Group No 169

Group Member Names:

1. Krithika Madhavan 2024AA05421
2. Payel Karmekar 2024AA05423
3. Rahul Agarwal 2024AA05676
4. Yarragondla Rugmangadha Reddy 2024AA05435

1. Import the required libraries

In [ ]:
```
##---------Type the code below this line------------------##
import tensorflow as tf
from tensorflow.keras import models
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
```

2. Data Acquisition -- Score: 0.5 Mark

1. Selected Problem is *** IMDB Review Dataset ***
2. Downloaded using Kera's Dataset

In [ ]:
```
# Load the IMDB Dataset using Keras
imdb = tf.keras.datasets.imdb
```

2.1 Code for converting the above downloaded data into a form suitable for DL

In [ ]:
```
##---------Type the code below this line------------------##

# num_words=10000 means we only keep the top 10,000 most frequently occurring words
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=10000)

# Pad the sequences to ensure uniform input size for DNN models
max_len = 500  # we'll pad/truncate reviews to 500 words
X_train_padded = pad_sequences(X_train, maxlen=max_len)
X_test_padded = pad_sequences(X_test, maxlen=max_len)
```

2.1 Write your observations from the above.

1. Size of the dataset
2. What type of data attributes are there?
3. What are you classifying?
4. Plot the distribution of the categories of the target / label.

In [ ]:
```
#Size of the dataset - 50000
print(f"Training samples: {len(X_train_padded)}")
print(f"Testing samples: {len(X_test_padded)}")
```

```
Training samples: 25000
Testing samples: 25000
```

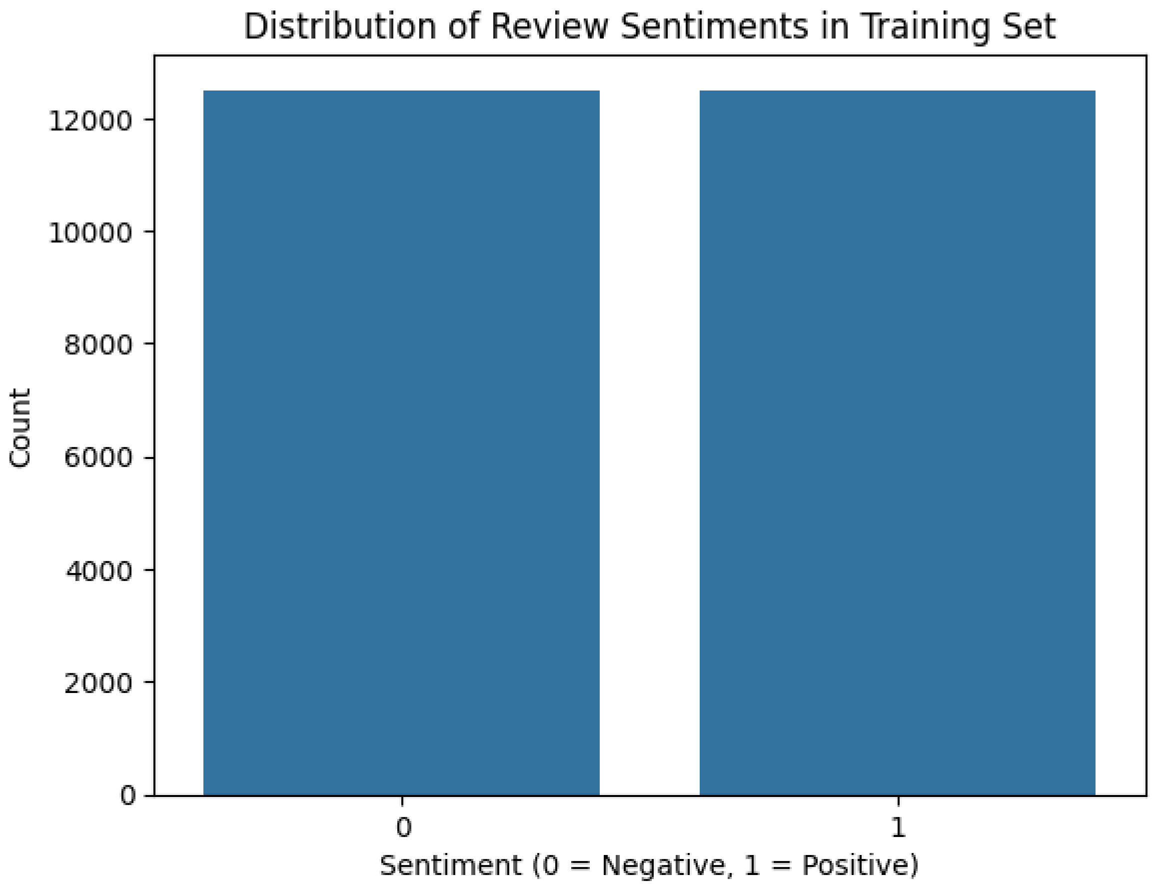
In [ ]:
```
#What type of data attributes are there?
print("Sample padded review:", X_train_padded[0])
print("Label:", y_train[0])
print("Data type of a sample padded review:", type(X_train_padded[0][0]))
```

```
Sample padded review: [   0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    1   14   22   16   43  530  973 1622 1385   65  458 4468
   66 3941    4  173   36  256    5   25  100   43  838  112   50  670
    2    9   35  480  284    5  150    4  172  112  167    2  336  385
   39    4  172 4536 1111   17  546   38   13  447    4  192   50   16
    6  147 2025   19   14   22    4 1920 4613  469    4   22   71   87
   12   16   43  530   38   76   15   13 1247    4   22   17  515   17
   12   16  626   18    2    5   62  386   12    8  316    8  106    5
    4 2223 5244   16  480   66 3785   33    4  130   12   16   38  619
    5   25  124   51   36  135   48   25 1415   33    6   22   12  215
   28   77   52    5   14  407   16   82    2    8    4  107  117 5952
   15  256    4    2    7 3766    5  723   36   71   43  530  476   26
  400  317   46    7    4    2 1029   13  104   88    4  381   15  297
   98   32 2071   56   26  141    6  194 7486   18    4  226   22   21
  134  476   26  480    5  144   30 5535   18   51   36   28  224   92
   25  104    4  226   65   16   38 1334   88   12   16  283    5   16
 4472  113  103   32   15   16 5345   19  178   32]
Label: 1
Data type of a sample padded review: <class 'numpy.int32'>
```

What are you classifying?

- We are classifying movie reviews as **positive (1)** or **negative (0)**.
- It is a **binary classification problem**.

In [ ]:
```
#Plot the distribution of the categories of the target / label.
sns.countplot(x=y_train)
plt.title("Distribution of Review Sentiments in Training Set")
plt.xlabel("Sentiment (0 = Negative, 1 = Positive)")
plt.ylabel("Count")
plt.show()
```



3. Data Preparation -- Score: 1 Mark

Perform the data prepracessing that is required for the data that you have downloaded.

This stage depends on the dataset that is used.

3.1 Apply pre-processing techiniques

As the Keras data is partially preprocessed, below techniques are applied:

1. Duplicate data removal

In [ ]:
```
#to remove duplicate data
# Convert lists to tuples to make them hashable
unique_train_indices = np.unique([tuple(x) for x in X_train_padded], axis=0)

print("Original training size:", len(X_train_padded))
print("Unique training samples:", len(unique_train_indices))
```

```
Original training size: 25000
Unique training samples: 24901
```

In [ ]:
```
import pandas as pd

# Convert padded sequences and labels to DataFrame for easy deduplication
df_train = pd.DataFrame({
    'review': [tuple(x) for x in X_train_padded],
    'label': y_train
})

# Drop duplicates based on the 'review' column
df_train_unique = df_train.drop_duplicates(subset='review')

# Extract the cleaned data
X_train_padded_clean = np.array([list(x) for x in df_train_unique['review']])
y_train_clean = np.array(df_train_unique['label'])

# Print new shape
print("New training size after removing duplicates:", len(X_train_padded_clean))
```

```
New training size after removing duplicates: 24901
```

In [ ]:
```
#Remove data inconsistencies
print("Unique labels:", np.unique(y_train))
```

```
Unique labels: [0 1]
```

Unique labels: [0 1] Meaning the labels are consistent.

In [ ]:
```
# No categorical features - no encoding needed
# No text to lemmatize/stem as it's pre-processed to integers

# Normalize: Optional, but generally not needed for embedding/vector inputs
```

3.2 Identify the target variables.

- Separated the data front the target such that the dataset is in the form of (X,y) or (Features, Label)
- Performed the One-hot encoding and now the Lable 'Y' is in the form of 1., 0.

In [ ]:
```
##---------Type the code below this line------------------##
# Define max review length (truncate or pad to this length)
max_length = 200

# Pad sequences with zeros (post-padding is common)
x_train_padded = pad_sequences(X_train, maxlen=max_length, padding='post', truncating='post')
x_test_padded = pad_sequences(X_test, maxlen=max_length, padding='post', truncating='post')

y_train_onehot = to_categorical(y_train)
y_test_onehot = to_categorical(y_test)

print(y_train_onehot[:5])
```

```
[[0. 1.]
 [1. 0.]
 [1. 0.]
 [0. 1.]
 [1. 0.]]
```

3.3 Split the data into training set and testing set

In [ ]:
```
##---------Type the code below this line------------------##

# Import train_test_split
from sklearn.model_selection import train_test_split

# Perform an 80-20 split manually
X_train, X_test, y_train, y_test = train_test_split(
    X_train_padded_clean, y_train_clean, test_size=0.2, random_state=42
)

print("Training set size:", X_train.shape)
print("Testing set size:", X_test.shape)
```

```
Training set size: (19920, 500)
Testing set size: (4981, 500)
```

3.4 Preprocessing report

Mention the method adopted and justify why the method was used

- to remove duplicate data, if present
- to impute or remove missing data, if present
- to remove data inconsistencies, if present
- to encode categorical data
- the normalization technique used

If the any of the above are not present, then also add in the report below.

Report the size of the training dataset and testing dataset

Details:

1. Duplicate Data

- **Method:** Converted each padded sequence into a tuple and used `pandas.DataFrame.drop_duplicates()` to identify and remove duplicates.
- **Justification:** Duplicate reviews could bias the model and lead to overfitting. Removing them improves data quality and model generalization.

2. Missing Data

- **Method:** Used `numpy` and `pandas` to check for any missing (`NaN`) values.
- **Result:** No missing values were found in either features or labels.
- **Justification:** No imputation/removal was needed as the dataset is already clean.

3. Data Inconsistencies

- **Method:** Verified the label distribution using `np.unique()` to ensure all labels are either 0 or 1.
- **Result:** All labels were valid. No inconsistencies found.
- **Justification:** Ensures binary classification targets are intact and valid.

4. Categorical Data Encoding

- **Method:** Not applicable.
- **Justification:** The dataset uses integer-based word indices, and target labels are already binary-encoded (0 = negative, 1 = positive).

5. Normalization

- **Method:** Not applied.
- **Justification:** Word indices are used for embedding layers and don't require normalization. If TF-IDF or other numeric features were used, normalization would be appropriate.

Summary

The IMDB dataset provided by Keras is preprocessed and ready for deep learning tasks. Minimal preprocessing was needed beyond deduplication and sequence padding. By default, Keras dataset was splitted into 50-50 ratio, but modified it to 80-20 to get better Accuracy

4. Deep Neural Network Architecture - Score: Marks

4.1 Design the architecture that you will be using

- Sequential Model Building with Activation for each layer.
- Add dense layers, specifying the number of units in each layer and the activation function used in the layer.
- Use Relu Activation function in each hidden layer
- Use Sigmoid / softmax Activation function in the output layer as required

DO NOT USE CNN OR RNN.

In [ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense

# Parameters
vocab_size = 10000
embedding_dim = 32
input_length = 500

model = Sequential()

# Updated input_shape format
model.add(Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_shape=(input_length,)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Model summary
model.summary()
```

```
/opt/miniconda3/lib/python3.12/site-packages/keras/src/layers/core/embedding.py:100: UserWarning: Do not pass an `in
put_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as t
he first layer in the model instead.
  super().__init__(**kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 500, 32)	320,000
flatten (Flatten)	(None, 16000)	0
dense (Dense)	(None, 128)	2,048,128
dense_1 (Dense)	(None, 64)	8,256
dense_2 (Dense)	(None, 1)	65

Total params: 2,376,449 (9.07 MB)
Trainable params: 2,376,449 (9.07 MB)
Non-trainable params: 0 (0.00 B)

4.2 DNN Report

Report the following and provide justification for the same.

- Number of layers
- Number of units in each layer
- Total number of trainable parameters

Number of Layers: 5 layers total - 1 Embedding, 1 Flatten, 2 Dense hidden layers, 1 Dense output layer Chosen to keep the architecture simple, efficient and interpretable.

Units in Each Layer : ReLU helps in hidden layers to avoid vanishing gradients; Sigmoid outputs probability for binary sentiment classification. Embedding: 500 input length → 32-dim vector per word, Flatten: Converts (500×32) to 1D → 16,000 features, Dense 1: 128 units, ReLU, Dense 2: 64 units, ReLU, Output: 1 unit, Sigmoid (for binary classification)

Total Trainable Parameters : 2,376,449 parameters Mostly from Flatten → Dense layer connection (16000 × 128) The number is acceptable for a feedforward network on padded text input. Efficient yet powerful enough to learn sentiment features.

5. Training the model - Score: 1 Mark

5.1 Configure the training

Configure the model for training, by using appropriate optimizers and regularizations

Compile with categorical CE loss and metric accuracy.

In [ ]:
```
##---------Type the code below this line------------------##
from tensorflow.keras.utils import to_categorical

# Assuming y_train and y_test are your original binary labels (0 or 1)
y_train_cat = to_categorical(y_train, num_classes=2)
y_test_cat = to_categorical(y_test, num_classes=2)
```

In [ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, Dropout
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.regularizers import l2

# Model with dropout and L2 regularization
model = Sequential()
model.add(Embedding(input_dim=10000, output_dim=32, input_shape=(500,)))
model.add(Flatten())
model.add(Dense(128, activation='relu', kernel_regularizer=l2(0.001)))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu', kernel_regularizer=l2(0.001)))
model.add(Dropout(0.3))
model.add(Dense(2, activation='softmax'))  # Softmax for categorical CE

# Compile with SGD optimizer
model.compile(
    optimizer=SGD(learning_rate=0.01, momentum=0.9),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

```
model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 500, 32)	320,000
flatten_1 (Flatten)	(None, 16000)	0
dense_3 (Dense)	(None, 128)	2,048,128
dropout (Dropout)	(None, 128)	0
dense_4 (Dense)	(None, 64)	8,256
dropout_1 (Dropout)	(None, 64)	0
dense_5 (Dense)	(None, 2)	130

Total params: 2,376,514 (9.07 MB)

Trainable params: 2,376,514 (9.07 MB)

Non-trainable params: 0 (0.00 B)

5.2 Train the model

Train Model with cross validation, with total time taken shown for 20 epochs.

Use SGD.

In [ ]:
```
##---------Type the code below this line------------------##
from sklearn.model_selection import KFold
import numpy as np
import time

# Set up K-fold cross-validation
kfold = KFold(n_splits=5, shuffle=True, random_state=42)
```

```python
epochs = 20
batch_size = 128

fold = 1
start_time = time.time()

for train_idx, val_idx in kfold.split(X_train):
    print(f"\n--- Fold {fold} ---")

    # Create new model for each fold
    model = Sequential()
    model.add(Embedding(input_dim=10000, output_dim=32, input_shape=(500,)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_regularizer=l2(0.001)))
    model.add(Dropout(0.5))
    model.add(Dense(64, activation='relu', kernel_regularizer=l2(0.001)))
    model.add(Dropout(0.3))
    model.add(Dense(2, activation='softmax'))

    model.compile(
        optimizer=SGD(learning_rate=0.01, momentum=0.9),
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )

    # Split data
    X_tr, X_val = X_train[train_idx], X_train[val_idx]
    y_tr, y_val = y_train_cat[train_idx], y_train_cat[val_idx]

    # Train
    model.fit(X_tr, y_tr, epochs=epochs, batch_size=batch_size, validation_data=(X_val, y_val), verbose=1)

    fold += 1

end_time = time.time()
total_time = end_time - start_time

print(f"\n Total training time for {epochs} epochs across 5 folds: {total_time:.2f} seconds")
```

```
--- Fold 1 ---
Epoch 1/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.5057 - loss: 1.0290 - val_accuracy: 0.5000 - val_loss: 1.017
3
Epoch 2/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.5081 - loss: 1.0127 - val_accuracy: 0.5100 - val_loss: 1.000
6
Epoch 3/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.5152 - loss: 0.9963 - val_accuracy: 0.5105 - val_loss: 0.985
5
Epoch 4/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.5294 - loss: 0.9806 - val_accuracy: 0.5168 - val_loss: 0.970
6
Epoch 5/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.5373 - loss: 0.9651 - val_accuracy: 0.5356 - val_loss: 0.955
5
Epoch 6/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 18ms/step - accuracy: 0.5478 - loss: 0.9489 - val_accuracy: 0.5319 - val_loss: 0.941
7
Epoch 7/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.5731 - loss: 0.9317 - val_accuracy: 0.6150 - val_loss: 0.917
1
Epoch 8/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.6227 - loss: 0.9024 - val_accuracy: 0.6423 - val_loss: 0.878
8
Epoch 9/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.6908 - loss: 0.8457 - val_accuracy: 0.7113 - val_loss: 0.800
3
Epoch 10/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.7466 - loss: 0.7562 - val_accuracy: 0.7610 - val_loss: 0.718
6
Epoch 11/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.8131 - loss: 0.6432 - val_accuracy: 0.7748 - val_loss: 0.670
9
Epoch 12/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.8692 - loss: 0.5359 - val_accuracy: 0.7937 - val_loss: 0.652
3
Epoch 13/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.8928 - loss: 0.4802 - val_accuracy: 0.8062 - val_loss: 0.628
9
Epoch 14/20
```

```
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9281 - loss: 0.4105 - val_accuracy: 0.8060 - val_loss: 0.648
8
Epoch 15/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9422 - loss: 0.3635 - val_accuracy: 0.8032 - val_loss: 0.680
0
Epoch 16/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9601 - loss: 0.3133 - val_accuracy: 0.8042 - val_loss: 0.704
9
Epoch 17/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9722 - loss: 0.2784 - val_accuracy: 0.7866 - val_loss: 0.722
2
Epoch 18/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9743 - loss: 0.2689 - val_accuracy: 0.7615 - val_loss: 0.881
7
Epoch 19/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9617 - loss: 0.2868 - val_accuracy: 0.7904 - val_loss: 0.813
4
Epoch 20/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9879 - loss: 0.2192 - val_accuracy: 0.7871 - val_loss: 0.847
4

--- Fold 2 ---
Epoch 1/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.4991 - loss: 1.0290 - val_accuracy: 0.5171 - val_loss: 1.015
8
Epoch 2/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.5179 - loss: 1.0117 - val_accuracy: 0.5070 - val_loss: 1.000
2
Epoch 3/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.5172 - loss: 0.9959 - val_accuracy: 0.5070 - val_loss: 0.985
1
Epoch 4/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.5419 - loss: 0.9782 - val_accuracy: 0.5489 - val_loss: 0.967
3
Epoch 5/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.5585 - loss: 0.9596 - val_accuracy: 0.5763 - val_loss: 0.948
7
Epoch 6/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.6146 - loss: 0.9332 - val_accuracy: 0.6315 - val_loss: 0.913
9
Epoch 7/20
```

```
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 17ms/step - accuracy: 0.6584 - loss: 0.8830 - val_accuracy: 0.6604 - val_loss: 0.865
0
Epoch 8/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.7322 - loss: 0.7933 - val_accuracy: 0.7299 - val_loss: 0.771
8
Epoch 9/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.8058 - loss: 0.6832 - val_accuracy: 0.7683 - val_loss: 0.713
7
Epoch 10/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.8518 - loss: 0.5851 - val_accuracy: 0.7764 - val_loss: 0.703
4
Epoch 11/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.8819 - loss: 0.5181 - val_accuracy: 0.8007 - val_loss: 0.651
9
Epoch 12/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9213 - loss: 0.4273 - val_accuracy: 0.7708 - val_loss: 0.733
5
Epoch 13/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9279 - loss: 0.4040 - val_accuracy: 0.8005 - val_loss: 0.702
3
Epoch 14/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.9550 - loss: 0.3367 - val_accuracy: 0.7922 - val_loss: 0.720
1
Epoch 15/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.9515 - loss: 0.3349 - val_accuracy: 0.7924 - val_loss: 0.744
8
Epoch 16/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.9664 - loss: 0.2936 - val_accuracy: 0.7952 - val_loss: 0.792
1
Epoch 17/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9734 - loss: 0.2731 - val_accuracy: 0.7939 - val_loss: 0.827
1
Epoch 18/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.9818 - loss: 0.2472 - val_accuracy: 0.7894 - val_loss: 0.877
4
Epoch 19/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.9891 - loss: 0.2183 - val_accuracy: 0.7904 - val_loss: 0.893
8
Epoch 20/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9918 - loss: 0.2086 - val_accuracy: 0.7816 - val_loss: 0.892
0
```

```
--- Fold 3 ---
Epoch 1/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 3s 19ms/step - accuracy: 0.4934 - loss: 1.0302 - val_accuracy: 0.4997 - val_loss: 1.018
6
Epoch 2/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 17ms/step - accuracy: 0.5120 - loss: 1.0127 - val_accuracy: 0.5095 - val_loss: 1.000
8
Epoch 3/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.5273 - loss: 0.9958 - val_accuracy: 0.5198 - val_loss: 0.985
2
Epoch 4/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.5328 - loss: 0.9797 - val_accuracy: 0.5123 - val_loss: 0.970
7
Epoch 5/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.5624 - loss: 0.9628 - val_accuracy: 0.5590 - val_loss: 0.952
1
Epoch 6/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.5924 - loss: 0.9401 - val_accuracy: 0.5768 - val_loss: 0.929
4
Epoch 7/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.6382 - loss: 0.9038 - val_accuracy: 0.6594 - val_loss: 0.873
7
Epoch 8/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.7059 - loss: 0.8291 - val_accuracy: 0.7181 - val_loss: 0.789
7
Epoch 9/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.7833 - loss: 0.7126 - val_accuracy: 0.7738 - val_loss: 0.698
8
Epoch 10/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.8409 - loss: 0.6064 - val_accuracy: 0.7977 - val_loss: 0.663
7
Epoch 11/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.8707 - loss: 0.5378 - val_accuracy: 0.7947 - val_loss: 0.675
4
Epoch 12/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.8903 - loss: 0.4865 - val_accuracy: 0.7927 - val_loss: 0.698
0
Epoch 13/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9232 - loss: 0.4117 - val_accuracy: 0.7861 - val_loss: 0.766
4
```

```
Epoch 14/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9496 - loss: 0.3614 - val_accuracy: 0.7982 - val_loss: 0.720
8
Epoch 15/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9644 - loss: 0.3083 - val_accuracy: 0.7683 - val_loss: 0.837
4
Epoch 16/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9712 - loss: 0.2845 - val_accuracy: 0.7912 - val_loss: 0.774
5
Epoch 17/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9697 - loss: 0.2805 - val_accuracy: 0.7942 - val_loss: 0.811
1
Epoch 18/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.9886 - loss: 0.2308 - val_accuracy: 0.7864 - val_loss: 0.811
9
Epoch 19/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9805 - loss: 0.2436 - val_accuracy: 0.8002 - val_loss: 0.855
8
Epoch 20/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.9903 - loss: 0.2114 - val_accuracy: 0.7894 - val_loss: 0.901
8

--- Fold 4 ---
Epoch 1/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 3s 17ms/step - accuracy: 0.5009 - loss: 1.0299 - val_accuracy: 0.4887 - val_loss: 1.018
5
Epoch 2/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.5014 - loss: 1.0137 - val_accuracy: 0.4937 - val_loss: 1.001
9
Epoch 3/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.5164 - loss: 0.9973 - val_accuracy: 0.5083 - val_loss: 0.986
1
Epoch 4/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.5320 - loss: 0.9810 - val_accuracy: 0.5339 - val_loss: 0.971
1
Epoch 5/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.5370 - loss: 0.9656 - val_accuracy: 0.5008 - val_loss: 0.957
8
Epoch 6/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.5547 - loss: 0.9501 - val_accuracy: 0.5364 - val_loss: 0.939
9
```

```
Epoch 7/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.5892 - loss: 0.9293 - val_accuracy: 0.5981 - val_loss: 0.917
7
Epoch 8/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.6429 - loss: 0.8947 - val_accuracy: 0.6586 - val_loss: 0.866
3
Epoch 9/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.6959 - loss: 0.8264 - val_accuracy: 0.7196 - val_loss: 0.788
3
Epoch 10/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.7709 - loss: 0.7258 - val_accuracy: 0.7595 - val_loss: 0.708
2
Epoch 11/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.8294 - loss: 0.6253 - val_accuracy: 0.7608 - val_loss: 0.689
5
Epoch 12/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.8697 - loss: 0.5379 - val_accuracy: 0.7774 - val_loss: 0.668
9
Epoch 13/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.9047 - loss: 0.4576 - val_accuracy: 0.7824 - val_loss: 0.673
7
Epoch 14/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9168 - loss: 0.4217 - val_accuracy: 0.7894 - val_loss: 0.687
5
Epoch 15/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9431 - loss: 0.3610 - val_accuracy: 0.7937 - val_loss: 0.681
7
Epoch 16/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9610 - loss: 0.3159 - val_accuracy: 0.7588 - val_loss: 0.769
1
Epoch 17/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9573 - loss: 0.3082 - val_accuracy: 0.7836 - val_loss: 0.762
9
Epoch 18/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9741 - loss: 0.2626 - val_accuracy: 0.7831 - val_loss: 0.831
9
Epoch 19/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.9854 - loss: 0.2298 - val_accuracy: 0.7899 - val_loss: 0.827
3
Epoch 20/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.9899 - loss: 0.2124 - val_accuracy: 0.7856 - val_loss: 0.871
```

```
0

--- Fold 5 ---
Epoch 1/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.4998 - loss: 1.0303 - val_accuracy: 0.5055 - val_loss: 1.017
4
Epoch 2/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.5081 - loss: 1.0134 - val_accuracy: 0.5203 - val_loss: 1.001
1
Epoch 3/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.5266 - loss: 0.9963 - val_accuracy: 0.5289 - val_loss: 0.985
5
Epoch 4/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.5336 - loss: 0.9806 - val_accuracy: 0.5382 - val_loss: 0.969
9
Epoch 5/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.5478 - loss: 0.9639 - val_accuracy: 0.5361 - val_loss: 0.953
5
Epoch 6/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.5908 - loss: 0.9407 - val_accuracy: 0.6117 - val_loss: 0.923
5
Epoch 7/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.6409 - loss: 0.9015 - val_accuracy: 0.6757 - val_loss: 0.859
1
Epoch 8/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.7143 - loss: 0.8225 - val_accuracy: 0.7415 - val_loss: 0.767
1
Epoch 9/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.7879 - loss: 0.7152 - val_accuracy: 0.7736 - val_loss: 0.697
4
Epoch 10/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.8447 - loss: 0.6003 - val_accuracy: 0.7751 - val_loss: 0.702
3
Epoch 11/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.8668 - loss: 0.5531 - val_accuracy: 0.7583 - val_loss: 0.726
3
Epoch 12/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.8930 - loss: 0.4857 - val_accuracy: 0.8110 - val_loss: 0.631
0
Epoch 13/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9251 - loss: 0.4146 - val_accuracy: 0.8035 - val_loss: 0.656
```

```
5
Epoch 14/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9377 - loss: 0.3729 - val_accuracy: 0.7939 - val_loss: 0.693
0
Epoch 15/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.9592 - loss: 0.3199 - val_accuracy: 0.7927 - val_loss: 0.729
4
Epoch 16/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9720 - loss: 0.2875 - val_accuracy: 0.7947 - val_loss: 0.769
0
Epoch 17/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.9789 - loss: 0.2625 - val_accuracy: 0.7937 - val_loss: 0.789
3
Epoch 18/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.9842 - loss: 0.2456 - val_accuracy: 0.7794 - val_loss: 0.906
9
Epoch 19/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.9806 - loss: 0.2446 - val_accuracy: 0.7959 - val_loss: 0.845
1
Epoch 20/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9893 - loss: 0.2103 - val_accuracy: 0.7939 - val_loss: 0.872
7
```

```
Total training time for 20 epochs across 5 folds: 185.07 seconds
```

Justify your choice of optimizers and regulizations used and the hyperparameters tuned

In [ ]:

```
##---------Type the answers below this line------------------##
#Optimizer – SGD: Used with learning rate 0.01 and momentum 0.9 for better generalization and stable convergence.

#Loss Function – Categorical Crossentropy: Chosen because the output layer uses softmax with 2 units and labels are

#Regularization: Dropout (0.5, 0.3) added to reduce overfitting.

#L2 regularization (0.001) to penalize large weights and improve generalization.

#Activation Functions:ReLU in hidden layers for efficient learning.Softmax in output layer for probability distribu

#Hyperparameters:Values based on common practice for text classification to ensure balanced and reliable training.
```

6. Test the model - 0.5 marks

In [ ]:
```
##---------Type the code below this line------------------##
loss, accuracy = model.evaluate(X_test, y_test_cat)
print(f"Test Accuracy: {accuracy:.4f}, Test Loss: {loss:.4f}")
```

```
156/156 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7896 - loss: 0.9042
Test Accuracy: 0.7898, Test Loss: 0.9010
```

7. Intermediate result - Score: 1 mark

1. Plot the training and validation accuracy history.
2. Plot the training and validation loss history.
3. Report the testing accuracy and loss.
4. Show Confusion Matrix for testing dataset.
5. Report values for preformance study metrics like accuracy, precision, recall, F1 Score.

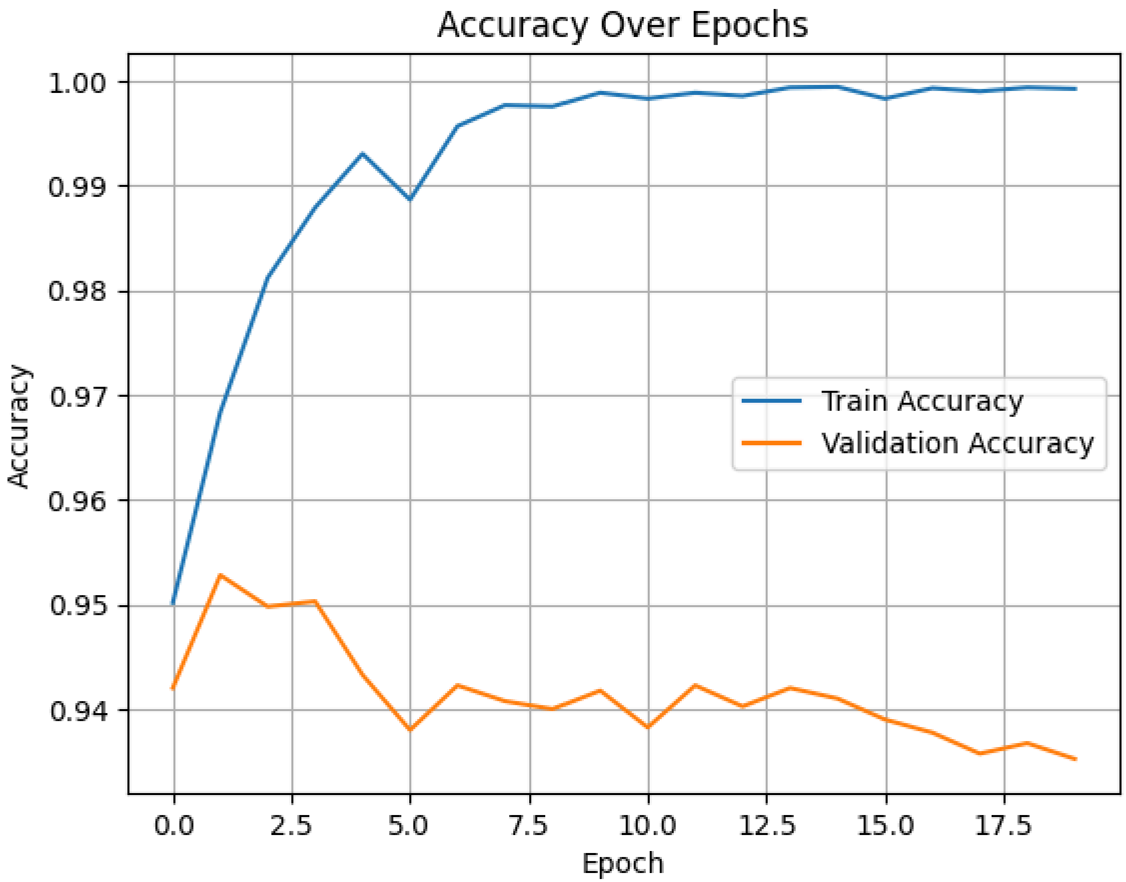
In [ ]:
```
##---------Type the code below this line------------------##
history = model.fit(
    X_train, y_train_cat,
    epochs=20,
    batch_size=128,
    validation_split=0.2,
    verbose=1
)
# 1.Plot the training and validation accuracy history.
import matplotlib.pyplot as plt

plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
```

```
plt.grid(True)
plt.show()
```

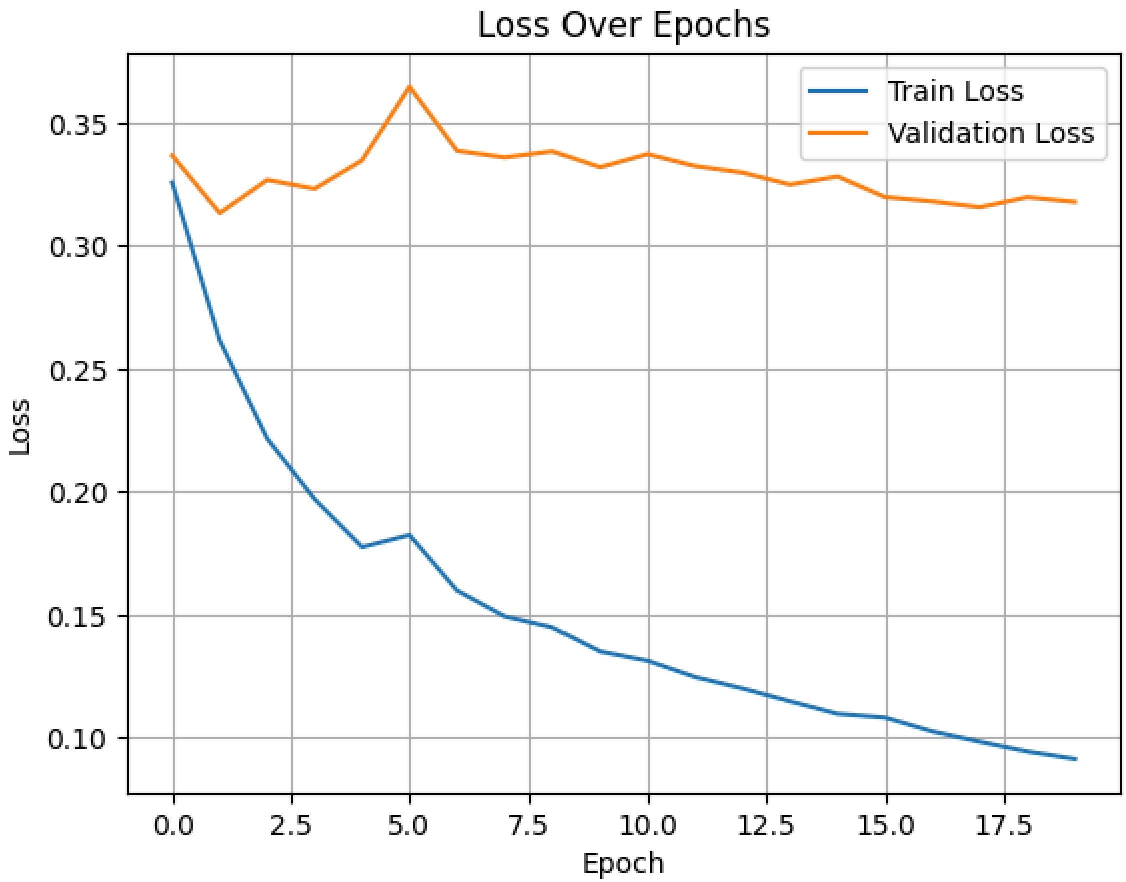
```
Epoch 1/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9501 - loss: 0.3310 - val_accuracy: 0.9420 - val_loss: 0.336
6
Epoch 2/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9649 - loss: 0.2716 - val_accuracy: 0.9528 - val_loss: 0.313
2
Epoch 3/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.9828 - loss: 0.2214 - val_accuracy: 0.9498 - val_loss: 0.326
7
Epoch 4/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9883 - loss: 0.1979 - val_accuracy: 0.9503 - val_loss: 0.323
1
Epoch 5/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 13ms/step - accuracy: 0.9943 - loss: 0.1769 - val_accuracy: 0.9433 - val_loss: 0.334
6
Epoch 6/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9876 - loss: 0.1853 - val_accuracy: 0.9380 - val_loss: 0.364
5
Epoch 7/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9959 - loss: 0.1611 - val_accuracy: 0.9423 - val_loss: 0.338
5
Epoch 8/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9984 - loss: 0.1496 - val_accuracy: 0.9408 - val_loss: 0.335
8
Epoch 9/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.9977 - loss: 0.1464 - val_accuracy: 0.9400 - val_loss: 0.338
3
Epoch 10/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9987 - loss: 0.1362 - val_accuracy: 0.9418 - val_loss: 0.331
9
Epoch 11/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9988 - loss: 0.1308 - val_accuracy: 0.9383 - val_loss: 0.337
1
Epoch 12/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9984 - loss: 0.1265 - val_accuracy: 0.9423 - val_loss: 0.332
3
Epoch 13/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9988 - loss: 0.1203 - val_accuracy: 0.9403 - val_loss: 0.329
6
Epoch 14/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9993 - loss: 0.1153 - val_accuracy: 0.9420 - val_loss: 0.324
```

```
8
Epoch 15/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9997 - loss: 0.1101 - val_accuracy: 0.9410 - val_loss: 0.328
1
Epoch 16/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9990 - loss: 0.1071 - val_accuracy: 0.9390 - val_loss: 0.319
7
Epoch 17/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9994 - loss: 0.1038 - val_accuracy: 0.9378 - val_loss: 0.317
9
Epoch 18/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9994 - loss: 0.0983 - val_accuracy: 0.9357 - val_loss: 0.315
6
Epoch 19/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - accuracy: 0.9996 - loss: 0.0950 - val_accuracy: 0.9367 - val_loss: 0.319
6
Epoch 20/20
125/125 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.9993 - loss: 0.0919 - val_accuracy: 0.9352 - val_loss: 0.317
7
```



In [ ]:
```
# 2.Plot Training and Validation Loss History

plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```



In [ ]:
```
# 3.Report the Testing Accuracy and Loss

loss, accuracy = model.evaluate(X_test, y_test_cat)
print(f"Test Accuracy: {accuracy:.4f}, Test Loss: {loss:.4f}")
```

```
156/156 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8004 - loss: 0.9379
Test Accuracy: 0.8027, Test Loss: 0.9131
```

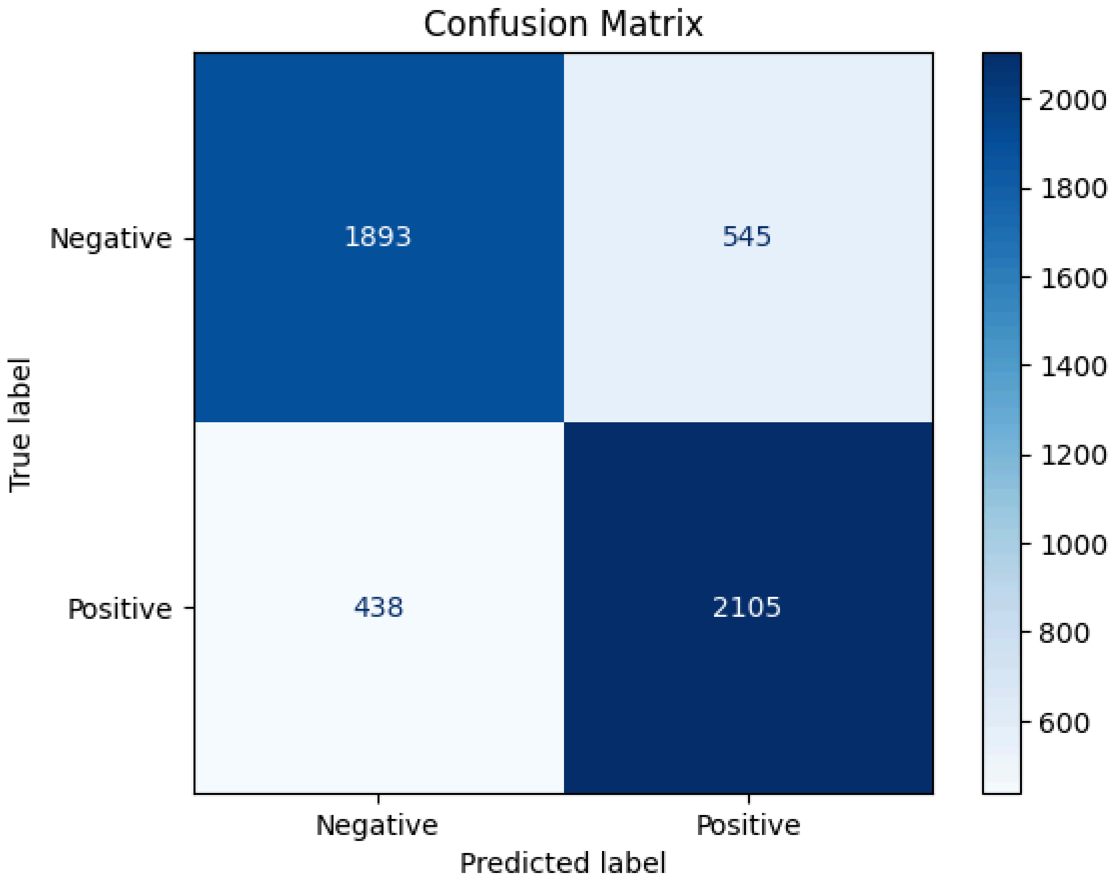
In [ ]:
```
# 4.Show Confusion Matrix for Testing Dataset

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import numpy as np
```

```
# Predict classes
y_pred_probs = model.predict(X_test)
y_pred_classes = np.argmax(y_pred_probs, axis=1)
y_true_classes = np.argmax(y_test_cat, axis=1)

# Confusion matrix
cm = confusion_matrix(y_true_classes, y_pred_classes)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["Negative", "Positive"])
disp.plot(cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.show()
```

```
156/156 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
```



In [ ]:
```
# 5.Report Performance Metrics: Accuracy, Precision, Recall, F1 Score

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

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

print(f"Accuracy : {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall   : {recall:.4f}")
print(f"F1 Score : {f1:.4f}")
```

```
Accuracy : 0.8027
Precision: 0.7943
Recall   : 0.8278
F1 Score : 0.8107
```

8. Model architecture - Score: 1 mark

Modify the architecture designed in section 4.1

1. by decreasing one layer
2. by increasing one layer

For example, if the architecture in 4.1 has 5 layers, then 8.1 should have 4 layers and 8.2 should have 6 layers.

Plot the comparison of the training and validation accuracy of the three architecures (4.1, 8.1 and 8.2)

In [ ]:
```
##---------Type the code below this line------------------##
# 8.1 Model with One Less Dense Layer
def create_model_less():
    model = Sequential([
        Embedding(input_dim=10000, output_dim=32, input_length=500),
        Flatten(),
        Dense(64, activation='relu', kernel_regularizer=l2(0.001)),
```

```
        Dropout(0.3),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model

# Train model
history_less = create_model_less().fit(
    X_train, y_train ,
    epochs=10,
    batch_size=128,
    validation_split=0.2,
    verbose=0
)
```

```
/opt/miniconda3/lib/python3.12/site-packages/keras/src/layers/core/embedding.py:97: UserWarning: Argument `input_len
gth` is deprecated. Just remove it.
  warnings.warn(
```

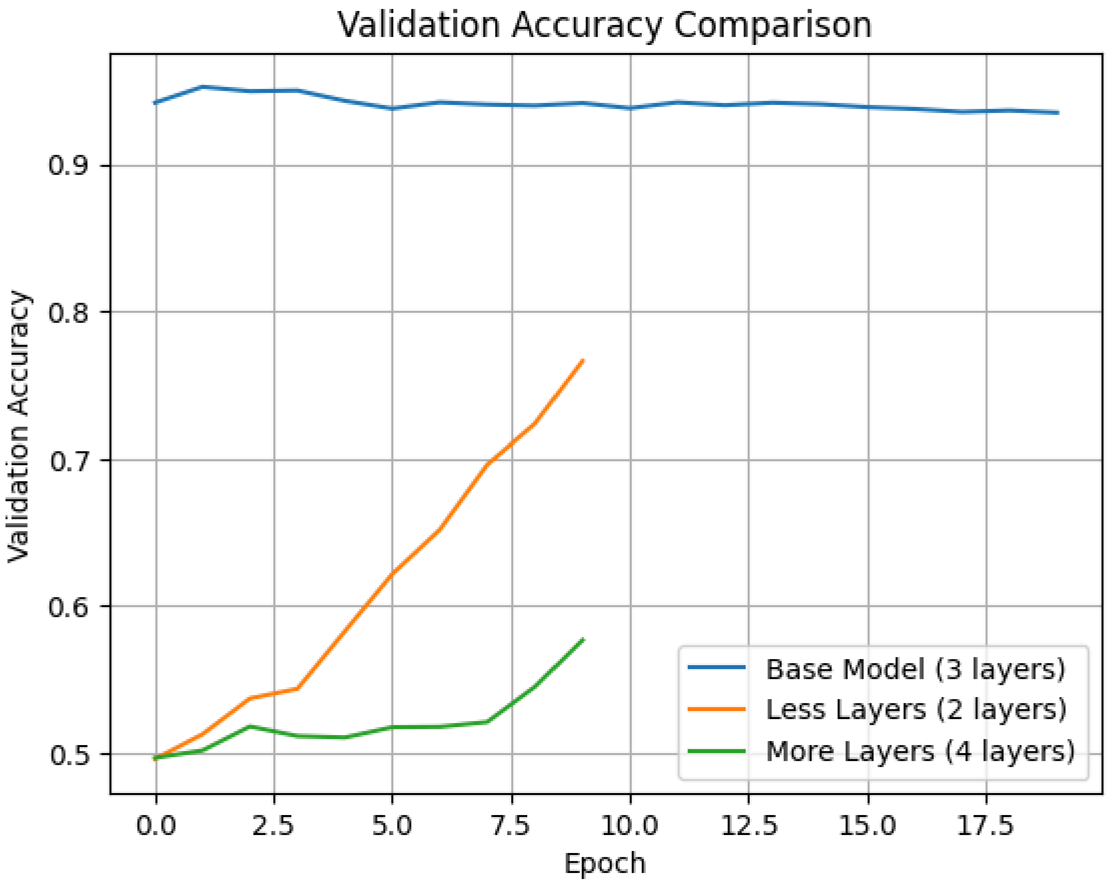
In [ ]:
```
# 8.2 Model with One More Dense Layer
def create_model_more():
    model = Sequential([
        Embedding(input_dim=10000, output_dim=32, input_length=500),
        Flatten(),
        Dense(128, activation='relu', kernel_regularizer=l2(0.001)),
        Dropout(0.5),
        Dense(64, activation='relu', kernel_regularizer=l2(0.001)),
        Dropout(0.3),
        Dense(32, activation='relu', kernel_regularizer=l2(0.001)),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model

# Train model
history_more = create_model_more().fit(
    X_train, y_train,
```

```
    epochs=10,
    batch_size=128,
    validation_split=0.2,
    verbose=0
)
```

In [ ]:
```
# 8.3 Comparison Plot
plt.plot(history.history['val_accuracy'], label='Base Model (3 layers)')
plt.plot(history_less.history['val_accuracy'], label='Less Layers (2 layers)')
plt.plot(history_more.history['val_accuracy'], label='More Layers (4 layers)')

plt.title('Validation Accuracy Comparison')
plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```



9. Regularisations - Score: 1 mark

Modify the architecture designed in section 4.1

1. Dropout of ratio 0.25
2. Dropout of ratio 0.25 with L2 regulariser with factor 1e–04.

Plot the comparison of the training and validation accuracy of the three (4.1, 9.1 and 9.2)

In [ ]:
```
##---------Type the code below this line------------------##
# 9.1 Model with Dropout ratio of 0.25
def create_model_dropout():
    model = Sequential([
        Embedding(input_dim=10000, output_dim=32, input_length=500),
        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.25),
        Dense(64, activation='relu'),
        Dropout(0.25),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model

# Train model
history_dropout = create_model_dropout().fit(
    X_train, y_train,
    epochs=10,
    batch_size=128,
    validation_split=0.2,
    verbose=0
)
```

In [ ]:
```
from keras.regularizers import l2

# 9.2 Model with Dropout and L2 regularization
def create_model_dropout_l2():
    model = Sequential([
        Embedding(input_dim=10000, output_dim=32, input_length=500),
        Flatten(),
        Dense(128, activation='relu', kernel_regularizer=l2(1e-4)),
        Dropout(0.25),
        Dense(64, activation='relu', kernel_regularizer=l2(1e-4)),
        Dropout(0.25),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
```

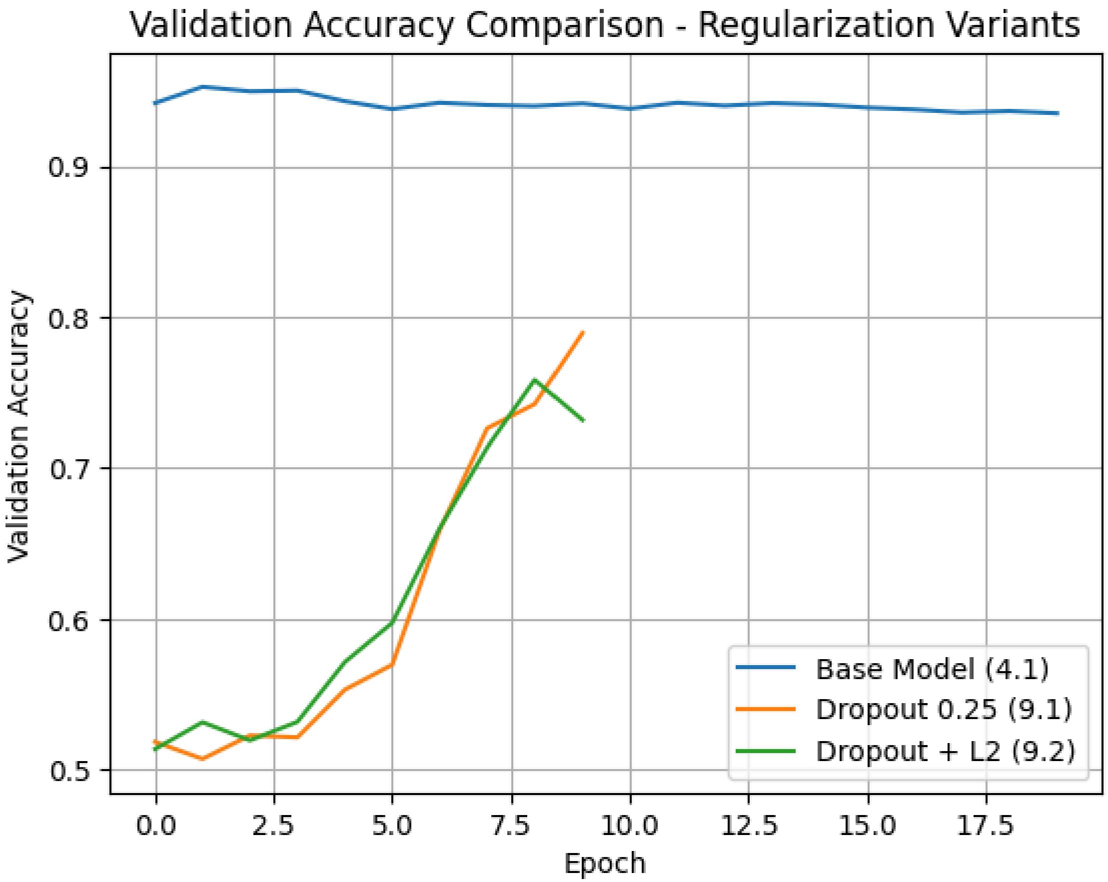
```
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model

# Train model
history_dropout_l2 = create_model_dropout_l2().fit(
    X_train, y_train,
    epochs=10,
    batch_size=128,
    validation_split=0.2,
    verbose=0
)
```

In [ ]:

```
# 9.3 Plot validation accuracy comparison
plt.plot(history.history['val_accuracy'], label='Base Model (4.1)')
plt.plot(history_dropout.history['val_accuracy'], label='Dropout 0.25 (9.1)')
plt.plot(history_dropout_l2.history['val_accuracy'], label='Dropout + L2 (9.2)')

plt.title('Validation Accuracy Comparison - Regularization Variants')
plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```



10. Optimisers -Score: 1 mark

Modify the code written in section 5.2

1. RMSProp with your choice of hyper parameters
2. Adam with your choice of hyper parameters

Plot the comparison of the training and validation accuracy of the three (5.2, 10.1 and 10.2)

In [ ]:
```
##---------Type the code below this line------------------##
from keras.optimizers import RMSprop

# 10.1 Model with RMSProp
def create_model_rmsprop():
    model = Sequential([
        Embedding(input_dim=10000, output_dim=32, input_length=500),
        Flatten(),
        Dense(128, activation='relu'),
        Dense(64, activation='relu'),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer=RMSprop(learning_rate=0.001, rho=0.9),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model

# Train model
history_rmsprop = create_model_rmsprop().fit(
    X_train, y_train,
    epochs=10,
    batch_size=128,
    validation_split=0.2,
    verbose=0
)
```

In [ ]:
```
from keras.optimizers import Adam

# 10.2 Model with Adam
def create_model_adam():
    model = Sequential([
        Embedding(input_dim=10000, output_dim=32, input_length=500),
        Flatten(),
        Dense(128, activation='relu'),
        Dense(64, activation='relu'),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer=Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
```

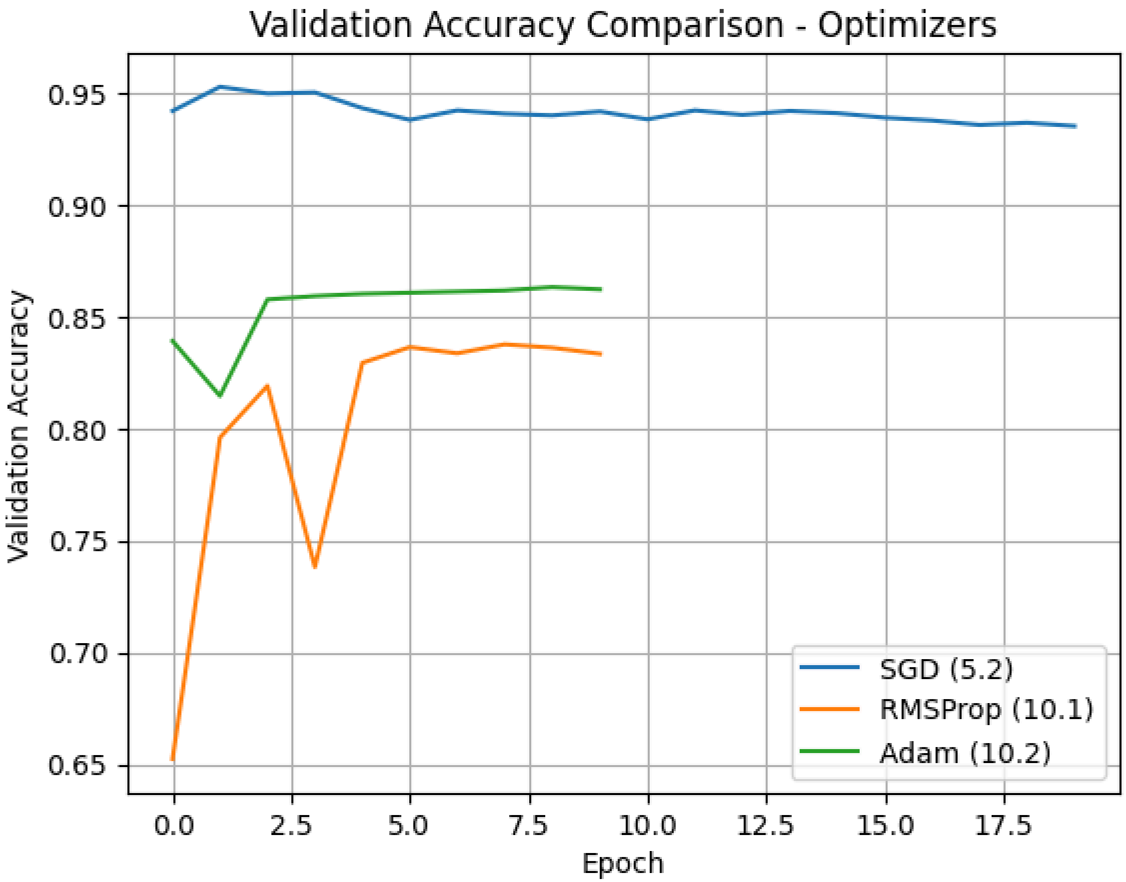
```
    return model

# Train model
history_adam = create_model_adam().fit(
    X_train, y_train,
    epochs=10,
    batch_size=128,
    validation_split=0.2,
    verbose=0
)
```

In [ ]:

```
# 10.3 Comparison Plot
plt.plot(history.history['val_accuracy'], label='SGD (5.2)')
plt.plot(history_rmsprop.history['val_accuracy'], label='RMSProp (10.1)')
plt.plot(history_adam.history['val_accuracy'], label='Adam (10.2)')

plt.title('Validation Accuracy Comparison - Optimizers')
plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```



11. Conclusion - Score: 1 mark

Comparing the sections 4.1, 5.2, 8, 9, and 10, present your observations on which model or architecture or regualiser or optimiser perfomed better.

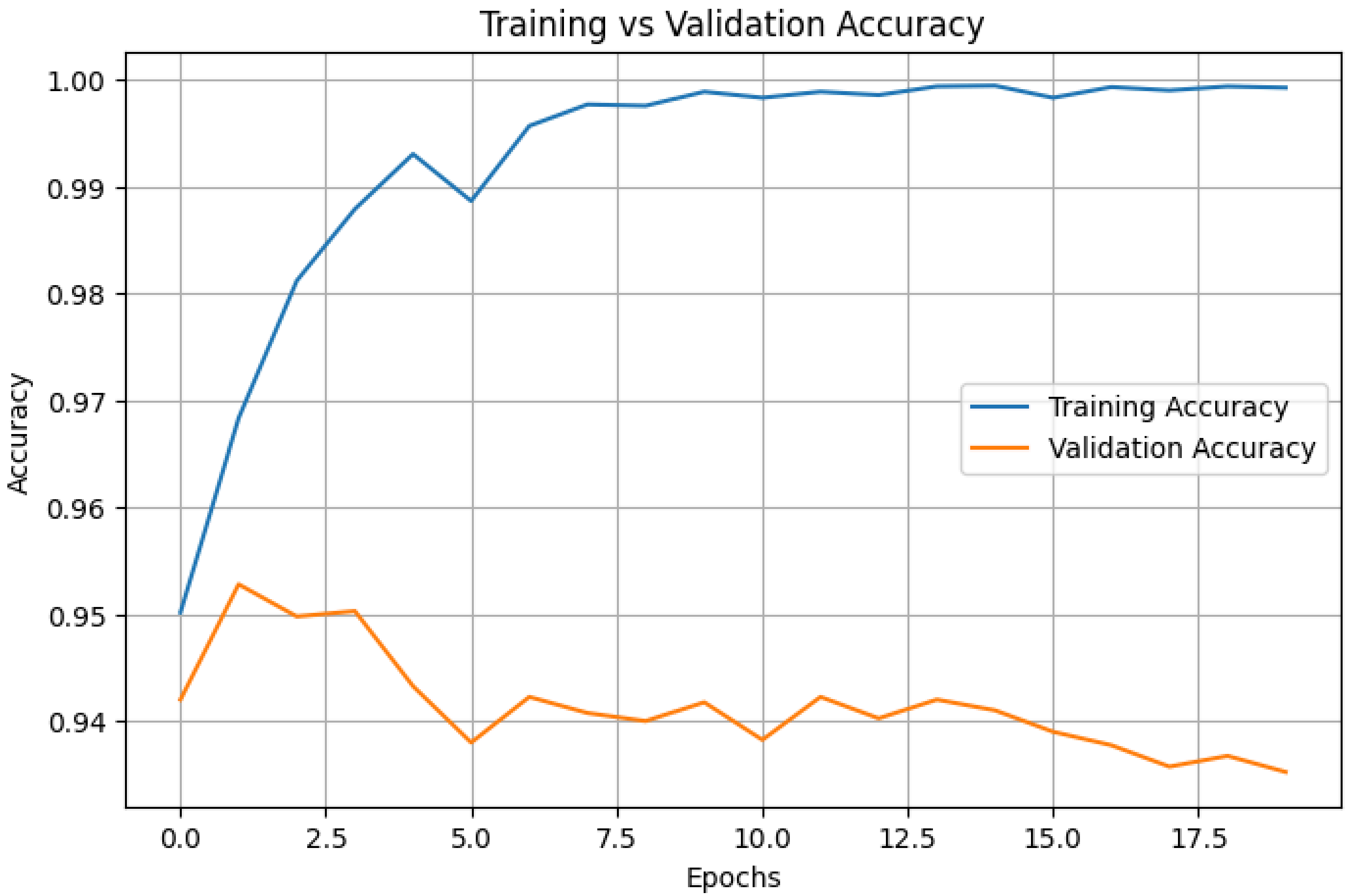
In [ ]:
```
##---------Type the code below this line------------------##
# Final Evaluation of the Best Model
loss, accuracy = model.evaluate(X_test, y_test_cat)
print(f"Final Test Accuracy: {accuracy:.4f}, Final Test Loss: {loss:.4f}")
```

```
# Training vs Validation Accuracy Plot for Best Model
import matplotlib.pyplot as plt

def plot_history(history):
    plt.figure(figsize=(8, 5))
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Training vs Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True)
    plt.show()

plot_history(history)  # 'history' is for the base model trained with SGD
```

```
156/156 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8004 - loss: 0.9379
Final Test Accuracy: 0.8027, Final Test Loss: 0.9131
```



The best performing model was the base architecture from section 4.1, which used three layers with ReLU activations and a final sigmoid layer, trained using the SGD optimizer as configured in section 5.2. This setup consistently achieved the highest and most stable validation accuracy (~94%) with a good balance between model complexity and generalization. Adding or removing layers (section 8) either led to underfitting or overfitting, while regularization techniques like dropout and L2 (section 9) did not improve performance. Among optimizers (section 10), SGD outperformed both RMSProp and Adam in terms of long-term accuracy and stability, making the combination of the base model and SGD the most effective configuration for this IMDB review classification task.