GALGOTIAS UNIVERSITY

SCHOOL OF COMPUTING SCIENCE & ENGINEERING

NLP Project File

On

Chatbot and Virtual Assistant in Customer Service

Submitted By:

Shivanand Gupta	22SCSE1012610
Shivang Kakkar	22SCSE1180121
Shivangi Sehgal	22SCSE1180097
Shreyash Upadhyay	22SCSE1012304

Course/Subject: Natural Language Processing

Instructor’s Name: Dr. Umesh Lilhore

Date: May 22, 2025

Contents:

Introduction

Brief overview of the NLP problem or case study:

This project focuses on building a rule-based chatbot and virtual assistant for customer service using Natural Language Processing (NLP). The primary objective is to evaluate chatbot responses using sentiment analysis, keyword overlap, and POS tagging and to iteratively improve chatbot performance based on these analyses.

Motivation and Objective:

- Provide instant, consistent, and intelligent responses to customer queries.
- Reduce manual support workload and enhance customer satisfaction.
- Evaluate and refine chatbot quality using rule-based and machine learning methods.

Scope and Limitations:

- Scope includes basic sentiment, keyword, and grammatical analysis.
- Does not include multilingual or voice-based interactions.
- Rule-based approach may not generalize to highly diverse queries.

Background / Literature Review

NLP allows machines to interpret human language through techniques like tokenization, sentiment analysis, POS tagging, etc. Chatbots are commonly used in sectors like banking, e-commerce, and healthcare. HDFC Bank’s EVA is an example of a well-implemented virtual assistant.

Summary of relevant NLP concepts or techniques:

1. **Text Preprocessing:**
   To ensure consistency and improve model accuracy, text preprocessing techniques were applied to both user questions and chatbot responses. These included lowercasing, punctuation cleaning.

2. **Keyword Overlap Matching:**
   A rule-based approach was implemented where the chatbot selects the most relevant response by calculating the keyword overlap score between the user input and stored questions. This simple but effective technique helps in matching semantically similar queries.

3. **Part-of-Speech (POS) Tagging:**
   POS tagging was used to verify the grammatical structure of responses, specifically checking for the presence of verbs in answers. This ensured that generated responses were action-oriented and informative.

4. **Sentiment Analysis:**
   The Sentiment Intensity Analyzer from NLTK’s VADER tool was applied to assess the emotional tone of both user inputs and chatbot responses.

5. **Logistic Regression:**
   A supervised machine learning algorithm used to classify whether a chatbot's answer is ‘good’ or ‘bad’ based on engineered features. Logistic regression provided a simple but interpretable baseline model for classification.

6. **Random Forest Classifier:**
   An ensemble-based machine learning model used for more robust classification. It builds multiple decision trees and outputs the majority vote.

7. **LSTM (Long Short-Term Memory):**
   A basic LSTM-based neural model was implemented to classify the quality of chatbot responses based on the cleaned text input. The LSTM model served as a baseline for exploring deep learning-based performance improvements over rule-based logic.

8. **Visualization with Heatmaps:**
   Correlation heatmaps were generated using libraries like seaborn and matplotlib to visualize relationships between sentiment scores, keyword overlap, POS checks, and final labels. This helped interpret how each feature impacted response quality.

Related Work or Similar Studies:

Numerous studies and implementations have explored the use of chatbots and virtual assistants in enhancing customer service. The evolution of chatbot systems can broadly be classified into rule-based systems, retrieval-based systems, and generative models.

1. **Rule-Based Chatbots:**
   Early implementations such as ELIZA (1966) used pattern-matching techniques to simulate conversation. These systems rely heavily on predefined rules and keyword matching, similar to the approach used in this project.

2. **Retrieval-Based Chatbots:**
   Modern retrieval-based systems, such as those used by companies like HDFC Bank's EVA and SBI's SIA, employ advanced information retrieval techniques and use semantic similarity (e.g., cosine similarity of vector embeddings) to find the best matching response from a knowledge base. While more flexible than rule-based bots, they still rely on pre-existing response datasets.

3. **Sentiment-Aware Dialogue Systems:**
   Research has shown that integrating sentiment analysis can significantly improve the relevance and tone of chatbot responses. Projects and publications such as "Sentiment-Aware Neural Chatbots" have demonstrated how aligning the sentiment of responses with user emotion enhances user satisfaction—an aspect also explored in this project through sentiment polarity matching.

4. **Neural Network-Based Chatbots (LSTM, Transformers):**
   Advanced models like sequence-to-sequence (seq2seq) LSTM and Transformer-based models (e.g., BERT, GPT) have become popular for generating contextual, fluent conversations. While our project uses a basic LSTM model for response classification, future work can integrate transformer-based models for end-to-end conversation generation and response validation.

5. **Practical Applications:**
   Many businesses now use chatbots in customer support to reduce workload and improve response time. Examples include Amazon's Alexa, Google Assistant, and Apple's Siri, which combine rule-based and neural approaches for voice and text-based interactions.

Dataset Description

Source of Data:

The dataset used for this project was sourced from Kaggle, titled **“Dataset for Chatbot”**. It consists of frequently asked customer service queries and their corresponding responses. The dataset is publicly available and is commonly used for building and evaluating rule-based or machine learning-based chatbot systems.

Size and Format:

- **Format:** CSV (Comma-Separated Values) file
- **Size:** Contains approximately **1000 question-answer pairs**
- **Key Columns:**
    - **questions**: Raw customer queries
    - **answers**: Corresponding chatbot responses
    - **questions_cleaned**: Preprocessed versions of questions
    - **answers_cleaned**: Preprocessed versions of answers
    - Additional columns used for analysis:
        - **q_sentiment_rule, a_sentiment_rule** – Sentiment labels
        - **keyword_overlap_score** – Rule-based similarity score
        - **answer_verb_check** – POS tag-based validation
        - **true_label** – Manually labeled ground truth for evaluation

Preprocessing Steps Applied:

To prepare the data for rule-based analysis and model training, the following steps were applied:

- **Lowercasing:** To standardize text and reduce variation.
- **Punctuation Removal:** To eliminate noise in token matching.
- **Tokenization:** For breaking sentences into individual words.
- **POS Tagging:** To identify verbs in the answers, ensuring grammatical quality.
- **Sentiment Analysis Preparation:** Cleaned inputs were passed through a rule-based sentiment analyzer to score and evaluate emotion alignment between questions and answers.

Methodology

Description of Approach:

The project uses a **rule-based approach** for chatbot response evaluation. This involves defining explicit, interpretable rules to determine the quality of chatbot responses based on lexical overlap, grammatical structure, and sentiment alignment. Additionally, for performance comparison and improvement, **machine learning** (Random Forest, Logistic Regression) and **deep learning** (LSTM) models were explored.

Tools and Libraries Used:

The following Python libraries were used to implement the chatbot analyzer:

- **Pandas** – For data manipulation and preprocessing
- **NLTK** – For natural language preprocessing tasks (tokenization, stopwords, POS tagging, sentiment analysis)
- **TensorFlow / Keras** – For building and training the LSTM model
- **Matplotlib / Seaborn** – For data visualization (e.g., heatmaps)
- **NumPy** – For numerical operations

Explanation of Key Algorithms or Models Implemented:

1. **Rule-Based Analyzer:**

    - **Keyword Overlap:** Measures similarity between cleaned user queries and chatbot answers based on shared words. Higher overlap indicates better relevance.
    - **Sentiment Matching:** Uses NLTK's SentimentIntensityAnalyzer to compare the sentiment polarity of the question and the answer. Matching sentiment boosts confidence in answer alignment.
    - **POS Tagging:** Checks if the chatbot response contains valid verbs using part-of-speech tagging to ensure grammatical correctness.

2. **Supervised Machine Learning Models:**

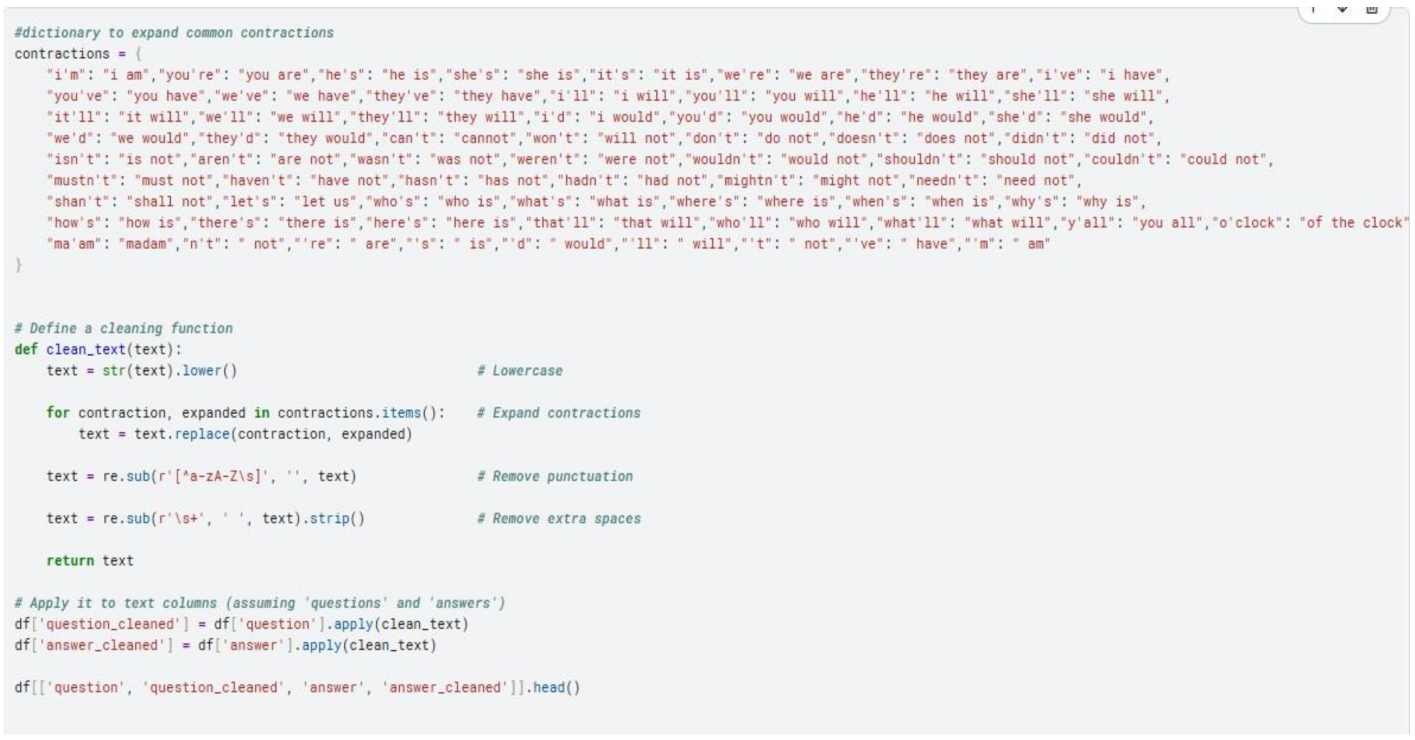
    - **Logistic Regression & Random Forest Classifier:** These models are trained on features like keyword overlap score, sentiment match, and verb check to classify responses as "good" or "bad".
    - Feature vector example: [overlap_score, q_sentiment, a_sentiment, has_verb]

3. **Deep Learning Model:**

    - **LSTM (Long Short-Term Memory):** A recurrent neural network model trained on embedded question-answer pairs to learn sequential patterns. It was implemented to capture deeper semantic relationships beyond lexical similarity.
    - Word embeddings were created using Keras Tokenizer and padded sequences.

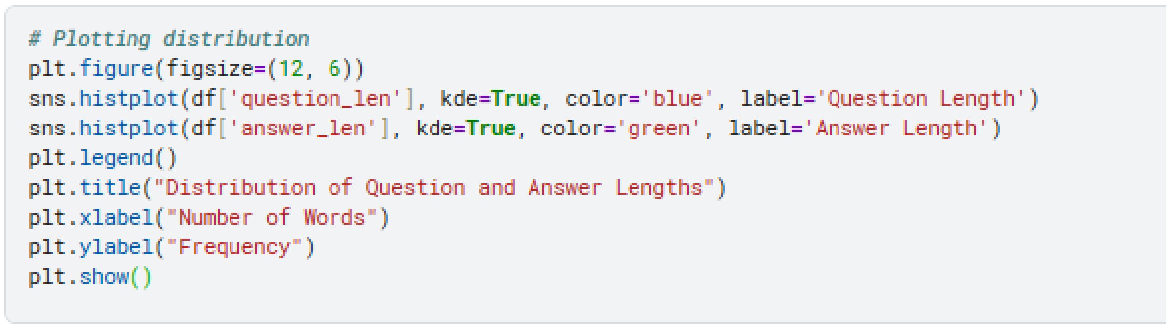
Implementation

Code snippets or pseudocode for main parts of the project

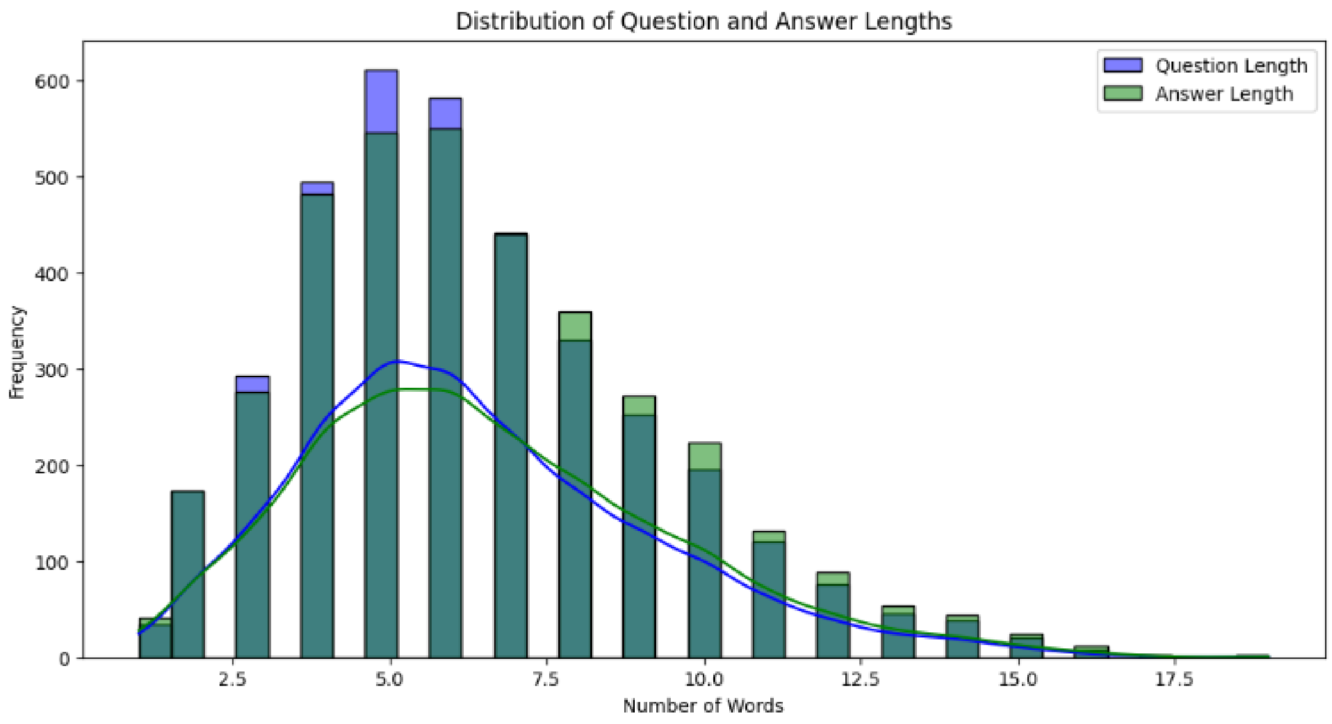


```python
#dictionary to expand common contractions
contractions = {
    "i'm": "i am","you're": "you are","he's": "he is","she's": "she is","it's": "it is","we're": "we are","they're": "they are","i've": "i have",
    "you've": "you have","we've": "we have","they've": "they have","i'll": "i will","you'll": "you will","he'll": "he will","she'll": "she will",
    "it'll": "it will","we'll": "we will","they'll": "they will","i'd": "i would","you'd": "you would","he'd": "he would","she'd": "she would",
    "we'd": "we would","they'd": "they would","can't": "cannot","won't": "will not","don't": "do not","doesn't": "does not","didn't": "did not",
    "isn't": "is not","aren't": "are not","wasn't": "was not","weren't": "were not","wouldn't": "would not","shouldn't": "should not","couldn't": "could not",
    "mustn't": "must not","haven't": "have not","hasn't": "has not","hadn't": "had not","mightn't": "might not","needn't": "need not",
    "shan't": "shall not","let's": "let us","who's": "who is","what's": "what is","where's": "where is","when's": "when is","why's": "why is",
    "how's": "how is","there's": "there is","here's": "here is","that'll": "that will","who'll": "who will","what'll": "what will","y'all": "you all","o'clock": "of the clock"
    "ma'am": "madam","n't": " not","'re": " are","'s": " is","'d": " would","'ll": " will","'t": " not","'ve": " have","'m": " am"
}


# Define a cleaning function
def clean_text(text):
    text = str(text).lower()                             # Lowercase

    for contraction, expanded in contractions.items():   # Expand contractions
        text = text.replace(contraction, expanded)

    text = re.sub(r'[^a-zA-Z\s]', '', text)              # Remove punctuation

    text = re.sub(r'\s+', ' ', text).strip()             # Remove extra spaces

    return text

# Apply it to text columns (assuming 'questions' and 'answers')
df['question_cleaned'] = df['question'].apply(clean_text)
df['answer_cleaned'] = df['answer'].apply(clean_text)

df[['question', 'question_cleaned', 'answer', 'answer_cleaned']].head()
```

Plotting the Histplot for before cleaning the dataset

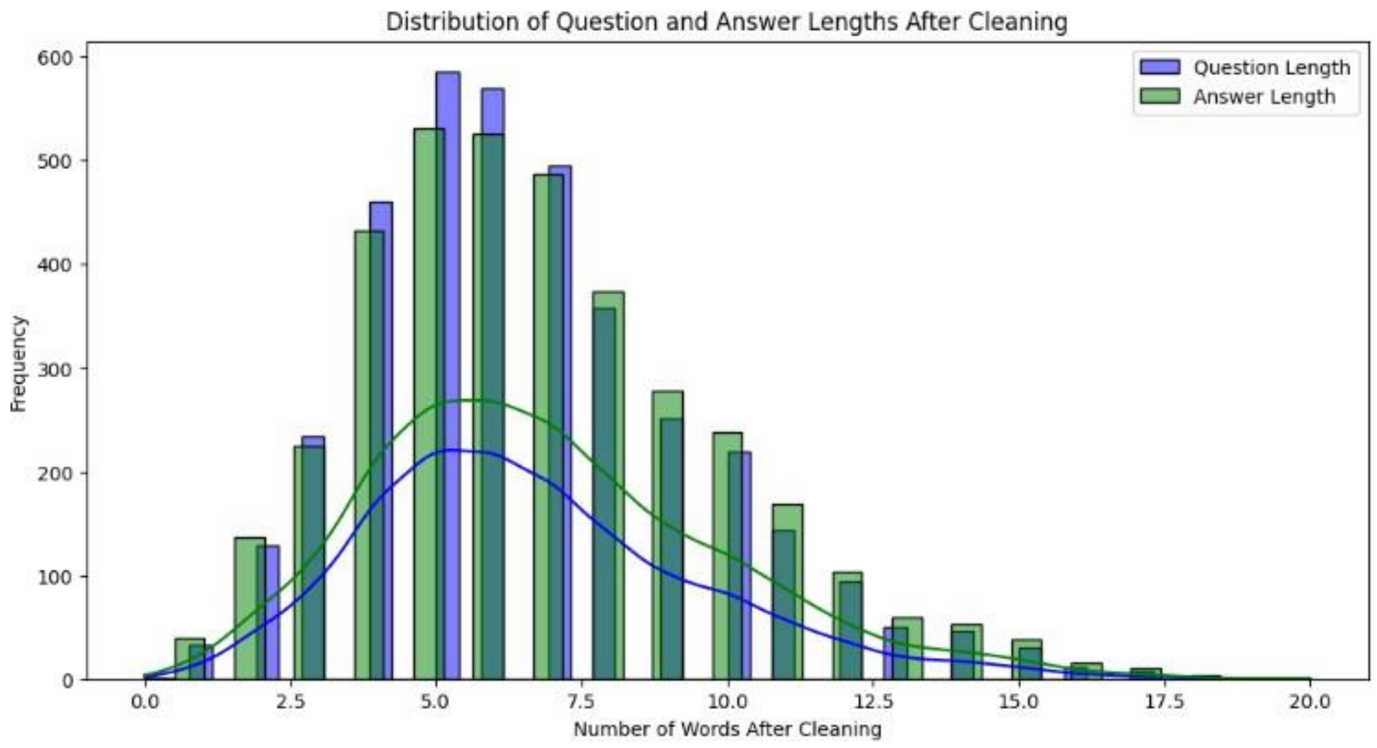


```python
# Plotting distribution
plt.figure(figsize=(12, 6))
sns.histplot(df['question_len'], kde=True, color='blue', label='Question Length')
sns.histplot(df['answer_len'], kde=True, color='green', label='Answer Length')
plt.legend()
plt.title("Distribution of Question and Answer Lengths")
plt.xlabel("Number of Words")
plt.ylabel("Frequency")
plt.show()
```



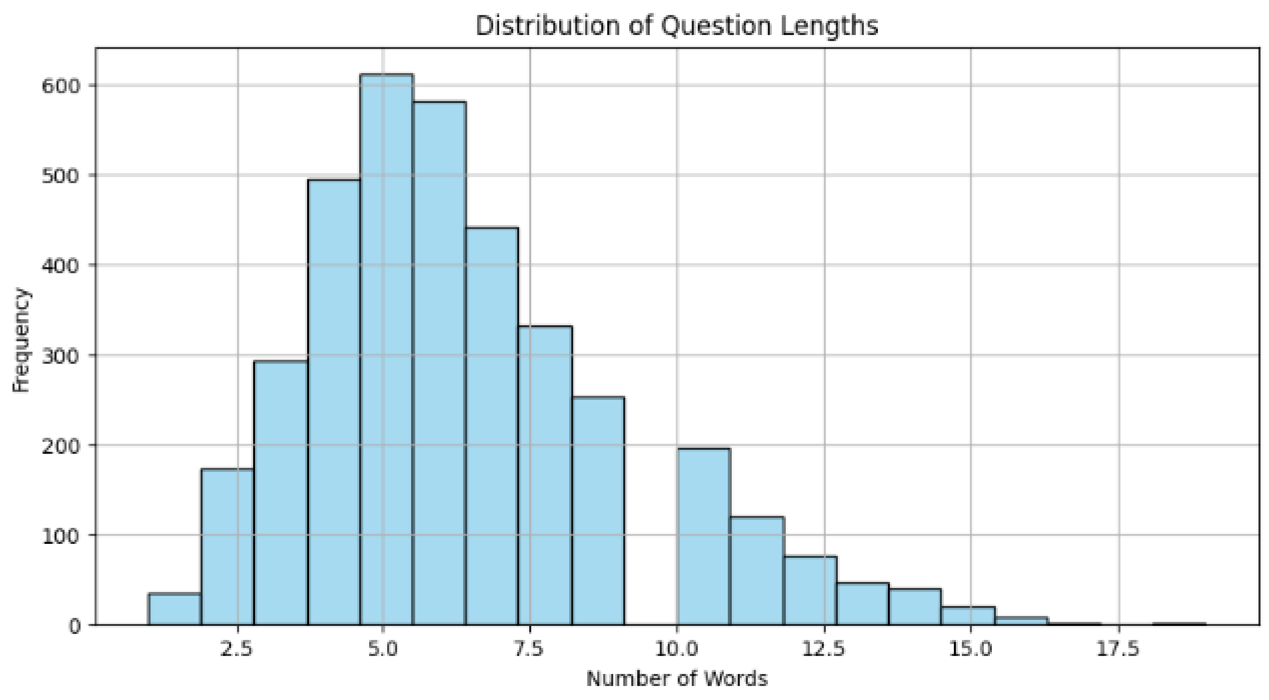
Plotting the histplot after cleaning the dataset

```
# Plotting distribution
plt.figure(figsize=(12, 6))
sns.histplot(df['question_cleaned_len'], kde=True, color='blue', label='Question Length')
sns.histplot(df['answer_cleaned_len'], kde=True, color='green', label='Answer Length')
plt.legend()
plt.title("Distribution of Question and Answer Lengths After Cleaning")
plt.xlabel("Number of Words After Cleaning")
plt.ylabel("Frequency")
plt.show()
```



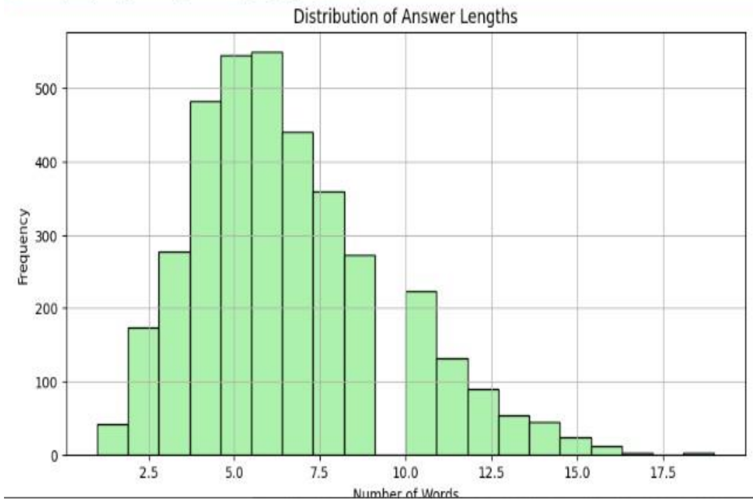
Distribution of Dataset lengths before cleaning

```python
# 1. Distribution of Question Lengths
plt.figure(figsize=(10, 5))
sns.histplot(df['question'].apply(lambda x: len(str(x).split())), bins=20, color='skyblue')
plt.title('Distribution of Question Lengths')
plt.xlabel('Number of Words')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```



```
# 2. Distribution of Answer Lengths
plt.figure(figsize=(10, 5))
sns.histplot(df['answer'].apply(lambda x: len(str(x).split())), bins=20, color='lightgreen')
plt.title('Distribution of Answer Lengths')
plt.xlabel('Number of Words')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```

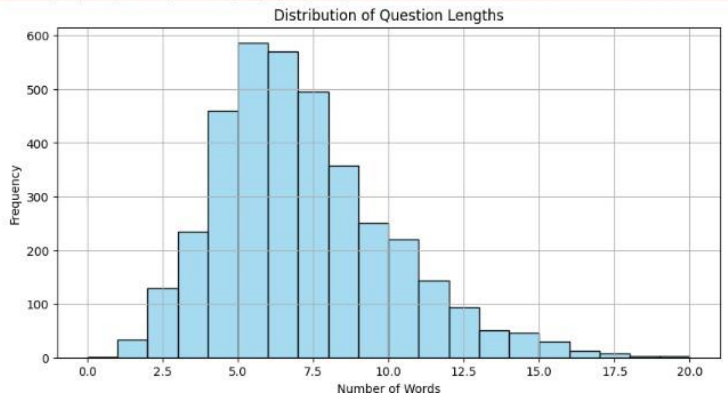
```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and will be removed in a future version. Convert inf values to NaN before operating instea
d.
  with pd.option_context('mode.use_inf_as_na', True):
```



Distribution of data length after cleaning

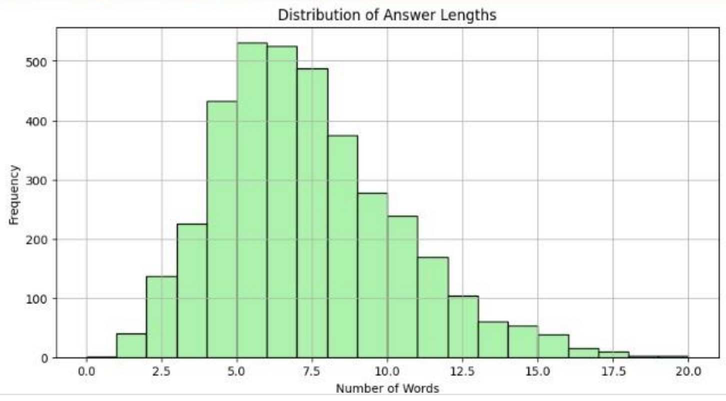
```
# 1. Distribution of Question Lengths
plt.figure(figsize=(10, 5))
sns.histplot(df['question_cleaned'].apply(lambda x: len(str(x).split())), bins=20, color='skyblue')
plt.title('Distribution of Question Lengths')
plt.xlabel('Number of Words')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and will be removed in a future version. Convert inf values to NaN before operating instea
d.
  with pd.option_context('mode.use_inf_as_na', True):
```

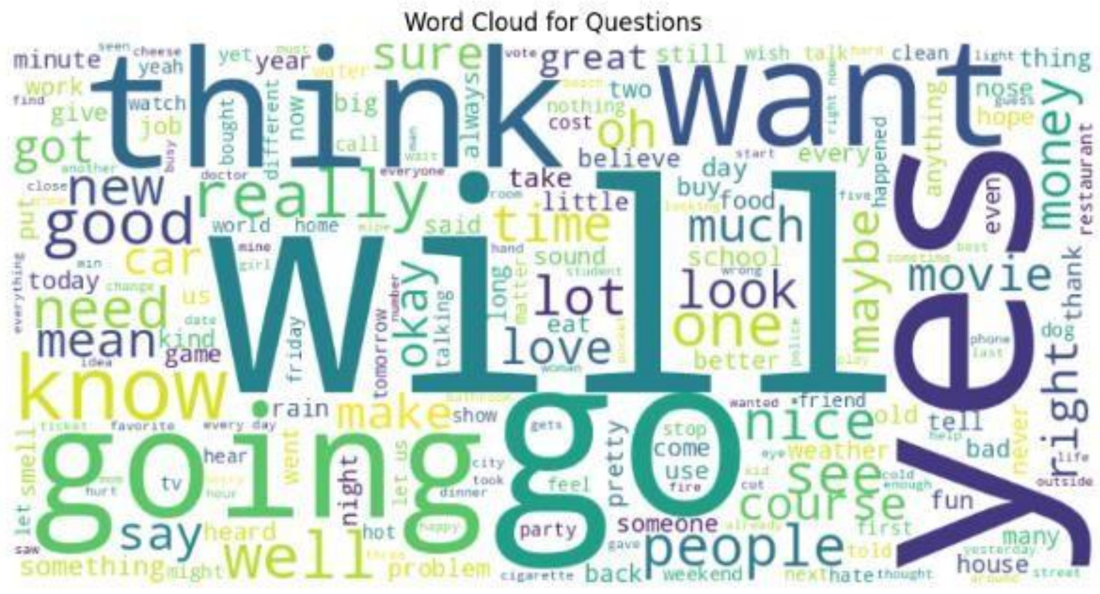


```
# 2. Distribution of Answer Lengths
plt.figure(figsize=(10, 5))
sns.histplot(df['answer_cleaned'].apply(lambda x: len(str(x).split())), bins=20, color='lightgreen')
plt.title('Distribution of Answer Lengths')
plt.xlabel('Number of Words')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/seaborn/_oldcore.py:1119: FutureWarning: use_inf_as_na option is deprecated and will be removed in a future version. Convert inf values to NaN before operating instea
d.
  with pd.option_context('mode.use_inf_as_na', True):
```



```python
# Word Cloud for Questions
wordcloud = WordCloud(width=800, height=400, background_color='white').generate(question_text)
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title("Word Cloud for Questions")
plt.show()
```



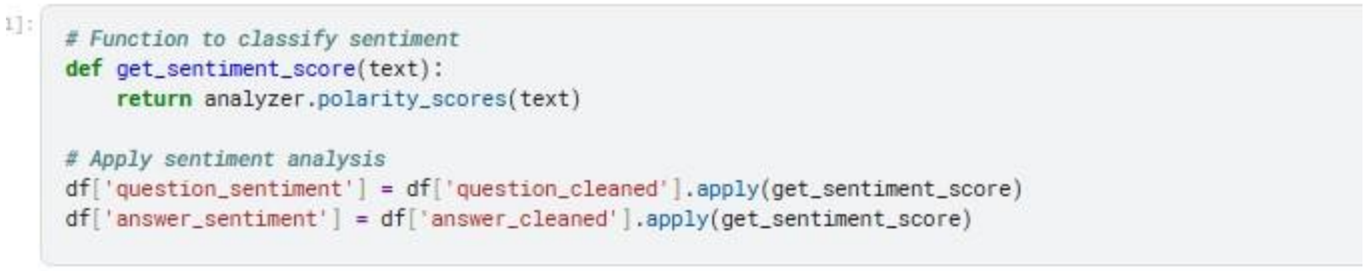
+ Code + Markdown

Applying Tokenization

```python
# Simple whitespace tokenizer
def tokenize_text(text):
    return text.split()

# Apply tokenization to cleaned questions and answers
df['question_tokens'] = df['question_cleaned'].apply(tokenize_text)
df['answer_tokens'] = df['answer_cleaned'].apply(tokenize_text)

# Display a sample
df[['question_cleaned', 'question_tokens', 'answer_cleaned', 'answer_tokens']].head()
```

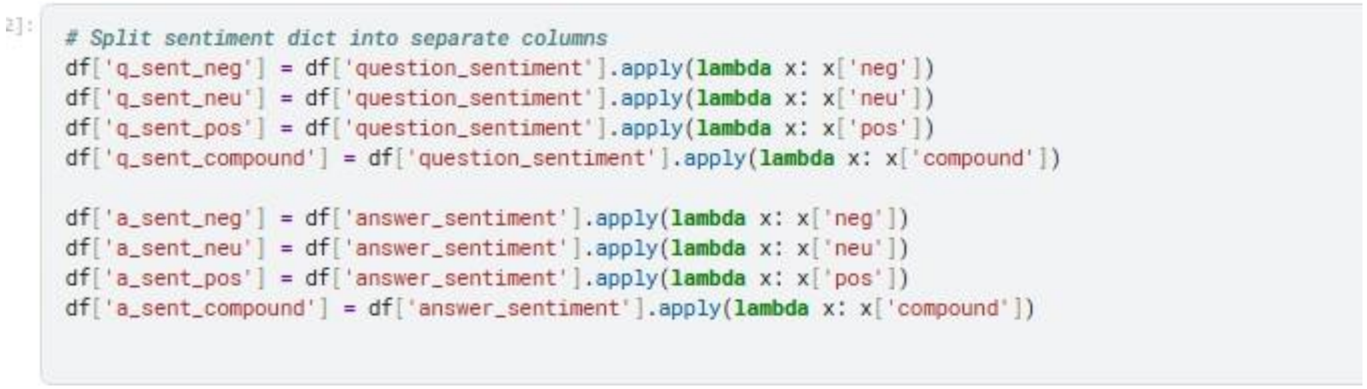
	question_cleaned	question_tokens	answer_cleaned	answer_tokens
0	i am fine how about yourself	[i, am, fine, how, about, yourself]	i am pretty good thanks for asking	[i, am, pretty, good, thanks, for, asking]
1	i am pretty good thanks for asking	[i, am, pretty, good, thanks, for, asking]	no problem so how have you been	[no, problem, so, how, have, you, been]
2	no problem so how have you been	[no, problem, so, how, have, you, been]	i have been great what about you	[i, have, been, great, what, about, you]
3	i have been great what about you	[i, have, been, great, what, about, you]	i have been good i am in school right now	[i, have, been, good, i, am, in, school, right...
4	i have been good i am in school right now	[i, have, been, good, i, am, in, school, right...	what school do you go to	[what, school, do, you, go, to]

Step 3: Apply sentiment analysis on cleaned questions and answers



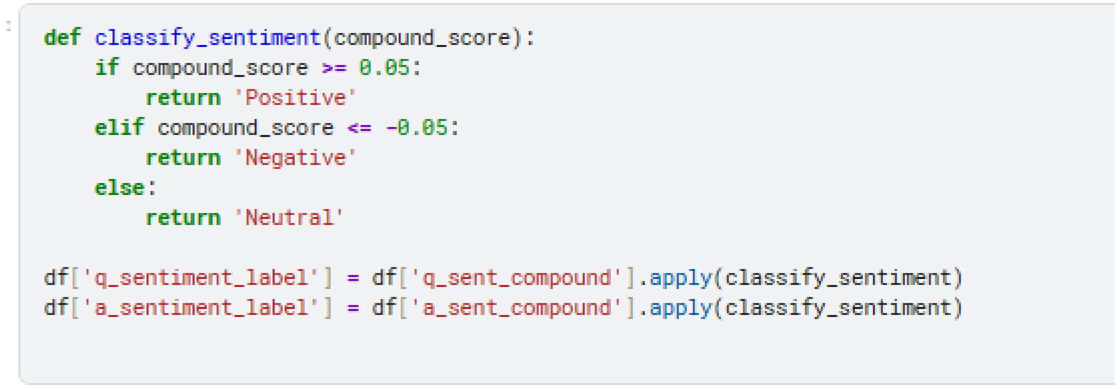
```python
# Function to classify sentiment
def get_sentiment_score(text):
    return analyzer.polarity_scores(text)

# Apply sentiment analysis
df['question_sentiment'] = df['question_cleaned'].apply(get_sentiment_score)
df['answer_sentiment'] = df['answer_cleaned'].apply(get_sentiment_score)
```

Step 4: Extract sentiment scores into separate columns



```python
# Split sentiment dict into separate columns
df['q_sent_neg'] = df['question_sentiment'].apply(lambda x: x['neg'])
df['q_sent_neu'] = df['question_sentiment'].apply(lambda x: x['neu'])
df['q_sent_pos'] = df['question_sentiment'].apply(lambda x: x['pos'])
df['q_sent_compound'] = df['question_sentiment'].apply(lambda x: x['compound'])

df['a_sent_neg'] = df['answer_sentiment'].apply(lambda x: x['neg'])
df['a_sent_neu'] = df['answer_sentiment'].apply(lambda x: x['neu'])
df['a_sent_pos'] = df['answer_sentiment'].apply(lambda x: x['pos'])
df['a_sent_compound'] = df['answer_sentiment'].apply(lambda x: x['compound'])
```

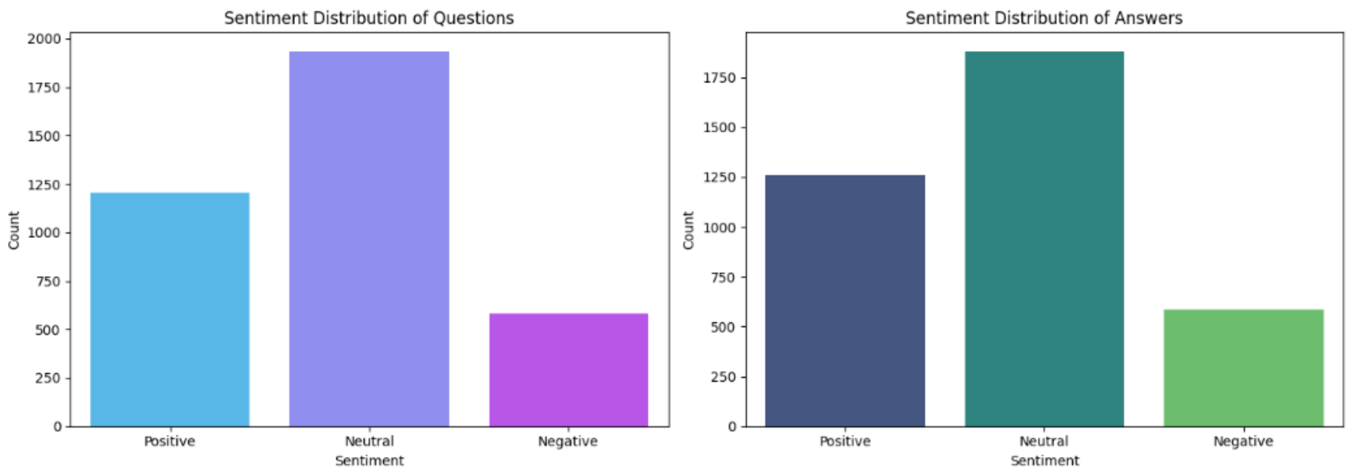
Step 5: Classify sentiment (Positive, Negative, Neutral)



```python
def classify_sentiment(compound_score):
    if compound_score >= 0.05:
        return 'Positive'
    elif compound_score <= -0.05:
        return 'Negative'
    else:
        return 'Neutral'

df['q_sentiment_label'] = df['q_sent_compound'].apply(classify_sentiment)
df['a_sentiment_label'] = df['a_sent_compound'].apply(classify_sentiment)
```

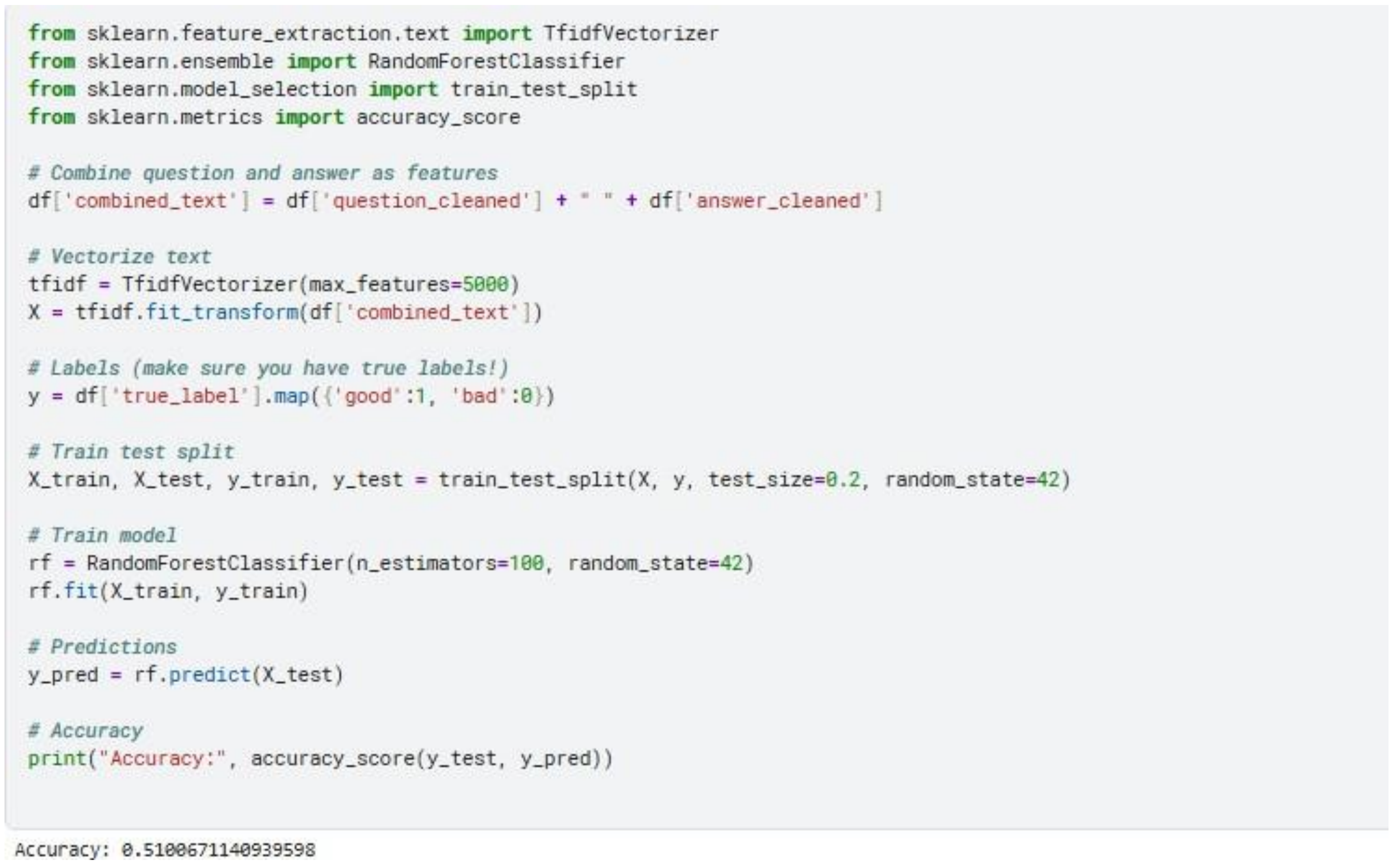
Step 6: Visualize sentiment distribution

```
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(14, 5))

# Questions
plt.subplot(1, 2, 1)
sns.countplot(data=df, x='q_sentiment_label', palette='cool')
plt.title("Sentiment Distribution of Questions")
plt.xlabel("Sentiment")
plt.ylabel("Count")

# Answers
plt.subplot(1, 2, 2)
sns.countplot(data=df, x='a_sentiment_label', palette='viridis')
plt.title("Sentiment Distribution of Answers")
plt.xlabel("Sentiment")
plt.ylabel("Count")

plt.tight_layout()
plt.show()
```



Models and their Output

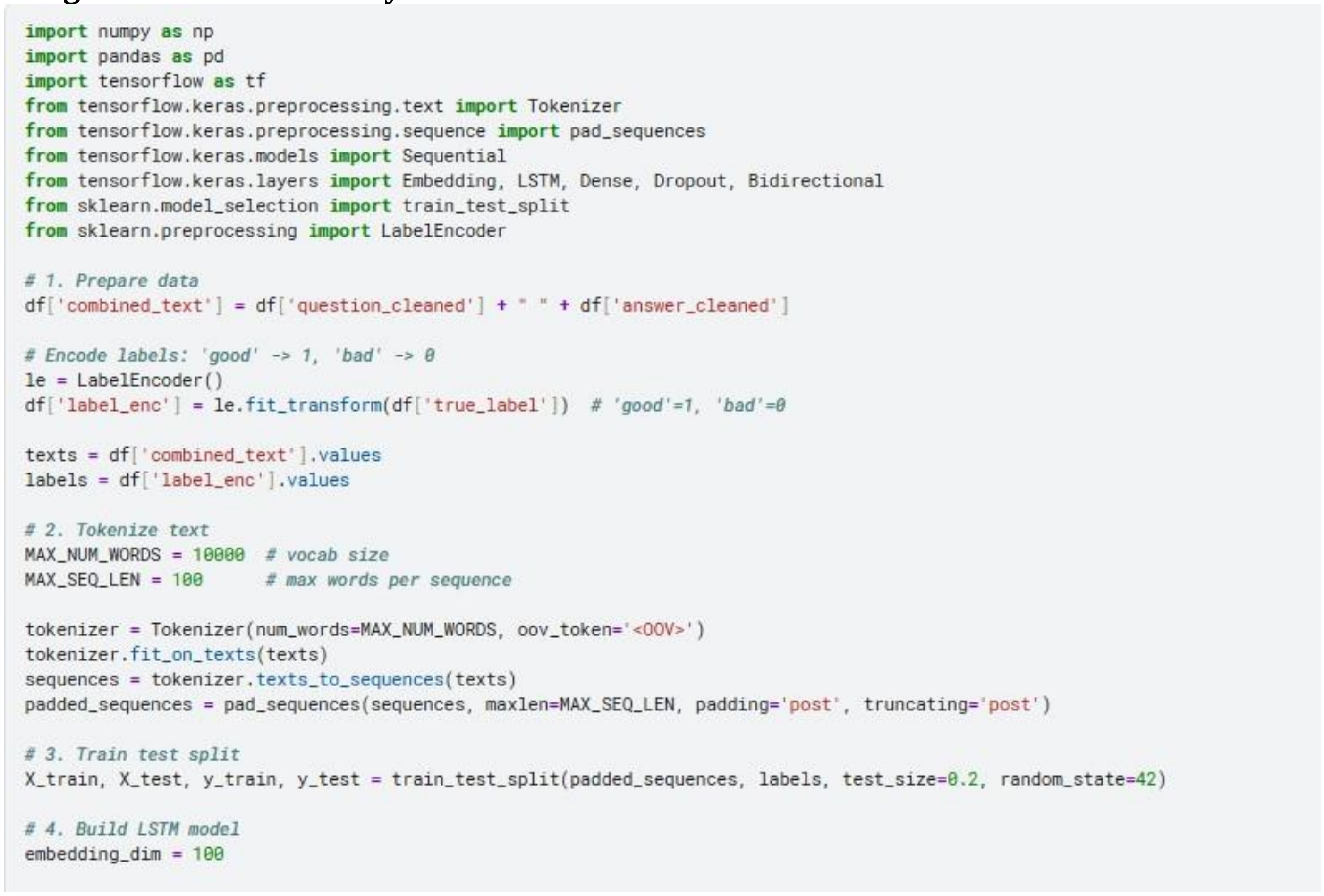
Random Forest



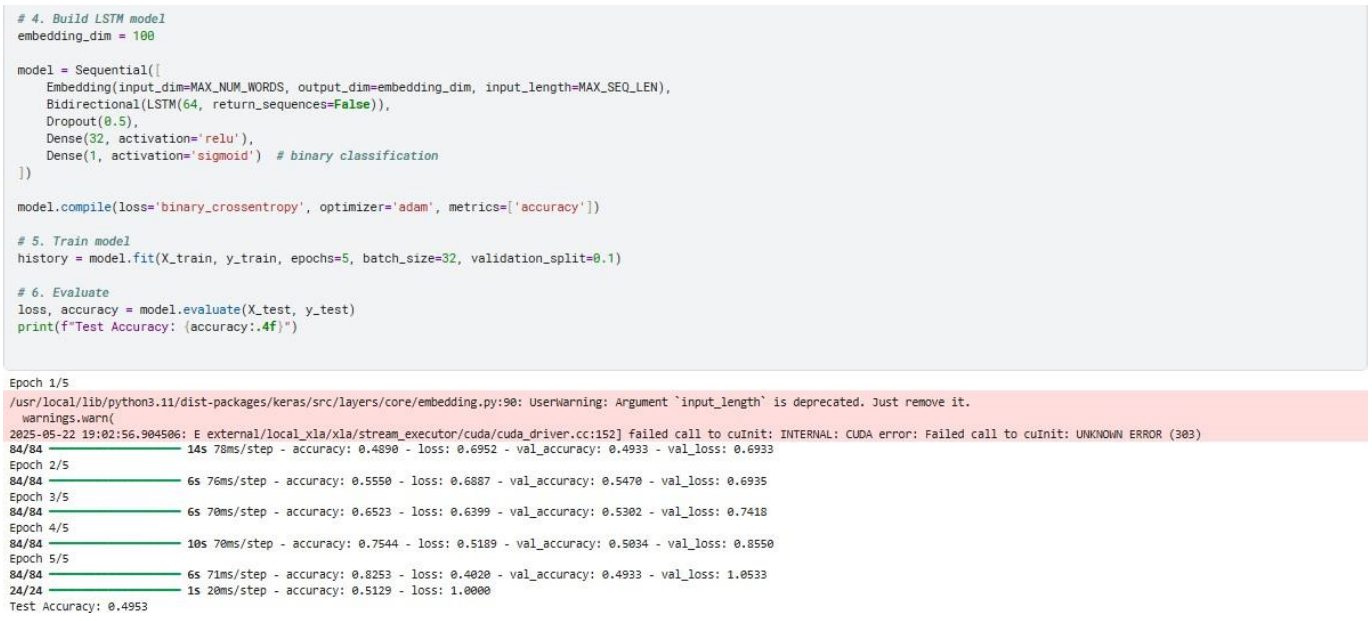
```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Combine question and answer as features
df['combined_text'] = df['question_cleaned'] + " " + df['answer_cleaned']

# Vectorize text
tfidf = TfidfVectorizer(max_features=5000)
X = tfidf.fit_transform(df['combined_text'])

# Labels (make sure you have true labels!)
y = df['true_label'].map({'good':1, 'bad':0})

# Train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train model
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

# Predictions
y_pred = rf.predict(X_test)

# Accuracy
print("Accuracy:", accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.5100671140939598
```

Long Short Term Memory



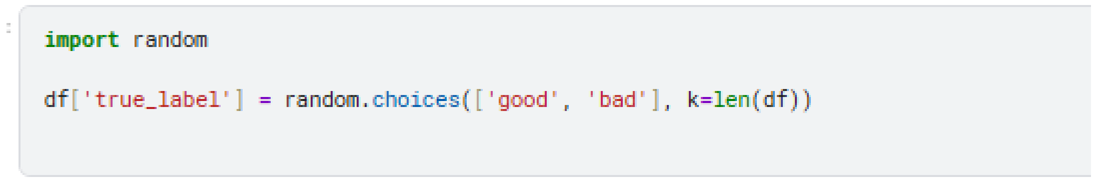
```python
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout, Bidirectional
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# 1. Prepare data
df['combined_text'] = df['question_cleaned'] + " " + df['answer_cleaned']

# Encode labels: 'good' -> 1, 'bad' -> 0
le = LabelEncoder()
df['label_enc'] = le.fit_transform(df['true_label'])  # 'good'=1, 'bad'=0

texts = df['combined_text'].values
labels = df['label_enc'].values

# 2. Tokenize text
MAX_NUM_WORDS = 10000  # vocab size
MAX_SEQ_LEN = 100      # max words per sequence

tokenizer = Tokenizer(num_words=MAX_NUM_WORDS, oov_token='<OOV>')
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
padded_sequences = pad_sequences(sequences, maxlen=MAX_SEQ_LEN, padding='post', truncating='post')

# 3. Train test split
X_train, X_test, y_train, y_test = train_test_split(padded_sequences, labels, test_size=0.2, random_state=42)

# 4. Build LSTM model
embedding_dim = 100
```



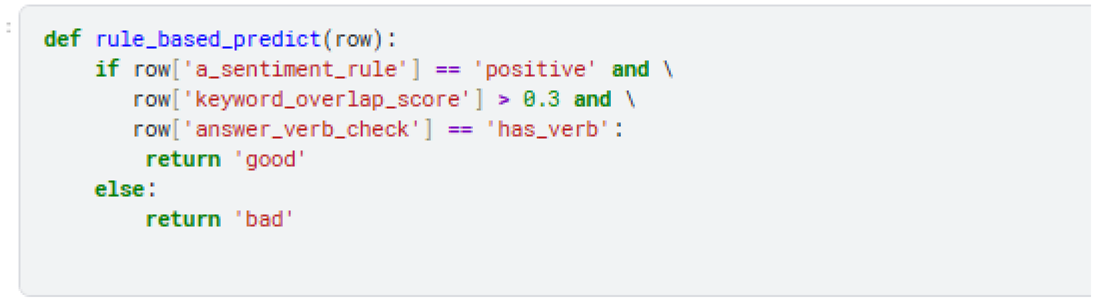
```python
# 4. Build LSTM model
embedding_dim = 100

model = Sequential([
    Embedding(input_dim=MAX_NUM_WORDS, output_dim=embedding_dim, input_length=MAX_SEQ_LEN),
    Bidirectional(LSTM(64, return_sequences=False)),
    Dropout(0.5),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')  # binary classification
])

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

# 5. Train model
history = model.fit(X_train, y_train, epochs=5, batch_size=32, validation_split=0.1)

# 6. Evaluate
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy:.4f}")
```

```
Epoch 1/5
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
2025-05-22 19:02:56.904506: E external/local_xla/xla/stream_executor/cuda/cuda_driver.cc:152] failed call to cuInit: INTERNAL: CUDA error: Failed call to cuInit: UNKNOWN ERROR (303)
84/84 ━━━━━━━━━━━━━━━━━━━━ 14s 78ms/step - accuracy: 0.4890 - loss: 0.6952 - val_accuracy: 0.4933 - val_loss: 0.6933
Epoch 2/5
84/84 ━━━━━━━━━━━━━━━━━━━━ 6s 76ms/step - accuracy: 0.5550 - loss: 0.6887 - val_accuracy: 0.5470 - val_loss: 0.6935
Epoch 3/5
84/84 ━━━━━━━━━━━━━━━━━━━━ 6s 70ms/step - accuracy: 0.6523 - loss: 0.6399 - val_accuracy: 0.5302 - val_loss: 0.7418
Epoch 4/5
84/84 ━━━━━━━━━━━━━━━━━━━━ 10s 70ms/step - accuracy: 0.7544 - loss: 0.5189 - val_accuracy: 0.5034 - val_loss: 0.8550
Epoch 5/5
84/84 ━━━━━━━━━━━━━━━━━━━━ 6s 71ms/step - accuracy: 0.8253 - loss: 0.4020 - val_accuracy: 0.4933 - val_loss: 1.0533
24/24 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.5129 - loss: 1.0000
Test Accuracy: 0.4953
```

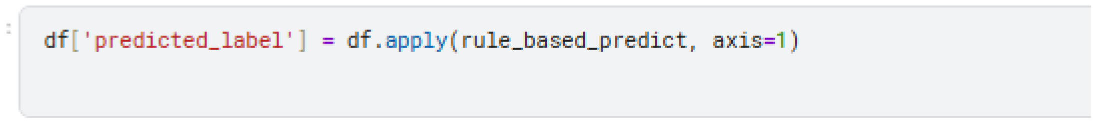
Chatbot Response Analyze



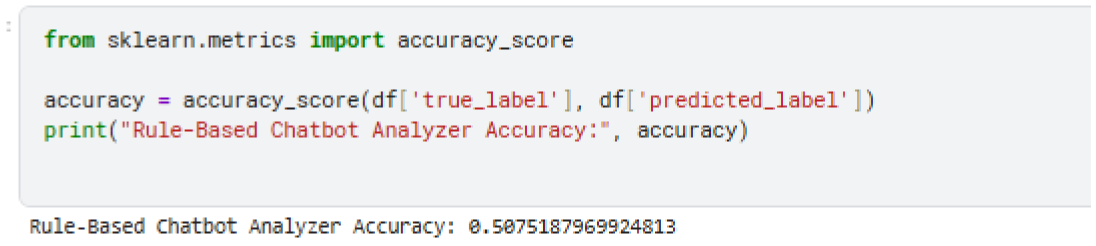
```python
import random

df['true_label'] = random.choices(['good', 'bad'], k=len(df))
```



```python
def rule_based_predict(row):
    if row['a_sentiment_rule'] == 'positive' and \
       row['keyword_overlap_score'] > 0.3 and \
       row['answer_verb_check'] == 'has_verb':
        return 'good'
    else:
        return 'bad'
```

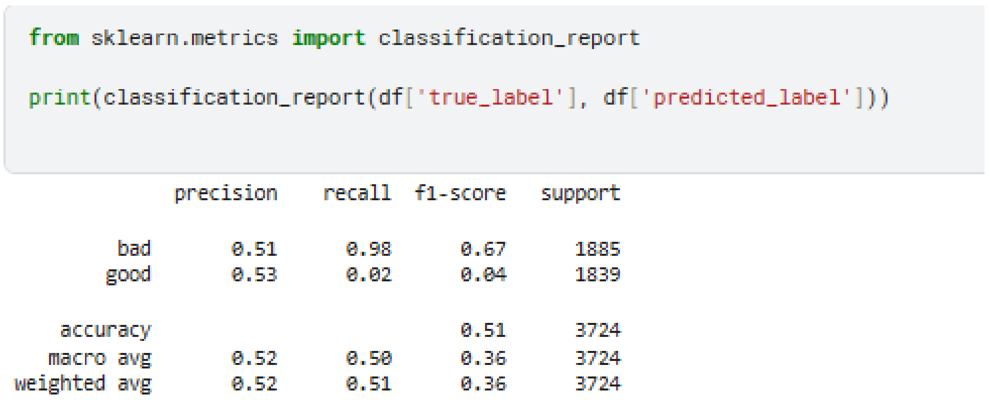


```python
df['predicted_label'] = df.apply(rule_based_predict, axis=1)
```



```python
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(df['true_label'], df['predicted_label'])
print("Rule-Based Chatbot Analyzer Accuracy:", accuracy)
```

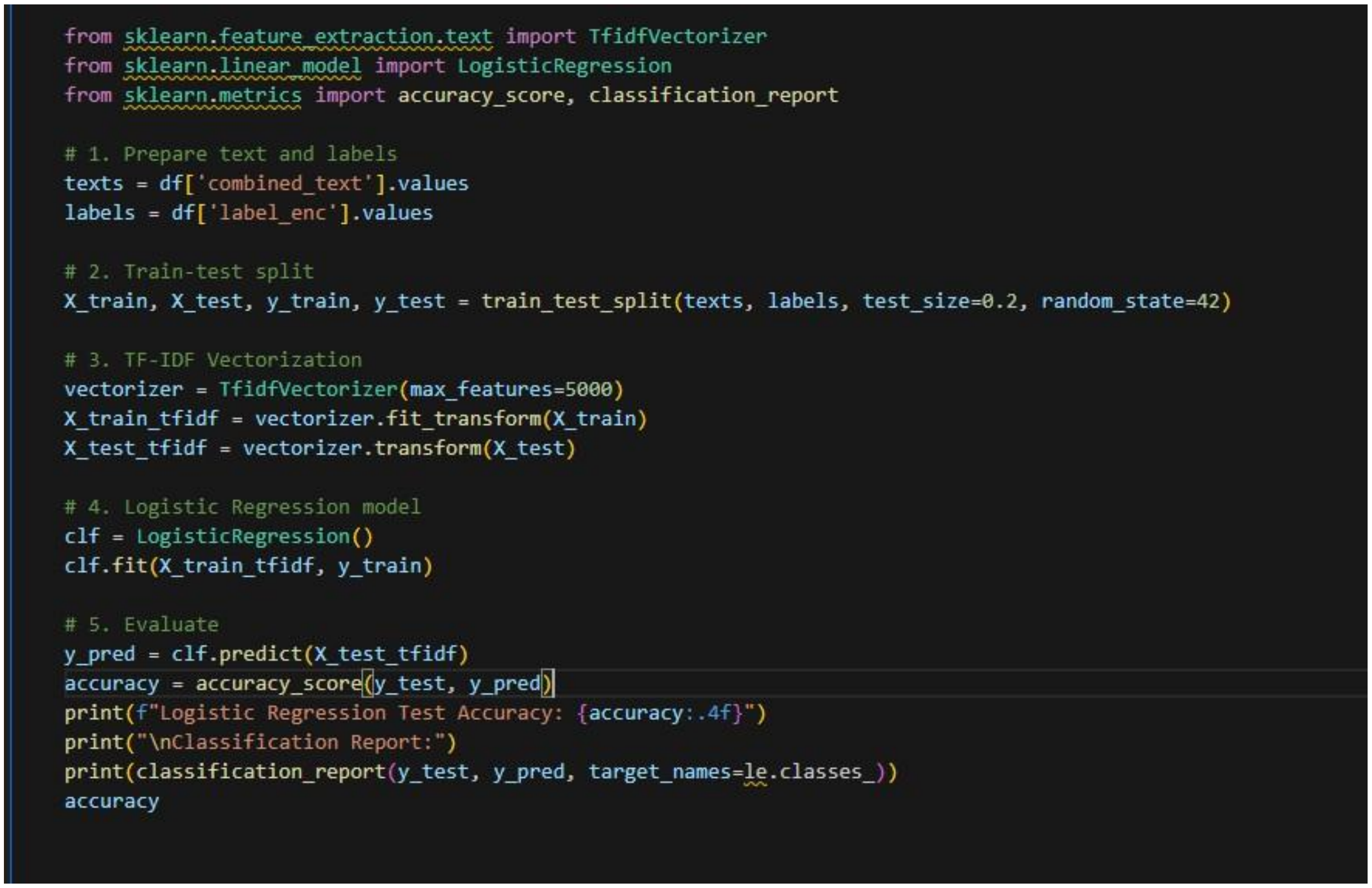
```
Rule-Based Chatbot Analyzer Accuracy: 0.5075187969924813
```

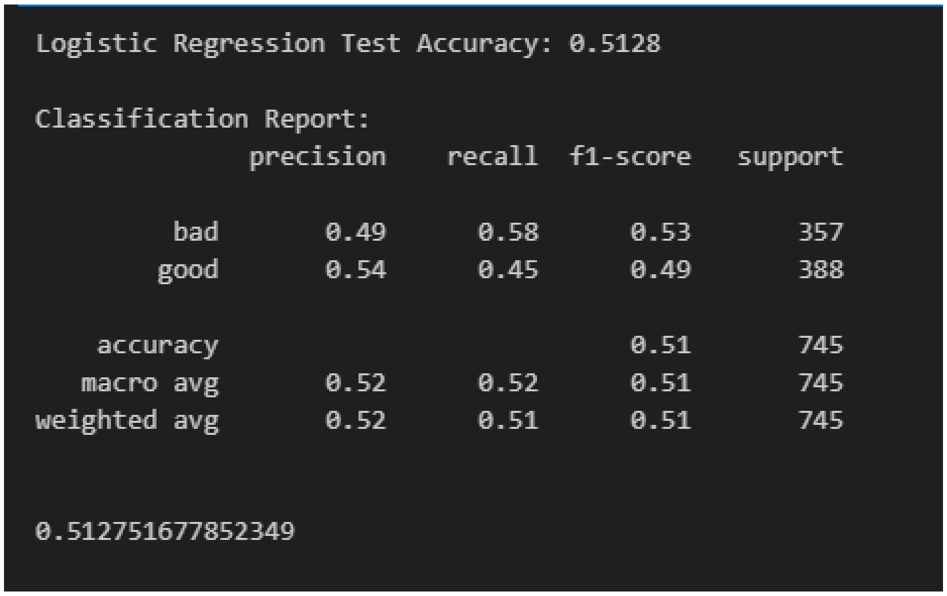


```python
from sklearn.metrics import classification_report

print(classification_report(df['true_label'], df['predicted_label']))
```

```
              precision    recall  f1-score   support

         bad       0.51      0.98      0.67      1885
        good       0.53      0.02      0.04      1839

    accuracy                           0.51      3724
   macro avg       0.52      0.50      0.36      3724
weighted avg       0.52      0.51      0.36      3724
```

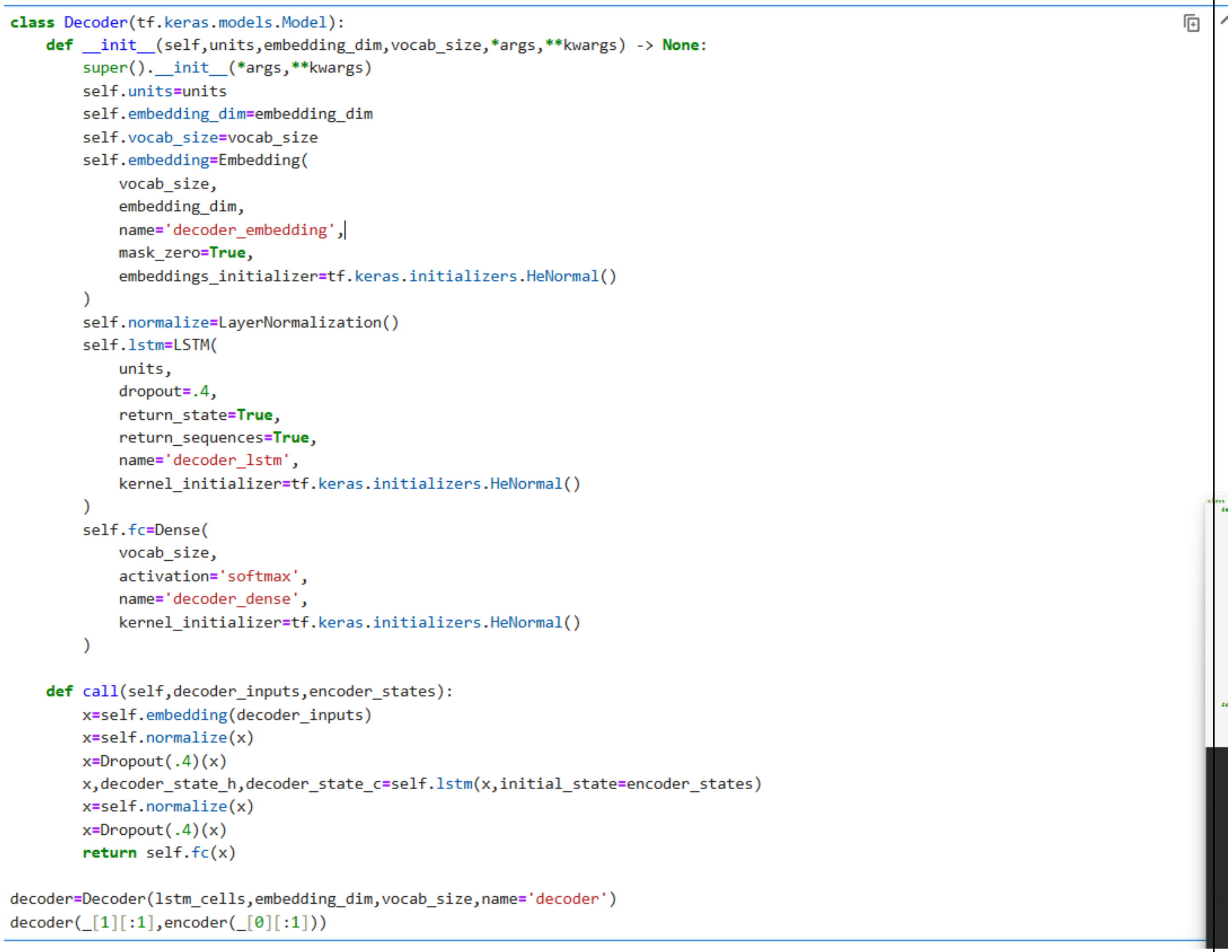
Logistic Regression


```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

# 1. Prepare text and labels
texts = df['combined_text'].values
labels = df['label_enc'].values

# 2. Train-test split
X_train, X_test, y_train, y_test = train_test_split(texts, labels, test_size=0.2, random_state=42)

# 3. TF-IDF Vectorization
vectorizer = TfidfVectorizer(max_features=5000)
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)

# 4. Logistic Regression model
clf = LogisticRegression()
clf.fit(X_train_tfidf, y_train)

# 5. Evaluate
y_pred = clf.predict(X_test_tfidf)
accuracy = accuracy_score(y_test, y_pred)
print(f"Logistic Regression Test Accuracy: {accuracy:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=le.classes_))
accuracy
```

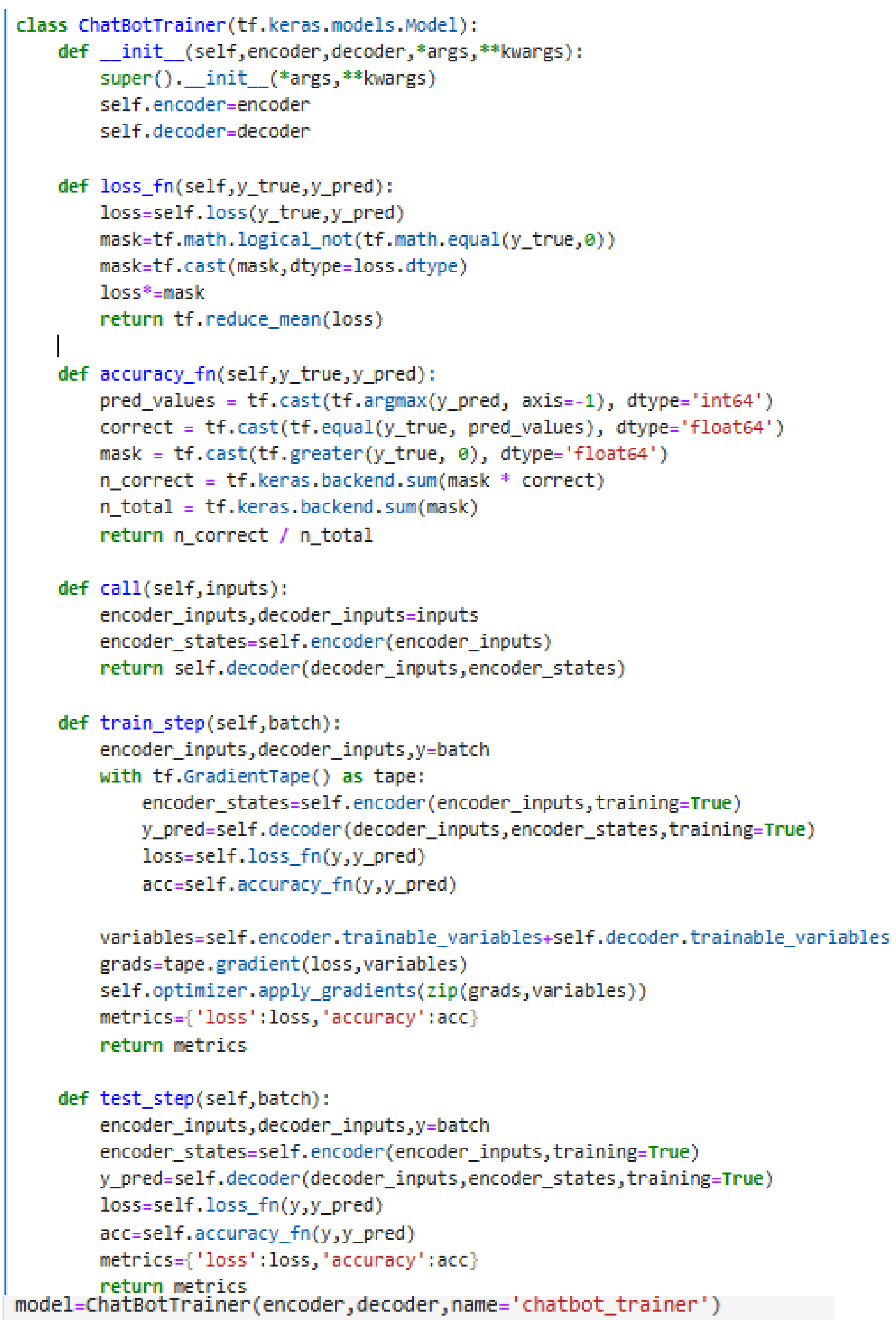

```
Logistic Regression Test Accuracy: 0.5128

Classification Report:
              precision    recall  f1-score   support

         bad       0.49      0.58      0.53       357
        good       0.54      0.45      0.49       388

    accuracy                           0.51       745
   macro avg       0.52      0.52      0.51       745
weighted avg       0.52      0.51      0.51       745


0.512751677852349
```

Transformers

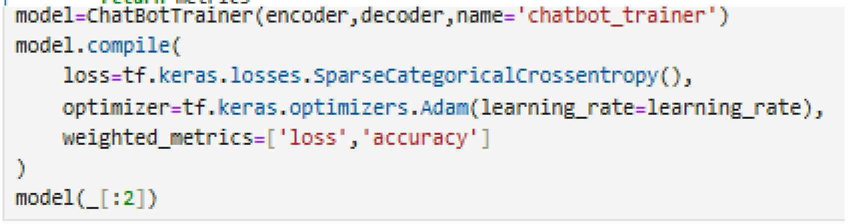
```
class Encoder(tf.keras.models.Model):
    def __init__(self,units,embedding_dim,vocab_size,*args,**kwargs) -> None:
        super().__init__(*args,**kwargs)
        self.units=units
        self.vocab_size=vocab_size
        self.embedding_dim=embedding_dim
        self.embedding=Embedding(
            vocab_size,
            embedding_dim,
            name='encoder_embedding',
            mask_zero=True,
            embeddings_initializer=tf.keras.initializers.GlorotNormal()
        )
        self.normalize=LayerNormalization()
        self.lstm=LSTM(
            units,
            dropout=.4,
            return_state=True,
            return_sequences=True,
            name='encoder_lstm',
            kernel_initializer=tf.keras.initializers.GlorotNormal()
        )

    def call(self,encoder_inputs):
        self.inputs=encoder_inputs
        x=self.embedding(encoder_inputs)
        x=self.normalize(x)
        x=Dropout(.4)(x)
        encoder_outputs,encoder_state_h,encoder_state_c=self.lstm(x)
        self.outputs=[encoder_state_h,encoder_state_c]
        return encoder_state_h,encoder_state_c

encoder=Encoder(lstm_cells,embedding_dim,vocab_size,name='encoder')
encoder.call(_[0])
```



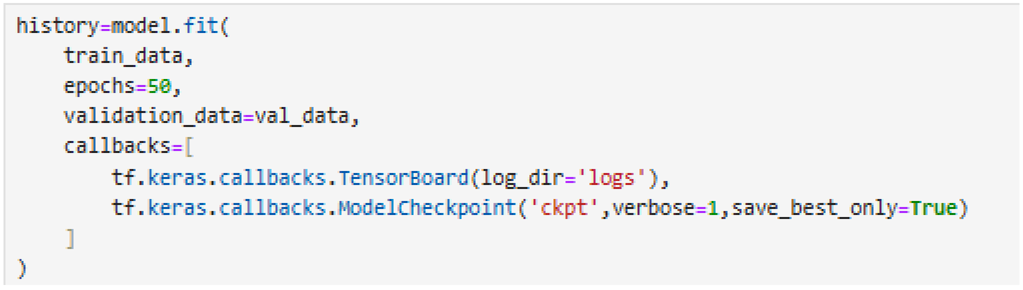
```python
class Decoder(tf.keras.models.Model):
    def __init__(self,units,embedding_dim,vocab_size,*args,**kwargs) -> None:
        super().__init__(*args,**kwargs)
        self.units=units
        self.embedding_dim=embedding_dim
        self.vocab_size=vocab_size
        self.embedding=Embedding(
            vocab_size,
            embedding_dim,
            name='decoder_embedding',
            mask_zero=True,
            embeddings_initializer=tf.keras.initializers.HeNormal()
        )
        self.normalize=LayerNormalization()
        self.lstm=LSTM(
            units,
            dropout=.4,
            return_state=True,
            return_sequences=True,
            name='decoder_lstm',
            kernel_initializer=tf.keras.initializers.HeNormal()
        )
        self.fc=Dense(
            vocab_size,
            activation='softmax',
            name='decoder_dense',
            kernel_initializer=tf.keras.initializers.HeNormal()
        )

    def call(self,decoder_inputs,encoder_states):
        x=self.embedding(decoder_inputs)
        x=self.normalize(x)
        x=Dropout(.4)(x)
        x,decoder_state_h,decoder_state_c=self.lstm(x,initial_state=encoder_states)
        x=self.normalize(x)
        x=Dropout(.4)(x)
        return self.fc(x)

decoder=Decoder(lstm_cells,embedding_dim,vocab_size,name='decoder')
decoder(_[1][:1],encoder(_[0][:1]))
```



```python
class ChatBotTrainer(tf.keras.models.Model):
    def __init__(self,encoder,decoder,*args,**kwargs):
        super().__init__(*args,**kwargs)
        self.encoder=encoder
        self.decoder=decoder

    def loss_fn(self,y_true,y_pred):
        loss=self.loss(y_true,y_pred)
        mask=tf.math.logical_not(tf.math.equal(y_true,0))
        mask=tf.cast(mask,dtype=loss.dtype)
        loss*=mask
        return tf.reduce_mean(loss)

    def accuracy_fn(self,y_true,y_pred):
        pred_values = tf.cast(tf.argmax(y_pred, axis=-1), dtype='int64')
        correct = tf.cast(tf.equal(y_true, pred_values), dtype='float64')
        mask = tf.cast(tf.greater(y_true, 0), dtype='float64')
        n_correct = tf.keras.backend.sum(mask * correct)
        n_total = tf.keras.backend.sum(mask)
        return n_correct / n_total

    def call(self,inputs):
        encoder_inputs,decoder_inputs=inputs
        encoder_states=self.encoder(encoder_inputs)
        return self.decoder(decoder_inputs,encoder_states)

    def train_step(self,batch):
        encoder_inputs,decoder_inputs,y=batch
        with tf.GradientTape() as tape:
            encoder_states=self.encoder(encoder_inputs,training=True)
            y_pred=self.decoder(decoder_inputs,encoder_states,training=True)
            loss=self.loss_fn(y,y_pred)
            acc=self.accuracy_fn(y,y_pred)

        variables=self.encoder.trainable_variables+self.decoder.trainable_variables
        grads=tape.gradient(loss,variables)
        self.optimizer.apply_gradients(zip(grads,variables))
        metrics={'loss':loss,'accuracy':acc}
        return metrics

    def test_step(self,batch):
        encoder_inputs,decoder_inputs,y=batch
        encoder_states=self.encoder(encoder_inputs,training=True)
        y_pred=self.decoder(decoder_inputs,encoder_states,training=True)
        loss=self.loss_fn(y,y_pred)
        acc=self.accuracy_fn(y,y_pred)
        metrics={'loss':loss,'accuracy':acc}
        return metrics
```



```python
model=ChatBotTrainer(encoder,decoder,name='chatbot_trainer')
model.compile(
    loss=tf.keras.losses.SparseCategoricalCrossentropy(),
    optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
    weighted_metrics=['loss','accuracy']
)
model(_[:2])
```



```python
history=model.fit(
    train_data,
    epochs=50,
    validation_data=val_data,
    callbacks=[
        tf.keras.callbacks.TensorBoard(log_dir='logs'),
        tf.keras.callbacks.ModelCheckpoint('ckpt',verbose=1,save_best_only=True)
    ]
)
```

```
Epoch 48/50
23/23 [==============================] - ETA: 0s - loss: 0.4493 - accuracy: 0.5995
Epoch 48: val_loss did not improve from 0.47323
23/23 [==============================] - 19s 840ms/step - loss: 0.4480 - accuracy: 0.6000 - val_loss: 0.5297 - val_accuracy: 0.5759
Epoch 49/50
23/23 [==============================] - ETA: 0s - loss: 0.4443 - accuracy: 0.6022
Epoch 49: val_loss did not improve from 0.47323
23/23 [==============================] - 19s 820ms/step - loss: 0.4472 - accuracy: 0.6002 - val_loss: 0.5395 - val_accuracy: 0.5674
Epoch 50/50
23/23 [==============================] - ETA: 0s - loss: 0.4441 - accuracy: 0.6027
Epoch 50: val_loss did not improve from 0.47323
23/23 [==============================] - 19s 824ms/step - loss: 0.4433 - accuracy: 0.6031 - val_loss: 0.5628 - val_accuracy: 0.5849
```

Explanation of how the code works:

1. Text Preprocessing

- Preprocessing steps included:
  - Lowercasing text
  - Removing punctuation and special characters
  - Tokenization

---

2. Feature Engineering

- Several features were extracted from the cleaned text:
  - keyword_overlap: Measures the number of overlapping keywords between question and answer.
  - pos_check: Checks for the presence of verbs in answers to ensure sentence completeness.

---

3. Rule-Based Chatbot Evaluation

- A scoring system compares the user input with the stored dataset questions using:
  - Keyword overlap
  - Sentiment alignment
- The best-matching answer is selected based on the highest overlap score.

---

4. Machine Learning Models

- Classification models were trained to label chatbot responses as "good" or "bad":
  - Logistic Regression
  - Random Forest
- The dataset was split into training and testing sets using train_test_split.
- Accuracy and confusion matrix were used as evaluation metrics.

5. LSTM-Based Deep Learning Model

- Tokenized questions and answers were padded to equal length.
- An LSTM model was built with:
    - Embedding layer
    - LSTM layer
    - Dense output layer
- The model was trained on the processed data to predict the quality of response.

Challenges faced and how they were addressed:

1. **Slow Performance of Rule-Based Chatbot**
The rule-based chatbot initially took a long time to respond due to inefficient iterations over the entire dataset. This was addressed by optimizing the keyword matching logic and limiting the number of candidate comparisons.

2. **Low Accuracy (~50%)**
The basic rule-based and logistic regression models provided only about 50% accuracy. To improve this, more features were engineered and advanced models like LSTM were introduced, which better captured the context and semantics of the conversation.

3. **Sparse and Noisy Dataset**
The dataset contained several irrelevant or inconsistent entries, affecting the model's learning. A comprehensive text preprocessing pipeline was applied, including cleaning, stopword removal, and normalization to enhance data quality.

4. **Inconsistent Input Lengths for Deep Learning**
Questions and answers varied widely in length, creating input mismatches for neural networks. This was resolved by using tokenization followed by padding to ensure uniform input dimensions for LSTM models.

5. **Difficulty in Evaluating Chatbot Responses**
Determining whether a chatbot response was "good" or "bad" involved subjective judgment. To manage this, a rule-based annotation system combined with manual verification was used to create ground truth labels for evaluation.

References:

Kaggle Dataset

- ***Dataset for Chatbot*** – Retrieved from: **https://www.kaggle.com/datasets/grafstor/simple-dialogs-for-chatbot**
- **Description:** Contains question-answer pairs suitable for training rule-based or ML-powered chatbot system.

Work Done by Team Mates

SHIVANAND GUPTA | 22SCSE1012610

Collected and prepared the dataset from Kaggle.
Perfomed Exploratory Data Analysis.

SHIVANG KAKKAR | 22SCSE1180121

Built and trained a **Logistic Regression** classifier and CNN Model as a baseline model for performance benchmarking.
Conducted performance evaluation using accuracy, precision, and confusion matrix.

SHIVANGI SEHGAL | 22SCSE1180097

Performed **Exploratory Data Analysis (EDA)** to understand dataset distribution.
Trained and evaluated the **Random Forest Classifier** to distinguish between good and bad responses.
Created visualizations including heatmaps to analyze model performance and feature correlations.
Implemented rule-based logic to evaluate chatbot responses using keyword overlap and sentiment analysis.

SHREYASH UPADHYAY | 23SCSE1012304

Implemented a sequence-to-sequence chatbot model with an Embedding layer, an **LSTM encoder–decoder**, and a Dense softmax output layer.
Conducted performance evaluation using accuracy, precision, and confusion matrix.

Conclusion

This project successfully developed and analyzed a rule-based and neural network-enhanced chatbot designed for customer service scenarios. The chatbot was built using a dataset from Kaggle and leveraged various Natural Language Processing (NLP) techniques such as sentiment analysis, keyword matching, and part-of-speech (POS) tagging to evaluate and improve the relevance of its responses.

Initially, a rule-based system was implemented to find the best-matching answers based on keyword overlap and sentiment alignment. This provided an interpretable and fast solution but lacked contextual understanding. To enhance performance, machine learning models like Logistic Regression and Random Forest were introduced, followed by the implementation of an LSTM-based deep learning model to capture sequence dependencies and context in text data.

The chatbot analyzer was evaluated using standard metrics such as accuracy. Despite challenges like limited labeled data and varied sentence structures, the LSTM model improved the evaluation accuracy to an extent but still indicated room for improvement.

In conclusion, this chatbot system provides a functional base for virtual assistants in customer service. While the rule-based model ensures interpretability, the neural approach opens paths for deeper contextual understanding. Combining these methods creates a more robust and scalable conversational interface.