning_rate=0.001)
# Compile the model with binary crossentropy loss, Adam optimizer, and accuracy metric
model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy'])
```

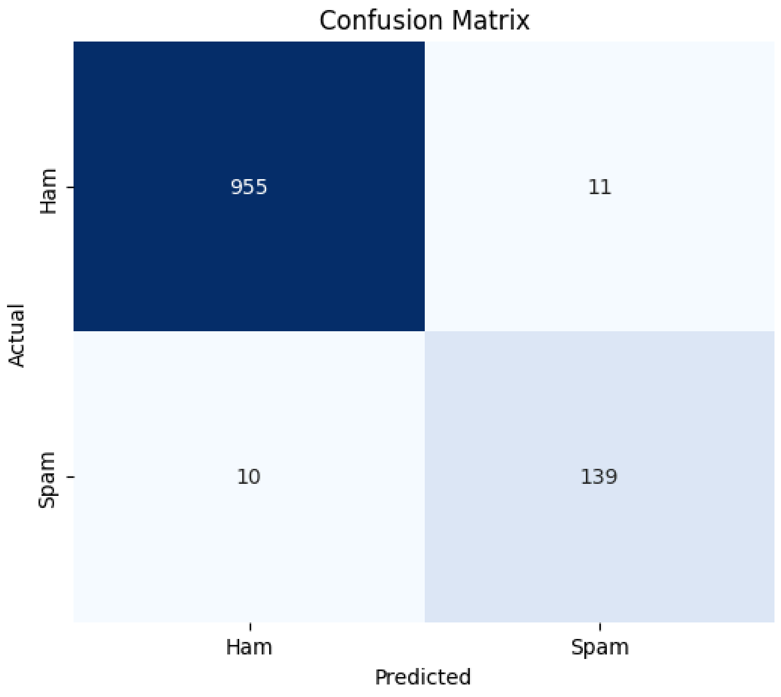
```python
# Train the model using the training data and validate with the test data
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))
```

```
Epoch 1/10
140/140 ━━━━━━━━━━━━━━━━━━━━ 12s 71ms/step - accuracy: 0.8703 - loss: 0.3511 - val_accuracy: 0.9740 - val_loss:
Epoch 2/10
140/140 ━━━━━━━━━━━━━━━━━━━━ 9s 62ms/step - accuracy: 0.9746 - loss: 0.0864 - val_accuracy: 0.9722 - val_loss: (
Epoch 3/10
140/140 ━━━━━━━━━━━━━━━━━━━━ 14s 98ms/step - accuracy: 0.9896 - loss: 0.0436 - val_accuracy: 0.9812 - val_loss:
Epoch 4/10
140/140 ━━━━━━━━━━━━━━━━━━━━ 14s 53ms/step - accuracy: 0.9971 - loss: 0.0167 - val_accuracy: 0.9803 - val_loss:
Epoch 5/10
140/140 ━━━━━━━━━━━━━━━━━━━━ 9s 46ms/step - accuracy: 0.9977 - loss: 0.0102 - val_accuracy: 0.9848 - val_loss: (
Epoch 6/10
140/140 ━━━━━━━━━━━━━━━━━━━━ 11s 55ms/step - accuracy: 0.9981 - loss: 0.0080 - val_accuracy: 0.9821 - val_loss:
Epoch 7/10
140/140 ━━━━━━━━━━━━━━━━━━━━ 12s 70ms/step - accuracy: 0.9996 - loss: 0.0033 - val_accuracy: 0.9821 - val_loss:
Epoch 8/10
140/140 ━━━━━━━━━━━━━━━━━━━━ 6s 46ms/step - accuracy: 0.9999 - loss: 0.0021 - val_accuracy: 0.9785 - val_loss: (
Epoch 9/10
140/140 ━━━━━━━━━━━━━━━━━━━━ 11s 53ms/step - accuracy: 0.9995 - loss: 0.0042 - val_accuracy: 0.9776 - val_loss:
Epoch 10/10
140/140 ━━━━━━━━━━━━━━━━━━━━ 8s 56ms/step - accuracy: 0.9979 - loss: 0.0042 - val_accuracy: 0.9812 - val_loss: (
```

```python
# Predict probabilities for the test set
y_pred_probs = model.predict(X_test)
# Convert probabilities to binary class labels (0 or 1)
y_pred = (y_pred_probs > 0.5).astype(int)

# Calculate evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

# Print the calculated metrics
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-Score: {f1:.4f}")

# Plot the confusion matrix
plt.figure(figsize=(6, 5))
```

```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
            xticklabels=['Ham', 'Spam'], yticklabels=['Ham', 'Spam'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

```
35/35 ━━━━━━━━━━━━━━━━━━━━ 1s 35ms/step
Accuracy: 0.9812
Precision: 0.9267
Recall: 0.9329
F1-Score: 0.9298
```


Confusion Matrix
Ham
Spam
Actual
Ham
Spam
Predicted