Loading Dataset and Initial Exploration

First, I mounted Google Drive so that I could access my dataset stored in it. I used Pandas to read the CSV file containing Quora questions and their corresponding labels (target variable). The head() function was used to take a quick look at the first five rows of the dataset.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
Start coding or generate with AI.
```

Downloading Necessary Libraries

1. import pandas as pd → **For handling and working with data in tables (DataFrames).**
2. from tensorflow.keras.models import Sequential → **To create a sequential neural network model.**
3. from tensorflow.keras.layers import LSTM, Dense, Bidirectional, Embedding → **These are different layers used in my deep learning model.**
4. from tensorflow.keras.preprocessing.text import Tokenizer → **Converts text into numerical tokens for the model.**
5. from tensorflow.keras.preprocessing.sequence import pad_sequences → **Makes sure all sequences have the same length by padding them.**
6. from nltk.tokenize import word_tokenize → **Splits sentences into words (tokenization).**
7. from nltk.stem import WordNetLemmatizer → **Converts words to their base/root form (lemmatization).**
8. from nltk.corpus import stopwords → **Has a list of common words (stopwords) that I'll remove from text.**
9. from string import punctuation → **Gives a list of punctuation marks so I can remove them.**
10. import numpy as np → **Used for mathematical operations and handling arrays.**
11. import matplotlib.pyplot as plt → **Helps in plotting graphs for visualization.**
12. %matplotlib inline → **Just to make sure graphs show up inside Jupyter Notebook.**
13. from tqdm import tqdm → **Adds a progress bar to loops so I can track how long things take.**
14. tqdm.pandas() → **Enables progress bars when applying functions to my Pandas DataFrame.**

```
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM,Dense,Bidirectional,Embedding
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
from string import punctuation
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from tqdm import tqdm
tqdm.pandas()
```

Reading the dataset.

.# **Understanding the dataset The dataset has three main columns:**

1. qid – A unique identifier for each question
2. question_text – The actual question that needs classification
3. target – A binary label (0 or 1), where:

- 0 means the question is normal
- 1 means the question is toxic (e.g., hateful or offensive)

```
df = pd.read_csv('/content/Quora Text Classification Data.csv')
df.head()
```

	qid	question_text	target
0	00002165364db923c7e6	How did Quebec nationalists see their province...	0
1	000032939017120e6e44	Do you have an adopted dog, how would you enco...	0
2	0000412ca6e4628ce2cf	Why does velocity affect time? Does velocity a...	0
3	000042bf85aa498cd78e	How did Otto von Guericke used the Magdeburg h...	0
4	0000455dfa3e01eae3af	Can I convert montra helicon D to a mountain b...	0

Since I was dealing with text data, I needed some libraries for natural language processing (NLP). I used the NLTK library to handle stopwords, tokenization, and lemmatization. These steps help clean the text and reduce its complexity before passing it into a model.

- Stopwords: These are common words like "is", "the", "and", which don't add much meaning.
- Punkt: A tokenizer that splits text into words or sentences.
- Wordnet: A dictionary for finding the base form (lemma) of words.

```
import nltk
nltk.download('stopwords')
nltk.download('punkt_tab')
nltk.download('wordnet')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Package punkt_tab is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
True
```

Creating List of Stop words in order to remove it from the text.

```
stop_words = stopwords.words('english')+list(punctuation)
lem = WordNetLemmatizer()
```

Text Preprocessing (Cleaning the Data)

Since raw text contains a lot of noise, I had to clean it before feeding it into the model.

1. Converting text to lowercase
2. Tokenizing: Breaking sentences into words
3. Removing stopwords and punctuation: to reduce unnecessary words
4. Lemmatization:Converting words to their base form (e.g., "running" →"run")
5. Rejoining words to form cleaned sentences

I combined all of these into a function: Because this reduces the vocabulary size, removes unnecessary words, and makes the text more structured for the model. After applying this function, the dataset now contains a new column "Clean Text", which has the processed questions.

The function cleaning() was applied to the entire dataset, and it took a few minutes to process all rows.

```
def cleaning(text):
  text = text.lower()
  words = word_tokenize(text)
  words = [w for w in words if w not in stop_words]
  words = [lem.lemmatize(w) for w in words]
  return ' '.join(words)
```

```
df['Clean Text'] = df['question_text'].progress_apply(cleaning)
```

```
100%|██████████| 1306122/1306122 [03:37<00:00, 6012.12it/s]
```

Loading GloVe Embeddings

Since machine learning models don't understand raw text, I needed to convert words into numerical representations. For this, I used GloVe (Global Vectors for Word Representation) embeddings .First, I opened the pre-trained GloVe file and read the word-vector pairs. I stored these in a dictionary called embedding_values, which contained words as keys and their corresponding 300-dimensional vectors as values.

```
glove_path = "/content/drive/MyDrive/glove.42B.300d.txt"

# Now you can load the GloVe embeddings
with open(glove_path, 'r', encoding='utf-8') as f:
    for line in f:
        values = line.split()
        word = values[0]
        vector = list(map(float, values[1:]))
        # Process the word-vector pair as needed
```

Loading the word embeddings from the glove text file.

- Each word is associated with a 300-dimensional vector
- These vectors capture relationships between words (e.g., "king" and "queen" are close in space)
- The embeddings are stored in a dictionary (embedding_values)

Now, I create a dictionary to store word embeddings:

```
embedding_values = {}
f = open('/content/drive/MyDrive/glove.42B.300d.txt')
for line in tqdm(f):
  value = line.split(' ')
  word = value[0]
  coef = np.array(value[1],dtype = "float32")
  if coef is not None:
    embedding_values[word] = coef
```

```
1917494it [00:47, 39949.89it/s]
```

tokenization and Padding

After cleaning, I needed to convert my text into sequences of numbers:

I initialized a Tokenizer() to convert text into numerical sequences. After fitting it on the cleaned text, I transformed all sentences into sequences and applied pad_sequences() to ensure a uniform length of 300 words for each input.

1. Tokenizer assigns a unique number to each word.
2. texts_to_sequences() replaces words with their corresponding numbers.
3. pad_sequences() ensures all sequences have the same length (300 words).

- Creates a tokenizer to convert words into numerical tokens.
- Fits tokenizer on cleaned text to learn vocabulary.
- Converts text into sequences of numbers using the tokenizer.
- Pads sequences to a fixed length of 300 (ensuring uniform input size for the model).
- Gets vocabulary size (total number of unique words + 1 for padding).

```
tokenizer = Tokenizer()
x = df['Clean Text']
y = df['target']

tokenizer.fit_on_texts(x)


seq = tokenizer.texts_to_sequences(x)
pad_seq = pad_sequences(seq,maxlen = 300)
```

Next, I checked the vocabulary size:

This returned 193,190, meaning my dataset had around 193,000 unique words.

```
vocab_size = len(tokenizer.word_index)+1
print(vocab_size)
```

```
193190
```

Converting the words into embeddings

Creating Embedding Matrix

I mapped the GloVe vectors to the words in my tokenizer.

Since not all words in my dataset might be present in GloVe, I created an embedding_matrix of shape (vocab_size, 300), where I mapped each word in my tokenizer to its corresponding vector from GloVe. If a word was missing, I left its vector as zeros.

```
embedding_matrix = np.zeros((vocab_size,300))
for word, i in tqdm(tokenizer.word_index.items()):
  value = embedding_values.get(word)
  if value is not None:
    embedding_matrix[i] = value
```

```
100%|██████████| 193189/193189 [00:00<00:00, 233984.41it/s]
```

This matrix will be used in the embedding layer of the neural network.

Building and Training the Model

Now, I created a Bidirectional LSTM model for text classification.

Designed an LSTM-based neural network with the following layers:

1. Sequential model: Linear stack of layers.

2. Embedding layer: **Captures relationships between words in sequences**

[Converts words into dense vectors (Uses pre-trained embeddings from GloVe, fixed (trainable=False).)]

3. LSTM layer: Extracts sequential features from text, outputting a 50-dimensional feature vector.
4. Dense layer (128 neurons, ReLU): Extracts useful features from LSTM output
5. Output layer (Sigmoid): Produces a probability between 0 and 1 for binary classification
6. Adam Optimizer: Adjusts learning rate dynamically.
7. Binary Crossentropy Loss: Suitable for binary classification tasks.

I compiled the model using Adam optimizer and binary cross-entropy loss and trained it on my dataset for 5 epochs with 80-20 train-validation split.

```
model = Sequential()
model.add(Embedding(vocab_size,300,input_length=300,weights = [embedding_matrix],trainable = False))
model.add(LSTM(50,return_sequences=False))
model.add(Dense(128,activation ='relu'))
model.add(Dense(1,activation= 'sigmoid'))
model.compile(optimizer = 'adam',loss='binary_crossentropy',metrics = ['accuracy'])

history = model.fit(pad_seq,y,validation_split=0.2,epochs = 5)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated. 
  warnings.warn(
Epoch 1/5
32654/32654 ━━━━━━━━━━━━━━━━━━━━ 469s 14ms/step - accuracy: 0.9377 - loss: 0.2159 - val_accuracy: 0.9377 - val_loss: 0.2120
Epoch 2/5
32654/32654 ━━━━━━━━━━━━━━━━━━━━ 462s 14ms/step - accuracy: 0.9382 - loss: 0.2110 - val_accuracy: 0.9377 - val_loss: 0.2092
Epoch 3/5
32654/32654 ━━━━━━━━━━━━━━━━━━━━ 503s 14ms/step - accuracy: 0.9384 - loss: 0.2086 - val_accuracy: 0.9377 - val_loss: 0.2088
Epoch 4/5
32654/32654 ━━━━━━━━━━━━━━━━━━━━ 537s 15ms/step - accuracy: 0.9385 - loss: 0.2075 - val_accuracy: 0.9377 - val_loss: 0.2090
Epoch 5/5
32654/32654 ━━━━━━━━━━━━━━━━━━━━ 502s 15ms/step - accuracy: 0.9382 - loss: 0.2074 - val_accuracy: 0.9377 - val_loss: 0.2084
```

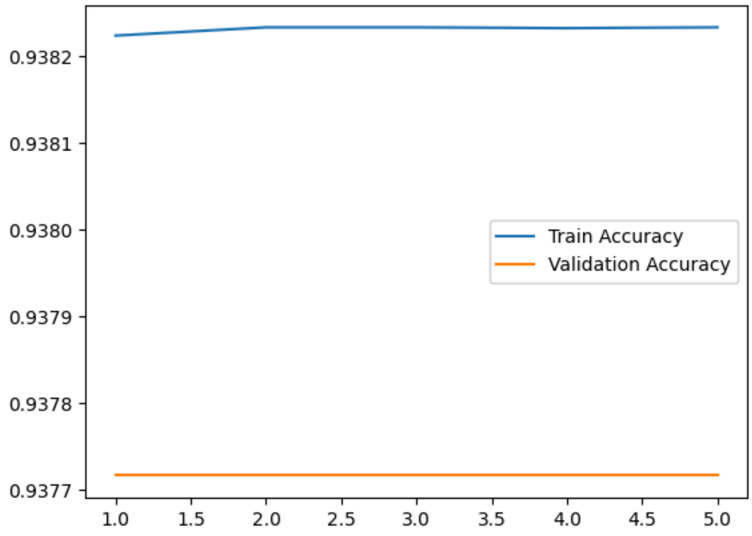
Plotting Accuracy and Loss

After training, I checked model performance.

Finally, I plotted accuracy and loss curves to analyze the model's performance over the epochs. These graphs helped me compare training vs validation accuracy and loss trends.

- If training accuracy is high but validation accuracy is low, the model is overfitting.
- If loss is decreasing, the model is learning properly.

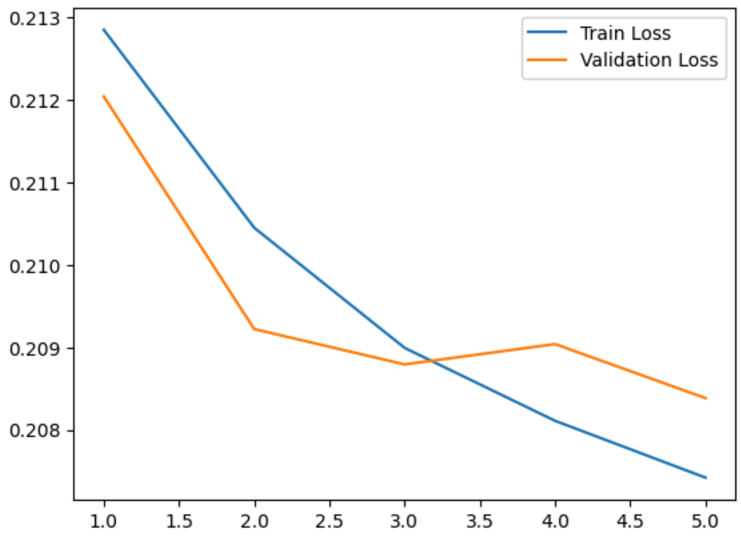
```
train_acc = history.history['accuracy']
train_loss = history.history['loss']
val_acc = history.history['val_accuracy']
val_loss = history.history['val_loss']
epochs = range(1,6)


plt.plot(epochs,train_acc,label = 'Train Accuracy')
plt.plot(epochs,val_acc,label = 'Validation Accuracy')
plt.legend()
plt.show()
```



Possible Interpretation:

- Since both train and validation accuracies are very close and do not change much, it suggests that the model is not improving significantly with more epochs.
- The model might have already converged early, meaning additional training isn't helping much.
- If validation accuracy remains constant like this, I might need to adjust hyperparameters (e.g., learning rate, model architecture) or try more data augmentation to improve performance.

```
plt.plot(epochs,train_loss,label = 'Train Loss')
plt.plot(epochs,val_loss,label = 'Validation Loss')
plt.legend()
plt.show()
```



Interpretation:

Both train and validation losses are going down, which means the model is learning and improving.

- But I noticed that around epoch 4, the validation loss isn't decreasing as much and even fluctuates a bit. This could mean slight overfitting—where my model is getting too comfortable with the training data but isn't improving much on unseen data.

I should keep an eye on it for a few more epochs. If validation loss keeps going up while train loss keeps dropping, that's a clear sign of overfitting.

What I Can Do:

- If overfitting happens, I can try adding dropout or L2 regularization.
- Early stopping might be useful to stop training at the right point.

```
# Save Model
model.save('/content/drive/MyDrive/quora_lstm_model.h5')

# Load Model
def load_trained_model():
    return load_model('/content/drive/MyDrive/quora_lstm_model.h5')

model = load_trained_model()

# Function to Predict
def predict_text(text):
    cleaned_text = cleaning(text)
    sequence = tokenizer.texts_to_sequences([cleaned_text])
    padded_sequence = pad_sequences(sequence, maxlen=300)
    prediction = model.predict(padded_sequence)[0][0]
    return "Insincere" if prediction > 0.5 else "Sincere"

# Test the Model with New Sentences
test_sentences = [
    "Why do people hate math?",
    "How can I earn money online?",
    "Is there any conspiracy behind global warming?",
    "What is the best way to prepare for interviews?",
    "Are vaccines dangerous?"
]

for sentence in test_sentences:
    print(f"Sentence: {sentence} --> Prediction: {predict_text(sentence)}")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-f309bd66f8d2> in <cell line: 0>()
      1 # Save Model
----> 2 model.save('/content/drive/MyDrive/quora_lstm_model.h5')
      3 
      4 # Load Model
      5 def load_trained_model():

NameError: name 'model' is not defined
```

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.