Amazon Toys & Games Q & A Chatbots

By John Callahan

Dataset URL: https://cseweb.ucsd.edu/~jmcauley/datasets/amazon/qa/

Dataset: YAML (NLP), JSON (Amazon)

Category: Toys & Games

Project Description

This project focuses on building two chatbots, Term frequency-Inverse document frequency (TF-IDF) and seq2seq. It involves leveraging a dataset of questions and answers from the "Toys & Games" category of Amazon reviews. It encompasses data preprocessing, exploratory data analysis (EDA), and natural language processing (NLP) techniques to create a chatbot system capable of meaningful interaction. It also incorporates NLP libraries, such as spaCy and Transformers, and employs machine learning models to ensure a responsive and adaptive interaction process.

In [ ]:
```
#Import Libraries
import numpy as np
import pandas as pd
import re
import ast
import io
import os
import zipfile
import yaml

import spacy
from spacy import displacy

import requests
from gensim.models import Word2Vec
from keras import Input, Model
from keras.activations import softmax
from keras.layers import Embedding, LSTM, Dense
from tensorflow.keras.optimizers import RMSprop
from keras_preprocessing.sequence import pad_sequences
from tensorflow.keras.utils import to_categorical
from keras_preprocessing.text import Tokenizer

from textblob import TextBlob
from operator import itemgetter

#from transformers import pipeline

from sklearn.metrics.pairwise import cosine_similarity
```

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import TfidfVectorizer

import nltk
from nltk import word_tokenize
from nltk import sent_tokenize
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from nltk.stem.snowball import SnowballStemmer
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.probability import FreqDist

from wordcloud import WordCloud

import matplotlib.pyplot as plt
%matplotlib inline

#Download NLTK data
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('universal_tagset')
nltk.download('omw-1.4')
```

In [ ]:

```
#Create a function to clean text
def clean_text(text_to_clean):
    text_to_clean = str(text_to_clean)
    res = text_to_clean.lower()
    res = re.sub(r"i'm", "i am", res)
    res = re.sub(r"he's", "he is", res)
    res = re.sub(r"she's", "she is", res)
    res = re.sub(r"it's", "it is", res)
    res = re.sub(r"that's", "that is", res)
    res = re.sub(r"what's", "what is", res)
    res = re.sub(r"where's", "where is", res)
    res = re.sub(r"how's", "how is", res)
    res = re.sub(r"\'ll", "will", res)
    res = re.sub(r"\'ve", "have", res)
    res = re.sub(r"\'re", "are", res)
    res = re.sub(r"\'d", "would", res)
    res = re.sub(r"won't", "will not", res)
    res = re.sub(r"can't", "cannot", res)
    res = re.sub(r"n't", " not", res)
    res = re.sub(r"n'", "ng", res)
    res = re.sub(r"'bout", "about", res)
    res = re.sub(r"'til", "until", res)
    res = re.sub(r"[-()\"#/@;:<>{}`+=~|.!?,]", "", res)
    return res
```

Load YAML dataset for NLP

In [ ]:

```
dir_path = 'nlp/data'

files_list = os.listdir(dir_path + os.sep)

nlp_questions = list()
```

```
nlp_answers = list()
for filepath in files_list:
    stream = open(dir_path + os.sep + filepath, 'rb')
    docs = yaml.safe_load(stream)
    conversations = docs['conversations']
    for con in conversations:
        if len(con) > 2:
            nlp_questions.append(con[0])
            ans = ''
            for rep in con[1:]:
                ans += ' ' + rep
            nlp_answers.append(ans)
        elif len(con) > 1:
            nlp_questions.append(con[0])
            nlp_answers.append(con[1])
```

Load JSON dataset for Amazon Questions and Answers

In [ ]:
```
#Create Empty Lists for Questions (Q) and Answers (A)
questions = []
answers = []

#Load Amazon Q and A into the newly created lists
with open ('dataset/qa_Toys_and_Games.json', 'r') as f:
    for line in f:
        data = ast.literal_eval (line)
        questions.append(data['question'].lower())
        answers.append(data['answer'].lower())
```

Data Cleaning

In [ ]:
```
#Clean NLP questions
index = 0
while index < len(nlp_questions):
    nlp_questions[index] = clean_text(nlp_questions[index])
    index += 1
```

In [ ]:
```
#Print First 20 and Length of NLP Questions
print(nlp_questions[0:20])
print(len(nlp_questions))
```

In [ ]:
```
#Clean NLP answers with clean_text function
index = 0
while index < len(nlp_answers):
    nlp_answers[index] = clean_text(nlp_answers[index])
    index += 1
```

In [ ]:
```
#Print First 20 and Length of NLP Answers
print(nlp_answers[0:20])
print(len(nlp_answers))
```

In [ ]:
```
#Clean Amazon data with clean_text function
#Clean Amazon questions
index = 0
while index < len(questions):
```

```
    questions[index] = clean_text(questions[index])
    index += 1
```

In [ ]:
```
#Print First 20 and Length of Amazon questions
print(questions[0:20])
print(len(questions))
```

In [ ]:
```
#Clean Amazon answers
index = 0
while index < len(answers):
    answers[index] = clean_text(answers[index])
    index += 1
```

In [ ]:
```
#Print First 20 and Length of Amazon answers
print(answers[0:20])
print(len(answers))
```

Combine Datasets

In [ ]:
```
#Combine NLP and Amazon Datasets
combined_questions = questions + nlp_questions
combined_answers = answers + nlp_answers
```

In [ ]:
```
#Print ength of Combined Questions and Answers
print(len(combined_questions))
print(len(combined_answers))
```

EDA

Visualize Dataset

In [ ]:
```
#Create a DataFrame that contains the Q and A
df = pd.DataFrame(combined_questions, combined_answers)
```

In [ ]:
```
df.head(10)
```

In [ ]:
```
df.tail(10)
```

In [ ]:
```
df.describe()
```

In [ ]:
```
df.info()
```

As we can see above the count in the describe() and the Non-Null in info() are matching so we do not need to deal with any missing values.

NLP

TextBlob

Visualize Word Frequency with Pandas

In [ ]:
```
#Make a Text Blob for both Q and A
tb_q = TextBlob(" ".join(combined_questions))
tb_a = TextBlob(" ".join(combined_answers))
```

In [ ]:
```
#Remove Stop Words
stop_words = stopwords.words('english')
```

In [ ]:
```
#Getting the Word Frequencies
items_q = tb_q.word_counts.items()
items_a = tb_a.word_counts.items()
```

In [ ]:
```
#Removing the Stop Words
items_q = [item for item in items_q if item[0] not in stop_words]
items_a = [item for item in items_a if item[0] not in stop_words]
```

In [ ]:
```
#Sort words by frequency
sort_items_q = sorted(items_q, key=itemgetter(1), reverse=True)
sort_items_a = sorted(items_a, key=itemgetter(1), reverse=True)
```

In [ ]:
```
#Getting the Top 20 Words
top20_q = sort_items_q[1:21]
top20_a = sort_items_a[1:21]
```

In [ ]:
```
#Convert Top 20 Q to a Dataframe
df_q = pd.DataFrame(top20_q, columns=['word', 'count'])

#Convert Top 20 Q to a Dataframe
df_a = pd.DataFrame(top20_a, columns=['word', 'count'])
```

In [ ]:
```
#Display Top 20 words from Questions Dataframe
df_q
```

In [ ]:
```
#Display Top 20 words from Answers Dataframe
df_a
```

Plot the Top 20 Words for Questions and Answers

In [ ]:
```
#Visualize the Top 20 Q DataFrame
axes = df_q.plot.bar(x='word', y='count', legend=False)
plt.tight_layout()
```

In [ ]:
```
#Visualize the Top 20 A DataFrame
axes = df_a.plot.bar(x='word', y='count', legend=False)
plt.tight_layout()
```

**As we can see from both the dataframes and as visually represented in the plots there are similarities between the key words in each. "One" and "would" are both the top words with just the order reversed. We also see words like "use", "2", "get", and "set" in each dataset again with just the ordering changed.

Count Vectorization

Create Vectorizer for Questions

Goal: To find out the Top 50 Most Common Words in Combined Questions

In [ ]:
```
# Create Model and Remove Stopwords
cv_questions = CountVectorizer(stop_words='english')
```

In [ ]:
```
#Fit model and train vocabulary from the questions
cv_questions.fit(combined_questions)
```

In [ ]:
```
#Print Top 50 of the Vocabulary
print('Vocabulary: ')
print(list(cv_questions.vocabulary_.items())[0:50])
```

In [ ]:
```
#Create copy of vocabulary dictionary
vocabulary_copy = cv_questions.vocabulary_.copy()
```

In [ ]:
```
#Sort the newly created vocabulary dictionary copy
sorted_vocabulary_copy = sorted(vocabulary_copy.items(), key=lambda x: x[1], reverse=T
```

In [ ]:
```
#Display First 50 of Sorted Vocabulary Dictionary
sorted_vocabulary_copy[0:50]
```

**Since we sorted the dictionary with Reverse=True the "z" words are all displayed first. We can see there are over 21,000 references to words such "zyx22", "zx", and "zumbuddies".

Convert First 100 Questions into Arrays

In [ ]:
```
#Convert First 100 Questions into Arrays and Remove Stop Words
cv_questions_array = CountVectorizer(stop_words='english')
```

In [ ]:
```
# Create Model and Remove Stopwords
cv_fit_questions = cv_questions_array.fit_transform(combined_questions[0:99])
```

In [ ]:
```
#Display array
print(cv_fit_questions.toarray())
```

Display Question 13 and the Array

In [ ]:
```
#Display Question 13
questions[12]
```

In [ ]:
```
#Question 13 in Vectorizer array
print(cv_fit_questions.toarray()[12])
```

Complete POS tagging for Question 13

In [ ]:
```
#Perform POS Tagging on Question 13
pos_text = word_tokenize(questions[12])
```

```
nltk.pos_tag(pos_text)
```

In [ ]:
```
import nltk
nltk.download('tagsets')

#Tag Definitions from NLTK Help
nltk.help.upenn_tagset()
```

Combine Questions And Answers from both Datasets

In [ ]:
```
##Combine all Questions Together
all_questions = "\n".join(combined_questions)
```

In [ ]:
```
print(all_questions[0:100])
```

In [ ]:
```
#Combine all Answers Together
all_answers = "\n".join(combined_answers)
```

In [ ]:
```
print(all_answers[0:100])
```

Tokenize Questions and Answers using Word Tokenizer

In [ ]:
```
#Tokenize Questions using Word Tokenizer
questions_wt = word_tokenize(all_questions)
```

In [ ]:
```
#Tokenize Answers using Word Tokenizer
answers_wt = word_tokenize(all_answers)
```

In [ ]:
```
#How Many Words in Questions
print("There are " + str(len(questions_wt)) + " words in " + str(len(combined_questior
```

In [ ]:
```
#How Many Words in Answers
print("There are " + str(len(answers_wt)) + " words in " + str(len(combined_answers))
```

**After combining Questions and Answers from both datasets we have 52,052 questions. THere are 638,044 and 1,337,281 words respectively.

Remove Numeric, Spaces and Symbols from Q & A

In [ ]:
```
#Remove Punctuation Marks and Numbers from Questions
questions_wt_no_pun = []
for w in questions_wt:
    if w.isalpha():
        questions_wt_no_pun.append(w.lower())
```

In [ ]:
```
#Remove Punctuation Marks and Numbers from Answers
answers_wt_no_pun = []
for w in answers_wt:
    if w.isalpha():
        answers_wt_no_pun.append(w.lower())
```

In [ ]:
```
#How Many Words in Questions without Punctuation and Numbers
print("There are " + str(len(questions_wt_no_pun)) + " words in " + str(len(combined_c
```

In [ ]:
```
#How Many Words in Answers without Punctuation and Numbers
print("There are " + str(len(answers_wt_no_pun)) + " words in " + str(len(combined_ans
```

**After removing numbers, spaces and symbols we have reduced Questions from 638,044 to 620,513. We have also reduced Answers from 1,337,281 to 1,291,706.

Remove Stop words

In [ ]:
```
#Removing Stop Words
stop_words = stopwords.words('english')
```

In [ ]:
```
clean_words_questions = []
for w in questions_wt_no_pun:
    if w not in stop_words:
        clean_words_questions.append(w)
```

In [ ]:
```
clean_words_answers = []
for w in answers_wt_no_pun:
    if w not in stop_words:
        clean_words_answers.append(w)
```

In [ ]:
```
#How Many Words in Questions without Punctuation and Stopwords
print("There are " + str(len(clean_words_questions)) + " words in " + str(len(combined
```

In [ ]:
```
#How Many Words in Answers without Punctuation
print("There are " + str(len(clean_words_answers)) + " words in " + str(len(combined_a
```

**After removing the 'english' stop words questions was even further reduced down to 289,111 words from the previous 620,513. Answers was reduced from 1,291,706 to 617347. This cleaning is key to reduce noise, improve computational efficiency, enhance signal-to-noise ratio (TF-IDF relies on the importance of less frequent terms), and to optimize storage and memory.

Create Frequency Distributions for Q & A

In [ ]:
```
#Create Frequency Distribution of Questions
fdist_questions = FreqDist(clean_words_questions)
```

In [ ]:
```
#Create Frequency Distribution of Answers
fdist_answers = FreqDist(clean_words_answers)
```

Visualize Frequency Distribution of Top 20 Q & A

In [ ]:
```
#20 Most Common Question Words
fdist_questions.most_common(20)
```

In [ ]:
```
#20 Most Common Answer Words
fdist_answers.most_common(20)
```

In [ ]:
```
#Graph Frequency Distribution of 20 Most Common Question Words
fdist_questions.plot(20)
```

In [ ]:
```
#Graph Frequency Distribution of 20 Most Common Answer Words
fdist_answers.plot(20)
```

Word Cloud for Combined Q & A

In [ ]:
```
#Create Word Cloud for Questions
wc_questions = WordCloud().generate(all_questions)
```

In [ ]:
```
plt.figure(figsize = (10, 10))
plt.imshow(wc_questions)
```

In [ ]:
```
#Create Word Cloud for Answers
wc_answers = WordCloud().generate(all_answers)
```

In [ ]:
```
plt.figure(figsize = (10, 10))
plt.imshow(wc_answers)
```

**In the frequency distribution we again see similar words in both the questions and answers with just their order changed. Words such as would, like, and old are in both. In questions the top 3 are come, one, and would. In answers the top 3 are yes, would, and one.

Named ENtity Recognition (NER)

spaCy

In [ ]:
```
#Convert the list into a string
q_str = ''
a_str = ''

for q in combined_questions:
    q_str += str(q)+' '
for a in combined_answers:
    a_str += str(a)+' '
```

In [ ]:
```
#Download and load the en_core_web_lg model
!python -m spacy download en_core_web_lg
nlp = spacy.load("en_core_web_lg")
#Questions
doc_q = nlp(q_str[0:300000])

displacy.render(doc_q, style='ent', jupyter=True)
```

In [ ]:
```
nlp = spacy.load("en_core_web_lg")
#Answers
doc_a = nlp(a_str[0:300000])

displacy.render(doc_a, style='ent', jupyter=True)
```

**While spaCy is an effective NER it does have a limit of 1,000,000 characters so ensure you either have a smaller dataset or limit the characters as we have done above. Also the models require roughly 1 GB of memory per 100,000 characters. Due to this you can encounter memory allocation errors.

NLP Transformer Pipeline NER

In [ ]:
```
#nlp_transformer = pipeline(task='ner')
```

In [ ]:
```
#for item in nlp_transformer(q_str[0:300000]):
    print(f"{item['word'], item['entity']}")
```

In [ ]:
```
#for item in nlp_transformer(a_str[0:300000]):
    print(f"{item['word'], item['entity']}")
```

**Though I was unable to load NLP Transformer in the Remote VM I was able to load this properly on my local machine. It was not able to properly apply NER to any words in the dataset. Based on this spaCy is the best choice between the two though it still had challenges.

Term Frequency-Inverse Document Frequency based Vectorizer

In [ ]:
```
corpus_questions = pd.Series(combined_questions)

# Convert pandas Series to list
if isinstance(corpus_questions, pd.Series):
    corpus_questions = corpus_questions.tolist()

# Filter and clean the list to ensure all elements are strings
corpus_questions = [str(doc) if isinstance(doc, (int, float)) else doc for doc in corp
corpus_questions = [doc for doc in corpus_questions if isinstance(doc, str)]
```

In [ ]:
```
corpus_answers = pd.Series(combined_answers)

# Convert pandas Series to list
if isinstance(corpus_answers, pd.Series):
    corpus_answers = corpus_answers.tolist()

# Filter and clean the list to ensure all elements are strings
corpus_answers = [str(doc) if isinstance(doc, (int, float)) else doc for doc in corpus
corpus_answers = [doc for doc in corpus_answers if isinstance(doc, str)]
```

Data preprocessing functions

In [ ]:
```
def text_clean(corpus, keep_list):
    cleaned_corpus = []
    for doc in corpus:
        if isinstance(doc, str):
            cc = []
            for word in doc.split():
                # Check if word is in keep_list, avoid replacing valid characters
                if word not in keep_list:
```

```
                    p1 = re.sub(pattern='[^a-zA-Z0-9]', repl=' ', string=word)  # Repl
                    cc.append(p1)
                else:
                    cc.append(word)
            cleaned_corpus.append(' '.join(cc))
        else:
            raise ValueError(f"Expected string, got {type(doc)}")
    return cleaned_corpus
```

In [ ]:
```
lemmatizer = WordNetLemmatizer()

def lemmatize(corpus):
    lemmatized_corpus = []
    for doc in corpus:
        # Split document into words, lemmatize each word, and rejoin
        lemmatized_doc = [lemmatizer.lemmatize(word) for word in doc.split()]
        lemmatized_corpus.append(' '.join(lemmatized_doc))
    return lemmatized_corpus
```

In [ ]:
```
def stem(corpus, stem_type=None):
    if stem_type == 'snowball':
        stemmer = SnowballStemmer(language = 'english')
        corpus = [[stemmer.stem(x) for x in x] for x in corpus]
    else:
        stemmer = PorterStemmer()
        corpus = [[stemmer.stem(x) for x in x] for x in corpus]
    return corpus
```

In [ ]:
```
def stopwords_removal(corpus):
    stop = set(stopwords.words('english'))  # Define your stop words
    filtered_corpus = []
    for doc in corpus:
        if isinstance(doc, str):  # Ensure the element is a string
            filtered_doc = [word for word in doc.split() if word not in stop]
            filtered_corpus.append(" ".join(filtered_doc))
        else:
            raise ValueError(f"Expected string, got {type(doc)}")  # Error if non-stri
    return filtered_corpus
```

In [ ]:
```
def preprocess(corpus, keep_list, cleaning=True, stemming=False, stem_type=None, lemma
    if not isinstance(corpus, list):
        raise TypeError("Input corpus must be a list.")

    # Handle dictionaries: extract text if necessary
    if all(isinstance(doc, dict) for doc in corpus):
        corpus = [doc.get('text', '') for doc in corpus]

    # Validate that all elements are strings
    corpus = [str(doc) for doc in corpus if isinstance(doc, (str, int, float))]

    if cleaning:
        corpus = text_clean(corpus, keep_list)

    if remove_stopwords:
        corpus = stopwords_removal(corpus)

    if lemmatization:
        corpus = lemmatize(corpus)
```

```
    if stemming:
        corpus = stem(corpus, stem_type)

    # Rejoin tokens into properly spaced sentences
    corpus = [' '.join(doc.split()) for doc in corpus]

    return corpus
```

Data preprocessing pipeline for the TF-IDF Vectorizer

In [ ]:
```
#Preprocessing with Lemmatization
common_dot_words = ['U.S.', 'Mr.', 'Mrs.', 'D.C.']
```

In [ ]:
```
preprocessed_corpus_questions = preprocess(corpus_questions, keep_list = common_dot_wo

preprocessed_corpus_questions[0:99]
```

In [ ]:
```
preprocessed_corpus_answers = preprocess(corpus_answers, keep_list = common_dot_words,

preprocessed_corpus_answers[0:99]
```

Tfidf Vectorizer Q & A

In [ ]:
```
#TfIdfVectorize Questions and Answers
vectorizer_tfidf_questions = TfidfVectorizer()
vectorizer_tfidf_answers = TfidfVectorizer()
```

In [ ]:
```
#Fit the questions
tfidf_questions_matrix = vectorizer_tfidf_questions.fit_transform(preprocessed_corpus_
```

In [ ]:
```
#Fit the answers
tfidf_answers_matrix = vectorizer_tfidf_answers.fit_transform(preprocessed_corpus_answ
```

Display obtained features and TF-IDF matrix

In [ ]:
```
#Print the Question features, toarray, and shape
print(vectorizer_tfidf_questions.get_feature_names_out())
print(tfidf_questions_matrix.toarray())
print("The shape of the TF-IDF Matrix: ", tfidf_questions_matrix.shape)
```

In [ ]:
```
#Print the Answer features, toarray, and shape
print(vectorizer_tfidf_answers.get_feature_names_out())
print(tfidf_answers_matrix.toarray())
print("The shape of the TF-IDF Matrix: ", tfidf_answers_matrix.shape)
```

In [ ]:
```
#Display feature names for the Questions
feature_names = vectorizer_tfidf_questions.get_feature_names_out()
for col in tfidf_questions_matrix.nonzero()[1]:
    print(feature_names[col], " - ", tfidf_questions_matrix[0, col])
```

In [ ]:
```
#Display feature names for the Answers
feature_names = vectorizer_tfidf_answers.get_feature_names_out()
```

```
for col in tfidf_answers_matrix.nonzero()[1]:
    print(feature_names[col], " - ", tfidf_answers_matrix[0, col])
```

TF-IDF and seq2seq Chatbots

Model: TF-IDF Transformer Chatbot

Tokenize text and covnert to matrix format

In [ ]:
```
#Tokenize and Fit Question Data
vectorizer = CountVectorizer(stop_words='english')
X_vec = vectorizer.fit_transform(combined_questions)
```

Transform using TF-IDF

In [ ]:
```
#Apply TF-IDF
tfidf = TfidfTransformer()
X_tfidf = tfidf.fit_transform(X_vec)
```

In [ ]:
```
X_tfidf
```

Create conversation function

Calculate angle between words in order to match questions and answer

In [ ]:
```
#Create Conversation Function
#def conversation(im):
    global tfidf, combined_answers, X_tfidf
    Y_vec = vectorizer.transform(im)
    Y_tfidf = tfidf.fit_transform(Y_vec)
    cos_sim = np.rad2deg(np.arccos(max(cosine_similarity(Y_tfidf, X_tfidf)[0])))
    if cos_sim > 60 :
        return "Sorry, I did not quite understand the question"
    else:
        return combined_answers[np.argmax(cosine_similarity(Y_tfidf, X-tfidf)[0])]
```

In [ ]:
```
def conversation(user_input):
    global vectorizer, X_tfidf, combined_answers  # Ensure these are accessible

    # Transform the user input into a TF-IDF vector
    Y_tfidf = vectorizer.transform(user_input)

    # Compute cosine similarity between user input and the training data
    similarity_scores = cosine_similarity(Y_tfidf, X_tfidf)

    # Find the most similar question
    if np.max(similarity_scores) < 0.1:  # Threshold to handle low similarity
        return "Sorry, I did not quite understand the question."
    else:
        best_match_index = np.argmax(similarity_scores[0])
        return combined_answers[best_match_index]
```

TF-IDF chatbot test function

In [ ]:
```
def tfidf_chatbot():
    usr = input("Please enter your name: ")
    print("Welcome to Amazon's Toys & Games Q&A Support. How can I help you?")
    while True:
        im = input("{}: ".format(usr))
        if im.lower() == 'bye':
            print("Q&A Support: Bye!")
            break
        else:
            response = conversation([im])
            print("Q&A Support: " + response)
```

Model: seq2seq Chatbot

In [ ]:
```
print(len(combined_questions))
print(len(combined_answers))
```

In [ ]:
```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
```

Encoder

In [ ]:
```
# Initialize the tokenizer
tokenizer = Tokenizer()

# Fit the tokenizer on both questions and answers
tokenizer.fit_on_texts(questions + answers)

# encoder_input_data
tokenized_questions = tokenizer.texts_to_sequences(combined_questions)
maxlen_questions = max( [ len(x) for x in tokenized_questions ] )
padded_questions = pad_sequences( tokenized_questions , maxlen=maxlen_questions , padd
encoder_input_data = np.array( padded_questions )
print( encoder_input_data.shape , maxlen_questions )
```

Decoder

In [ ]:
```
# decoder_input_data
tokenized_answers = tokenizer.texts_to_sequences(combined_answers)
maxlen_answers = max( [ len(x) for x in tokenized_answers ] )
padded_answers = pad_sequences( tokenized_answers , maxlen=maxlen_answers , padding='p
decoder_input_data = np.array( padded_answers )
print( decoder_input_data.shape , maxlen_answers )
```

In [ ]:
```
from keras.utils import to_categorical

VOCAB_SIZE = len(tokenizer.word_index) + 1

# decoder_output_data
tokenized_answers = tokenizer.texts_to_sequences( combined_answers )
```

```
# It removes the first element (corresponding to <START>) from each tokenized answer s
# This is because the decoder's input will be <START> followed by the actual sequence,
# and the target (output) should be the actual sequence.
for i in range(len(tokenized_answers)) :
    tokenized_answers[i] = tokenized_answers[i][1:]

# It pads the sequences of tokenized answers to make them all have the same length.
padded_answers = pad_sequences( tokenized_answers , maxlen=maxlen_answers , padding='p

# Let's perform one-hot encoding on the padded sequences.
# VOCAB_SIZE is the size of the vocabulary,
# and each element in the one-hot encoding corresponds to a word in the vocabulary.
onehot_answers = to_categorical( padded_answers , VOCAB_SIZE )

# Let's convert the one-hot encoded sequences into a NumPy array.
decoder_output_data = np.array( onehot_answers )
print( decoder_output_data.shape )
```

In [ ]:

```
from keras.preprocessing.sequence import pad_sequences
import numpy as np
from tensorflow.keras.losses import SparseCategoricalCrossentropy

# Tokenize and remove <START> token
tokenized_answers = tokenizer.texts_to_sequences(combined_answers)
for i in range(len(tokenized_answers)):
    tokenized_answers[i] = tokenized_answers[i][1:]

# Pad sequences
padded_answers = pad_sequences(tokenized_answers, maxlen=maxlen_answers, padding='post

# Use tokenized, padded answers directly as sparse targets
decoder_output_data = padded_answers

print("Decoder output data shape:", decoder_output_data.shape)

# Adjust model's loss to use sparse categorical crossentropy
model.compile(
    optimizer=tf.keras.optimizers.RMSprop(),
    loss=SparseCategoricalCrossentropy(from_logits=False),
    metrics=['accuracy']
)

# Train the model
model.fit([encoder_input_data, decoder_input_data], decoder_output_data, batch_size=32
model.save('model.keras')
```

Define seq2seq layers (input, embedding, LSTM)

In [ ]:

```
import tensorflow as tf

# Calculate max lengths
maxlen_questions = max(len(seq) for seq in tokenized_questions)
maxlen_answers = max(len(seq) for seq in tokenized_answers)

# Pad sequences to ensure uniform shape
from keras.preprocessing.sequence import pad_sequences
encoder_input_data = pad_sequences(tokenized_questions, maxlen=maxlen_questions, paddi
decoder_input_data = pad_sequences(tokenized_answers, maxlen=maxlen_answers, padding='
```

```python
# Check shapes
print("Encoder input data shape:", encoder_input_data.shape)
print("Decoder input data shape:", decoder_input_data.shape)

VOCAB_SIZE = len(tokenizer.word_index) + 1


# We need to define an input layer for the encoder
# shape=(maxlen_questions,) specifies the shape of the input,
# where maxlen_questions is the maximum length of the input sequences.
encoder_inputs = tf.keras.layers.Input(shape=(maxlen_questions,))

# It adds an embedding layer to the encoder.
# VOCAB_SIZE is the size of the vocabulary.
# 200 is the dimensionality of the embedding.
# mask_zero=True masks the padded zeros in the input sequences.
encoder_embedding = tf.keras.layers.Embedding( VOCAB_SIZE, 200 , mask_zero=True ) (enc

# It adds an LSTM layer to the encoder.
# 200 is the number of units in the LSTM layer.
# return_state=True returns the hidden state and cell state as part of the output.
encoder_outputs , state_h , state_c = tf.keras.layers.LSTM( 200 , return_state=True )(

# Let's create a list containing the hidden state (state_h) and cell state (state_c) o
encoder_states = [ state_h , state_c ]

# Let's define an input layer for the decoder.
# shape=(maxlen_answers,) specifies the shape of the input,
# where maxlen_answers is the maximum length of the output sequences.
decoder_inputs = tf.keras.layers.Input(shape=(maxlen_answers,))

# Let's adds an embedding layer to the decoder with the same configuration as the enco
decoder_embedding = tf.keras.layers.Embedding( VOCAB_SIZE, 200 , mask_zero=True) (deco

# Decoder LSTM Layer:
# We are adding an LSTM layer to the decoder.
#return_state=True returns the hidden state and cell state as part of the output.
#return_sequences=True returns the full sequence of outputs for each timestep.
decoder_lstm = tf.keras.layers.LSTM( 200 , return_state=True , return_sequences=True )
decoder_outputs , _ , _ = decoder_lstm ( decoder_embedding , initial_state=encoder_sta

# Let's add the Decoder Dense Layer:
# It adds a dense layer to the decoder with a softmax activation function.
# The output is the probability distribution over the vocabulary for each timestep.
decoder_dense = tf.keras.layers.Dense( VOCAB_SIZE , activation=tf.keras.activations.so
output = decoder_dense ( decoder_outputs )

# It constructs the final model with both encoder and decoder inputs and the output.
model = tf.keras.models.Model([encoder_inputs, decoder_inputs], output )

# Let's compiles the model, specifying the RMSprop optimizer and categorical cross-ent
model.compile(optimizer=tf.keras.optimizers.RMSprop(), loss='categorical_crossentropy'

# Print the model architecture, including layer names, types, output shapes, and the n
model.summary()
```

Train and Save seq2seq model

In [ ]:
```
#Train and Save the model
model.fit([encoder_input_data, decoder_input_data], decoder_output_data, batch_size=32
model.save('model.keras')
```

Create functions for seq2seq model

In [ ]:
```
#Define function, make_inference_models, which is responsible for creating inference m
def make_inference_models():

    # model for the encoder.
    encoder_model = tf.keras.models.Model(encoder_inputs, encoder_states)

    # Decoder State Inputs:
    decoder_state_input_h = tf.keras.layers.Input(shape=( 200 ,))
    decoder_state_input_c = tf.keras.layers.Input(shape=( 200 ,))

    #Decoder States Inputs List
    decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]

    # Decoder Outputs and States
    decoder_outputs, state_h, state_c = decoder_lstm(
        decoder_embedding , initial_state=decoder_states_inputs)
    decoder_states = [state_h, state_c]

    # Decoder Dense Layer
    decoder_outputs = decoder_dense(decoder_outputs)

    #Decoder Model
    # It returns both the encoder and decoder models
    decoder_model = tf.keras.models.Model(
        [decoder_inputs] + decoder_states_inputs,
        [decoder_outputs] + decoder_states)

    return encoder_model , decoder_model
```

In [ ]:
```
#Define function str_to_tokens to convert string into a sequence of tokens
def str_to_tokens( sentence : str ):
    # converts the input sentence to lowercase
    # and splits the sentence into a list of words
    words = sentence.lower().split()

    # Tokenization
    # It iterates through the list of words and uses the Keras Tokenizer (tokenizer)
    # to convert each word to its corresponding integer index.
    # The result is a list of tokenized indices representing the words in the input se
    tokens_list = list()
    for word in words:
        tokens_list.append( tokenizer.word_index[ word ] )

    #Padding Sequences
    #It takes the list of tokenized indices (tokens_list) and pads
    # or truncates the sequence to ensure it has the same length (maxlen_questions)
    # as expected by the model.
    # maxlen=maxlen_questions specifies the maximum length of the padded sequence.
    # padding='post' indicates that padding should be added to the end of the sequence
    return preprocessing.sequence.pad_sequences( [tokens_list] , maxlen=maxlen_questic
```

```
    # Output
    #The function returns the padded sequence of tokenized indices,
    #which will be used as input to the chatbot model during the inference phase.
```

Test TF-IDF and seq2seq chatbots

In [ ]:
```
#Call tfidf_chatbot function to test TF-IDF Chatbot
tfidf_chatbot()
```

In [ ]:
```
# 1. Inference Model Initialization
enc_model, dec_model = make_inference_models()

# 2. Interactive Loop
while True:
    # 3. User Input
    user_input = input('Enter question (type "bye" to exit): ')

    # 4. Check if the user wants to exit
    if user_input.lower() == 'bye':
        print('Goodbye!')
        break

    # 5. Encode User Input
    # It tokenizes and encodes the user's input using the encoder model
    # to obtain the initial states for the decoder.
    states_values = enc_model.predict(str_to_tokens(user_input))

    # 6. Initialize Target Sequence and Decoded Translation
    # It initializes the target sequence for the decoder with a single <START> token.
    # It sets up variables for stopping the generation loop and storing the decoded tr
    empty_target_seq = np.zeros((1, 1))
    empty_target_seq[0, 0] = tokenizer.word_index['start']
    stop_condition = False
    decoded_translation = ''

    # 7. Decoding Loop
    # It runs a loop until a stopping condition is met
    # The decoder model predicts the next word in the sequence (sampled_word_index)
    # based on the current target sequence and states.
    while not stop_condition:
        dec_outputs, h, c = dec_model.predict([empty_target_seq] + states_values)
        sampled_word_index = np.argmax(dec_outputs[0, -1, :])

        # 8. Word Lookup and Decoded Translation Update
        sampled_word = None
        for word, index in tokenizer.word_index.items():
            if sampled_word_index == index:
                decoded_translation += ' {}'.format(word)
                sampled_word = word

        # 9. Stopping Condition Check
        # It checks whether the generated word is the <END> token or if the length of
        # the generated translation exceeds a certain limit, signaling the end of the
        if sampled_word == 'end' or len(decoded_translation.split()) > maxlen_answers:
            stop_condition = True

        # Update Target Sequence and States
        # it updates the target sequence for the next iteration and the states for the
```

```
        empty_target_seq = np.zeros((1, 1))
        empty_target_seq[0, 0] = sampled_word_index
        states_values = [h, c]

    # 11. It prints the generated response after each iteration of the inference loop
    print('Chatbot:', decoded_translation)
```

**Though the TF-IDF Vectorizer chatbot had it's limitations it tried to answer all questions asked even if not always a completely relevant answer. This combined with the inability to properly train the decoder output data for the seq2seq model based on memory or time constraints shows that the TF-IDF is the only choice on a dataset of this size and the limited computational resources available to me at this time.

Project Summary

This project, Amazon Toys & Games Q&A Chatbots, focused on developing a chatbot system using Amazon's "Toys & Games" Q&A dataset alongside additional data sources for enhanced context. The project involved cleaning and preprocessing both datasets to prepare them for natural language processing (NLP) tasks. This included data preprocessing, tokenization, and text normalization. While multiple NLP approaches were tested, only SpaCy's Named Entity Recognition (NER) proved effective for this specific dataset, enabling the extraction of meaningful entities to support conversational functionality. Ultimately, the TF-IDF-based chatbot was the only viable solution for this dataset as the seq2seq based chatbot exceeded memory allocations.

In [ ]: