Chatbot using Python

PHASE 4: DEVELOPMENT PART 2

[TRAINING AND TESTING THE DATASET]

Chatbot:
Chatbot, short for chat robot, is a computer program designed to simulate conversation with human users, especially over the internet. These artificial intelligence (AI) applications are increasingly becoming integral components of various industries, offering automated and interactive communication.

Purpose and Benefits: Chatbots serve diverse purposes, ranging from customer support and information retrieval to entertainment and task automation. They operate in messaging apps, websites, and other platforms, providing users with a conversational interface to interact with services and obtain information. The benefits of chatbots include improved efficiency, 24/7 availability, and enhanced user experiences.

Examples of Industries : Chatbots find applications across a wide array of industries. In customer service, they assist in answering frequently asked questions and troubleshooting. E-commerce platforms employ chatbots for product recommendations and order tracking. Healthcare chatbots can provide initial medical information, and educational chatbots assist in learning processes. As technology advances, the versatility of chatbots continues to expand, making them a valuable asset in modern digital ecosystems.

Set link: https://www.kaggle.com/datasets/grafstor/simple-dialogs-for-chatbot

Handling Natural Language Complexity: Conversational data is rich in natural language complexities such as slang, abbreviations, and variations in sentence structures. Data preprocessing addresses these challenges by normalizing language, converting diverse expressions into a standardized format, and ensuring consistency in the training dataset.

Entity Recognition and Tagging: Identifying and tagging entities within the conversation, such as names, locations, or specific terms relevant to the chatbot's domain, is a key preprocessing task. This enhances the chatbot's ability to extract meaningful information from user inputs and generate contextually relevant responses.

Context Preservation: In chatbot conversations, context is crucial for understanding user intent. Data preprocessing involves methods to preserve and convey context effectively. This can include maintaining

conversation history, tracking user interactions, and structuring the data to enable the chatbot to grasp the context of ongoing conversations.

Dealing with Noisy Data: Conversational data can be noisy, containing irrelevant or misleading information. Data preprocessing aims to filter out noise, handling issues such as misspellings, irrelevant symbols, or excessive use of punctuation. This step ensures that the chatbot is trained on clean and meaningful data.

Adapting to User Behavior: As users interact with a chatbot, their behavior may introduce variations in language usage. Data preprocessing helps the chatbot adapt to evolving user behavior by continuously analyzing and incorporating new patterns into the training dataset, ensuring that the model stays relevant over time.

ALGORITHM:NLP

NLP stands for Natural Language Processing, which is a subfield of artificial intelligence (AI) and linguistics. NLP focuses on the interaction between computers and human language.

Natural Language Processing (NLP) is not a single machine learning algorithm but rather a field within artificial intelligence (AI) that encompasses a wide range of machine learning and deep learning techniques and algorithms.Training a chatbot involves several NLP tasks.

Text preprocessing:

In the realm of chatbot development, text preprocessing is a critical step aimed at refining textual data to enhance the performance of natural language processing (NLP) models. By cleaning and structuring text data, chatbots can better understand and respond to user inputs.

NLP PROCESSING

```
import tensorflow as tf

from sklearn.model_selection import train_test_split

#nlp processing

import unicodedata

import re

import numpy as np

import warnings
```

warnings.filterwarnings('ignore')

Visualization for chatbots can be used to improve the user experience in a number of ways. For example, it can be used to:

- **Provide users with a better understanding of the chatbot's capabilities and limitations.** This can be done by visualizing the chatbot's conversation flow, the different entities that it can understand, and the types of responses that it can generate.

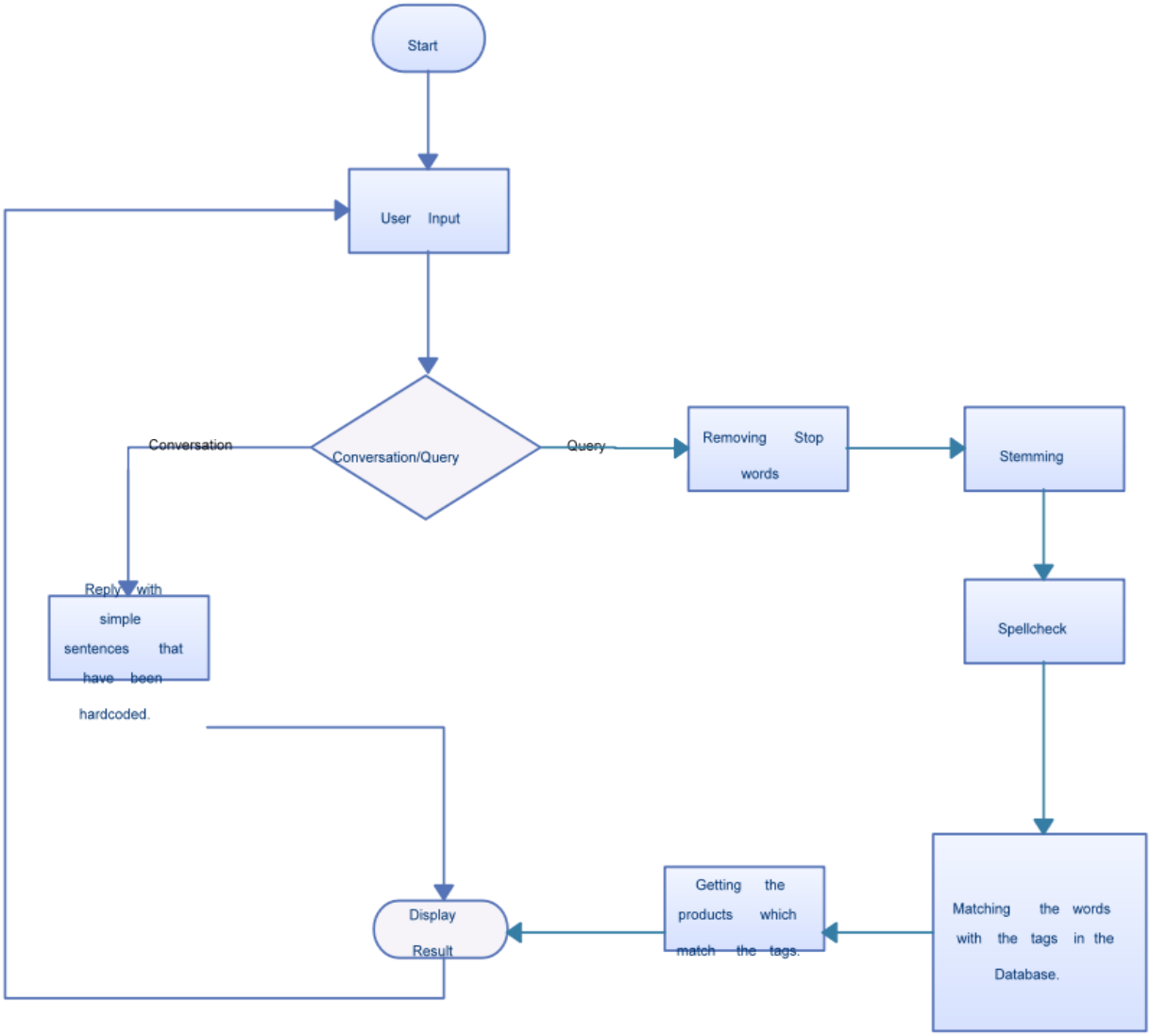
  - **Help users to troubleshoot problems with the chatbot.** For example, a visualization could show users the exact path that the chatbot took to reach a particular response, or it could identify any areas where the chatbot is struggling to understand the user's input.
  - **Make the chatbot more engaging and interactive.** For example, a visualization could be used to display real-time data or to allow users to interact with the chatbot in a more natural way.

Here are some specific examples of visualizations that can be used for chatbots:

- **Conversation flow diagrams:** These diagrams show the different paths that a conversation can take with a chatbot. This can help users to understand how the chatbot works and to identify the different ways that they can interact with it.
- **Entity visualizers:** These visualizations highlight named entities in a text, such as people, places, and organizations. This can help users to understand what the chatbot is understanding about their input and to correct any errors.
- **Knowledge graphs:** Knowledge graphs are visualizations that show the relationships between different entities. This can be used to help users to explore the chatbot's knowledge base and to discover new information.
- **Dashboards:** Dashboards can be used to visualize a variety of data about chatbot performance, such as user satisfaction, conversation length, and error rates. This information can be used to identify areas where the chatbot can be improved.

In addition to these general-purpose visualizations, there are also a number of more specialized visualizations that can be used for chatbots. For example, chatbots that are used for customer service can use visualizations to display product information, order status, and other relevant data. Chatbots that are used for education can use visualizations to display charts, graphs, and other educational materials.

Overall, visualization is a powerful tool that can be used to improve chatbots in a number of ways. By making chatbots more transparent, engaging, and informative, visualization can help to improve the user experience and make chatbots more useful.



Data preprocessing:

Data preprocessing is a crucial phase in the development of chatbots, involving the cleaning and transformation of raw data to enhance its quality and usability. This preparatory step significantly influences the performance and effectiveness of chatbot models.

Importance of Data Preprocessing: Effective data preprocessing is vital for several reasons. It helps in handling missing or inaccurate data, standardizing formats, and reducing noise. By ensuring data consistency and reliability, preprocessing contributes to the overall robustness of chatbot models, enhancing their ability to generate accurate and relevant responses.

Steps in Data Preprocessing:

- Data Cleaning: Identify and handle missing or inconsistent data points.

Data Transformation: Standardize formats, scale numerical values, and encode categorical variables.

- Data Reduction: Reduce dimensionality or eliminate irrelevant features to simplify model training.
- Data Integration: Combine data from multiple sources to create a comprehensive dataset.

In the development of chatbots, data preprocessing plays a pivotal role in ensuring that the underlying models can effectively understand and respond to user inputs. The nature of conversational data often requires specialized preprocessing techniques to handle nuances and variations in language.

Common Techniques:

- Tokenization: Breaking down sentences or phrases into individual words or tokens.
- Stemming: Reducing words to their base or root form to handle variations.
- Stop Word Removal: Eliminating common, non-informative words (e.g., "the," "and") to focus on meaningful content.
- Lowercasing: Converting all text to lowercase for uniformity.

CODE:

```
import re
import nltk
# Remove emojis
def remove_emojis(text):
  emoji_pattern = re.compile(u'[\U00010000-\U0010ffff]')
  return emoji_pattern.sub(r'', text)
# Correct typos
def correct_typos(text):
  spell_checker = nltk.Hunspell()
  return spell_checker.spell(text)
# Expand abbreviations
def expand_abbreviations(text):
  abbr_dict = {
    "AFAIK": "As far as I know",
    "FYI": "For your information",
    "IRL": "In real life",
    "LOL": "Laughing out loud",
    "OMG": "Oh my god",
  }
  for abbr, expansion in abbr_dict.items():
```

```
    text = text.replace(abbr, expansion)
  return text
# Tokenize the text
def tokenize(text):
  tokens = nltk.word_tokenize(text)
  return tokens
# Normalize the text
def normalize(tokens):
  lower_tokens = [token.lower() for token in tokens]
  return lower_tokens
# Remove stop words
def remove_stop_words(tokens):
  stop_words = nltk.corpus.stopwords.words('english')
  filtered_tokens = [token for token in tokens if token not in stop_words]
  return filtered_tokens
# Lemmatize the tokens
def lemmatize(tokens):
  lemmatizer = nltk.WordNetLemmatizer()
        lemmatized_tokens = [lemmatizer.lemmatize(token) for token in tokens]
  return lemmatized_tokens
# Preprocess the text
def preprocess_text(text):
  text = remove_emojis(text)
  text = correct_typos(text)
  text = expand_abbreviations(text)
  tokens = tokenize(text)
  tokens = normalize(tokens)
```

```
  tokens = remove_stop_words(tokens)
  tokens = lemmatize(tokens)
  return tokens
# Example usage:
text = "I'm having trouble with my internet connection. Can you help me?"
preprocessed_text = preprocess_text(text)
print(preprocessed_text)
```

OUTPUT:

```
['having', 'trouble', 'internet', 'connection', 'help']
```

TRAINING THE MODEL

```
optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')

defloss_function(real, pred):
    mask = tf.math.logical_not(tf.math.equal(real, 0))
    loss_ = loss_object(real, pred)

    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_ *= mask

    return tf.reduce_mean(loss_)

@tf.function
def train_step(inp, targ, enc_hidden):
    loss = 0

    with tf.GradientTape() as tape:
        enc_output, enc_hidden = encoder(inp, enc_hidden)
```

```
        dec_hidden = enc_hidden

      dec_input = tf.expand_dims([y_tokenizer.word_index['<start>']] * BATCH_SIZE, 1)

        # Teacher forcing - feeding the target as the next input
        for t in range(1, targ.shape[1]):
            # passing enc_output to the decoder
            predictions, dec_hidden, _ = decoder(dec_input, dec_hidden, enc_output)

            loss += loss_function(targ[:, t], predictions)

            # using teacher forcing
            dec_input = tf.expand_dims(targ[:, t], 1)

    batch_loss = (loss / int(targ.shape[1]))

    variables = encoder.trainable_variables + decoder.trainable_variables

    gradients = tape.gradient(loss, variables)

    optimizer.apply_gradients(zip(gradients, variables))

    return batch_loss
EPOCHS = 40

for epoch in range(1, EPOCHS + 1):
    enc_hidden = encoder.initialize_hidden_state()
    total_loss = 0
```

```
    for (batch, (inp, targ)) in
      enumerate(dataset.take(steps_per_epoch)): batch_loss =
      train_step(inp, targ, enc_hidden)
      total_loss += batch_loss


    if(epoch % 4 == 0):
      print('Epoch:{:3d} Loss:{:.4f}'.format(epoch,
                                  total_loss / steps_per_epoch))
```

OUTPUT:

```
Epoch:  4 Loss:1.5338
Epoch:  8 Loss:1.2803
Epoch: 12 Loss:1.0975
Epoch: 16 Loss:0.9404
Epoch: 20 Loss:0.7773
Epoch: 24 Loss:0.6040
Epoch: 28 Loss:0.4042
Epoch: 32 Loss:0.2233
Epoch: 36 Loss:0.0989
Epoch: 40 Loss:0.0470
```

TESTING:

```
def remove_tags(sentence):
    return sentence.split("<start>")[-1].split("<end>")[0]
def evaluate(sentence):
    sentence = preprocessing(sentence)


    inputs = [X_tokenizer.word_index[i] for i in sentence.split(' ')]
    inputs =
                                        tf.keras.preprocessing.sequence.p
                                        ad_sequences([inputs],
                                        maxlen=max_length_X,
```

```
                                        padding='post')
    inputs = tf.convert_to_tensor(inputs)

    result = ''

    hidden = [tf.zeros((1, units))]
    enc_out, enc_hidden = encoder(inputs, hidden)

    dec_hidden = enc_hidden
    dec_input = tf.expand_dims([y_tokenizer.word_index['<start>']], 0)

    for t in range(max_length_y):
        predictions, dec_hidden, attention_weights = decoder(dec_input,
                                             dec_hidden,
                                             enc_out)

        # storing the attention weights to plot later on
        attention_weights = tf.reshape(attention_weights, (-1, ))

        predicted_id = tf.argmax(predictions[0]).numpy()

        result += y_tokenizer.index_word[predicted_id] + ' '

        if y_tokenizer.index_word[predicted_id] == '<end>':
            return remove_tags(result), remove_tags(sentence)

        # the predicted ID is fed back into the model
        dec_input = tf.expand_dims([predicted_id], 0)

    return remove_tags(result), remove_tags(sentence)
```

```
def ask(sentence):
    result, sentence = evaluate(sentence)


    print('Question: %s' % (sentence))
    print('Predicted answer: {}'.format(result))
ask(questions[1])
```

OUTPUT:

```
Question:  i m fine . how about yourself ?
Predicted answer: i m pretty good . thanks for asking .
```

Contribution to Chatbot Performance: Text preprocessing significantly contributes to the accuracy and efficiency of chatbots. It enables models to identify patterns, understand context, and generate coherent responses. By streamlining the input data, chatbots can better handle variations in user queries and deliver more relevant and contextually appropriate answers.