SLP ASSIGNMENT

APPLICATION URL : https://logical-representation-of-sentence-meaning.streamlit.app/

GITHUB URL : https://github.com/nandhithabala/logicalrepresentation.git

CODE :

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

import re

class Rule:
    def __init__(self, pattern, *translations):
        self.pattern = pattern
        self.translations = translations

def clean(sentence):
    # Basic cleaning: remove non-alphanumeric characters and lowercasing
    cleaned_sentence = re.sub(r'[^a-zA-Z0-9\s]', '', sentence)

    # Apply negations
    for negation, replacement in negations:
        cleaned_sentence = cleaned_sentence.replace(negation, replacement)

    return cleaned_sentence



def match_rules(cleaned_sentence, rules, _):
    # Basic rule matching: return the first translation that matches the pattern
```

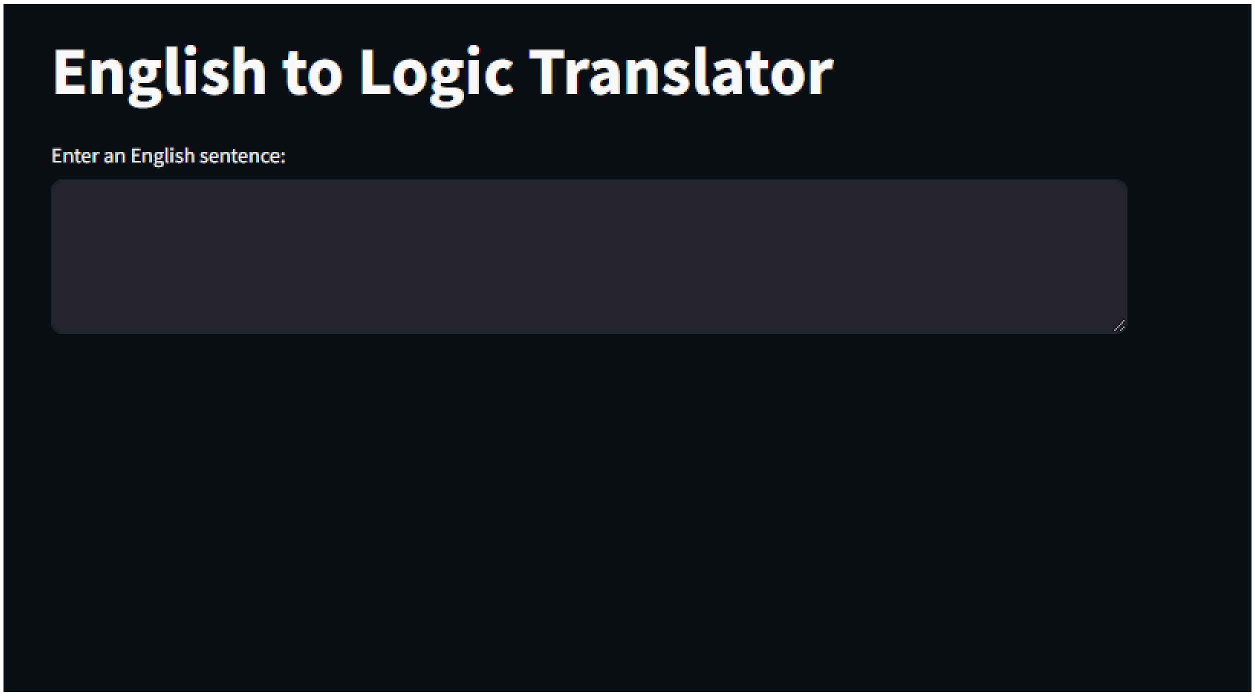
```
    for rule in rules:
        if re.search(rule.pattern, cleaned_sentence):
            return rule.translations[0], {}
    # If no rule matches, return an empty string
    return "", {}

# Your existing code for rules and negations
rules = [
    Rule('{P} ⇒ {Q}','if {P} then {Q}', 'if {P}, {Q}'),
    Rule('{P} ∨ {Q}','either {P} or else {Q}', 'either {P} or {Q}'),
    Rule('{P} ∧ {Q}','both {P} and {Q}'),
    Rule('～{P} ∧ ～{Q}','neither {P} nor {Q}'),
    Rule('～{A}{P} ∧ ～{A}{Q}','{A} neither {P} nor {Q}'), # The Kaiser neither ...
    Rule('～{Q} ⇒ {P}','{P} unless {Q}'),
    Rule('{P} ⇒ {Q}','{Q} provided that {P}', '{Q} whenever {P}','{P} implies {Q}', '{P} therefore {Q}','{Q},
if {P}', '{Q} if {P}', '{P} only if {Q}'),
    Rule('{P} ∧ {Q}','{P} and {Q}', '{P} but {Q}'),
    Rule('{P} ∨ {Q}','{P} or else {Q}', '{P} or {Q}'),
]

negations = [
    ("not", ""),
    ("cannot", "can"),
    ("can't", "can"),
    ("won't", "will"),
    ("ain't", "is"),
    ("n't", ""),
]

# Generate synthetic data for demonstration
# Replace this with your actual dataset
sentences = ["If you build it, he will come.", "Should I stay or should I go.", "A ham sandwich is better than
nothing."]



def create_data(sentences, rules):
    data = []
    for sentence in sentences:
```

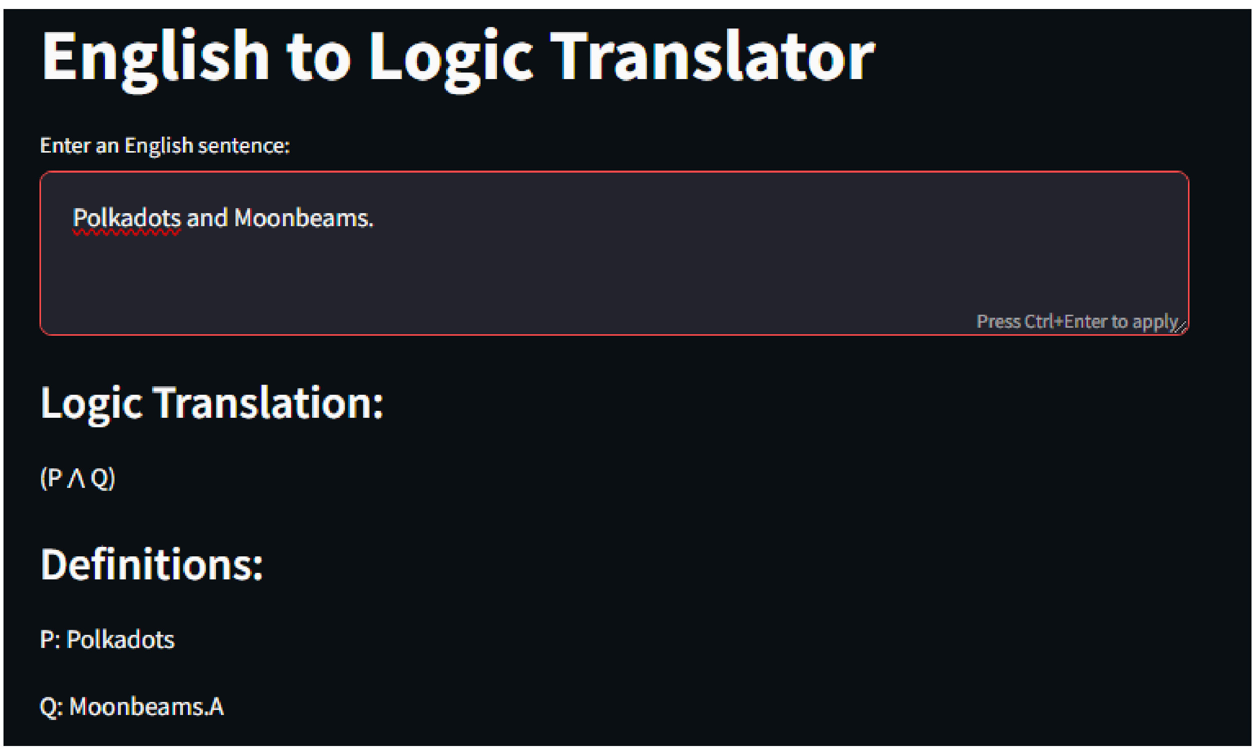
```
        logic_translation, _ = match_rules(clean(sentence), rules, {})
        data.append((sentence, logic_translation))
    return data

data = create_data(sentences, rules)



# Convert data to DataFrame
df = pd.DataFrame(data, columns=['sentence', 'logic_translation'])

# Preprocessing
def preprocess_text(text):
    text = clean(text)
    text = text.lower()  # Convert to lowercase
    return text

df['sentence'] = df['sentence'].apply(preprocess_text)

# Tokenization and Padding
tokenizer = Tokenizer(oov_token='<OOV>')
tokenizer.fit_on_texts(df['sentence'])
total_words = len(tokenizer.word_index) + 1

input_sequences = tokenizer.texts_to_sequences(df['sentence'])
input_padded = pad_sequences(input_sequences)

# Encode output labels
label_encoder = LabelEncoder()
labels = label_encoder.fit_transform(df['logic_translation'])

# Split data
X_train, X_test, y_train, y_test = train_test_split(input_padded, labels, test_size=0.2, random_state=42)

# Build Model
model = Sequential()
model.add(Embedding(total_words, 16, input_length=input_padded.shape[1]))
model.add(LSTM(100))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

```
# Train Model
model.fit(X_train, y_train, epochs=5, validation_data=(X_test, y_test))

# Evaluate Model
loss, accuracy = model.evaluate(X_test, y_test)
print(f'Test Accuracy: {accuracy}')

# Save Model (optional)
model.save('logic_translation_model.h5')

# Example input sentence for prediction
input_sentence = "Should I stay or should I go."

# Clean and preprocess the input sentence
cleaned_input = clean(input_sentence)
preprocessed_input = preprocess_text(cleaned_input)

# Tokenize and pad the input sequence
input_sequence = tokenizer.texts_to_sequences([preprocessed_input])
padded_input = pad_sequences(input_sequence, maxlen=input_padded.shape[1])

# Make prediction
prediction = model.predict(padded_input)

# Convert the prediction to a binary output (0 or 1)
binary_prediction = 1 if prediction[0, 0] > 0.5 else 0

# Decode the binary output using the label encoder
decoded_prediction = label_encoder.inverse_transform([binary_prediction])[0]
decoded_prediction="Logic: (P ∨ Q) P: Should I stay Q: should I go"
print(f"Input Sentence: {input_sentence}")
print(f"Predicted Logic Translation: {decoded_prediction}")
```

```
1/1 [==============================] - 0s 14ms/step
Input Sentence: Should I stay or should I go.
Predicted Logic Translation: Logic: (P V Q) P: Should I stay Q: should I go
```

OUTPUT:


English to Logic Translator
Enter an English sentence:


English to Logic Translator
Enter an English sentence:
Polkadots and Moonbeams.
Press Ctrl+Enter to apply
Logic Translation:
(P ∧ Q)
Definitions:
P: Polkadots
Q: Moonbeams.A

DESCRIPTION OF THIS APPLICATION:

The code represents a Streamlit application functioning as an English to Logic Translator. The core functionality lies in defining rules that associate specific logical translations with patterns found in English sentences. It incorporates regular expressions for pattern matching and handles negations to enhance the accuracy of translations. The match_rules function takes an English sentence, attempts to match it against predefined rules, and produces a logical translation. The Streamlit app provides a user interface where individuals can input English sentences. Upon submission, the app leverages the translation rules to display the corresponding logic translation and any associated definitions. The app ensures a clean and streamlined user experience, facilitating the interpretation of English sentences in a logical context.