The World of Haiku

by Chao Xu

I: Introduction

Haiku is a type of short form poetry that originated in Japan. For this project, I want to use NLP techniques I learned from this class to do the text summarization, data analysis, and finally haiku generation.

To see my project on GutHub: https://github.com/Lilacbibi/CISB63_final

Data used for this project:

- **Haiku introduction**: https://en.wikipedia.org/wiki/Haiku
- **Haiku datase**t: https://www.kaggle.com/datasets/hjhalani30/haiku-dataset

The main steps are as follows:

- Use text summarization (**TF-IDF**) to generate a brief introduction to haiku and visualize Named Entities with **spaCy**.
- Use **TextBolb** to translate original Japanese haiku.
- Visualize haiku Dependency Parse using **spaCy**
- Create **Word2Vec** embeddings and visualize word vectors using **UMAP**
- Clustering haiku using **NMF**
- Visualize top words of each topic using **matplotlib** and clustered topics using **WordCloud**
- Generate haiku using **LSTM**
- To clean the text, multipule NLP and EDA techniques are used such as **ReGex**, **Tokenization**, **stop words**, **handling missing values**, etc.

II: What is Haiku? —*A introduction using text summarization*

Import libraries

In [1]:

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import sent_tokenize, word_tokenize
from sklearn.feature_extraction.text import TfidfVectorizer

import numpy as np
import pandas as pd
import re
import matplotlib.pyplot as plt

import warnings
warnings.filterwarnings('ignore')
```

Text Preparation

The text below is copied directly from Wikipedia: https://en.wikipedia.org/wiki/Haiku

In [2]:

```
intro = """
Haiku (俳句, listenⓘ) is a type of short form poetry that originated in Japan. Traditional Japanese haiku consist of th

Haiku originated as an opening part of a larger Japanese poem called renga. These haiku written as an opening stanza we

Originally from Japan, haiku today are written by authors worldwide. Haiku in English and haiku in other languages have

In Japanese, haiku are traditionally printed as a single line, while haiku in English often appear as three lines, alth
"""
```

Split the text into sentences

In [3]:

```
# Split the text into sentences keeping the original format
original_sentences = intro.strip().split('.')
```

```
#Print the whole text
print(original_sentences)
```

```
['Haiku (俳句, listenⓘ) is a type of short form poetry that originated in Japan', ' Traditional Japanese haiku consist
of three phrases composed of 17 phonetic units (called on in Japanese, which are similar to syllables) in a 5, 7, 5 pat
tern;[1] that include a kireji, or "cutting word";[2] and a kigo, or seasonal reference', ' Similar poems that do not a
dhere to these rules are generally classified as senryū', '[3]\n\nHaiku originated as an opening part of a larger Japan
ese poem called renga', ' These haiku written as an opening stanza were known as hokku and over time they began to be w
ritten as stand-alone poems', ' Haiku was given its current name by the Japanese writer Masaoka Shiki at the end of the
19th century', '[4]\n\nOriginally from Japan, haiku today are written by authors worldwide', ' Haiku in English and hai
ku in other languages have different styles and traditions while still incorporating aspects of the traditional haiku f
orm', ' Non-Japanese haiku vary widely on how closely they follow traditional elements', ' Additionally, a minority mov
ement within modern Japanese haiku (現代俳句, gendai-haiku), supported by Ogiwara Seisensui and his disciples, has varie
d from the tradition of 17 on as well as taking nature as their subject', '\n\nIn Japanese, haiku are traditionally pri
nted as a single line, while haiku in English often appear as three lines, although variations exist', ' There are seve
ral other forms of Japanese poetry related to haiku, such as tanka, as well as other art forms that incorporate haiku,
such as haibun and haiga', '']
```

Some characters can be removed to make the text clean:

- In-text citation numbers
- Extra space at the beginning of some sentences
- The ', listenⓘ' is an audio link that we do not need
- There are some Japanese words cannot be removed as well as parentheses

Clean the text with RegEx

I Create a function to clean the original text with **RegEx**.

In [4]:

```
def clean_text(text):
    #Create a list of replecements
    replacements = [
        (r', listenⓘ', ''),
        (r'(\[\d+\])', ''),
        (r'^ ', '')
    ]
    for old, new in replacements:
        text = re.sub(old, new, text)
    return text
```

In [5]:
```
#Apply the function to create clean sentences
clean_sentences = []
for sentence in original_sentences:
    clean_sentence = clean_text(sentence)
    clean_sentences.append(clean_sentence)
```

Print the sentences to see the result:

In [6]:
```
for sentence in clean_sentences:
    print(sentence)
```

```
Haiku (俳句) is a type of short form poetry that originated in Japan
Traditional Japanese haiku consist of three phrases composed of 17 phonetic units (called on in Japanese, which are sim
ilar to syllables) in a 5, 7, 5 pattern; that include a kireji, or "cutting word"; and a kigo, or seasonal reference
Similar poems that do not adhere to these rules are generally classified as senryū


Haiku originated as an opening part of a larger Japanese poem called renga
These haiku written as an opening stanza were known as hokku and over time they began to be written as stand-alone poem
s
Haiku was given its current name by the Japanese writer Masaoka Shiki at the end of the 19th century


Originally from Japan, haiku today are written by authors worldwide
Haiku in English and haiku in other languages have different styles and traditions while still incorporating aspects of
the traditional haiku form
Non-Japanese haiku vary widely on how closely they follow traditional elements
Additionally, a minority movement within modern Japanese haiku (現代俳句, gendai-haiku), supported by Ogiwara Seisensui
and his disciples, has varied from the tradition of 17 on as well as taking nature as their subject


In Japanese, haiku are traditionally printed as a single line, while haiku in English often appear as three lines, alth
ough variations exist
There are several other forms of Japanese poetry related to haiku, such as tanka, as well as other art forms that incor
porate haiku, such as haibun and haiga
```

Now the text looks nice and clean. It's time to preprocess the text for the TF-IDF matrix:

Remove punctuation and stopwords

In [7]:

```
# Preprocess the text (remove punctuation and stopwords)
nltk.download('punkt')
nltk.download('stopwords')
stop_words = set(stopwords.words('english'))
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\lilac\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\lilac\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Create a function to remove puctuations, parentheses, stop words and lower the cases.

In [8]:

```
def preprocess_text(text):
    words = word_tokenize(text)
    words = [word.lower() for word in words if word.isalnum()]
    words = [word for word in words if word not in stop_words]
    return ' '.join(words)
```

In [9]:

```
#sentences = sent_tokenize(intro)
preprocessed_sentences = []
for sentence in clean_sentences:
    preprocessed_sentence = preprocess_text(sentence)
    preprocessed_sentences.append(preprocessed_sentence)
```

In [10]:

```
print(preprocessed_sentences)
```

```
['haiku 俳句 type short form poetry originated japan', 'traditional japanese haiku consist three phrases composed 17 ph
onetic units called japanese similar syllables 5 7 5 pattern include kireji cutting word kigo seasonal reference', 'sim
ilar poems adhere rules generally classified senryū', 'haiku originated opening part larger japanese poem called reng
a', 'haiku written opening stanza known hokku time began written poems', 'haiku given current name japanese writer masa
oka shiki end 19th century', 'originally japan haiku today written authors worldwide', 'haiku english haiku languages d
ifferent styles traditions still incorporating aspects traditional haiku form', 'haiku vary widely closely follow tradi
tional elements', 'additionally minority movement within modern japanese haiku 現代俳句 supported ogiwara seisensui disc
iples varied tradition 17 well taking nature subject', 'japanese haiku traditionally printed single line haiku english
often appear three lines although variations exist', 'several forms japanese poetry related haiku tanka well art forms
incorporate haiku haibun haiga', '']
```

There is an empty string as the last list element. I want to remove it before create the matrix.

In [11]:

```
preprocessed_sentences = preprocessed_sentences[:-1]
print(preprocessed_sentences)
```

```
['haiku 俳句 type short form poetry originated japan', 'traditional japanese haiku consist three phrases composed 17 ph
onetic units called japanese similar syllables 5 7 5 pattern include kireji cutting word kigo seasonal reference', 'sim
ilar poems adhere rules generally classified senryū', 'haiku originated opening part larger japanese poem called reng
a', 'haiku written opening stanza known hokku time began written poems', 'haiku given current name japanese writer masa
oka shiki end 19th century', 'originally japan haiku today written authors worldwide', 'haiku english haiku languages d
ifferent styles traditions still incorporating aspects traditional haiku form', 'haiku vary widely closely follow tradi
tional elements', 'additionally minority movement within modern japanese haiku 現代俳句 supported ogiwara seisensui disc
iples varied tradition 17 well taking nature subject', 'japanese haiku traditionally printed single line haiku english
often appear three lines although variations exist', 'several forms japanese poetry related haiku tanka well art forms
incorporate haiku haibun haiga']
```

Calculate the TF-IDF scores and generate summary

In [12]:

```
# Calculate TF-IDF scores
tfidf = TfidfVectorizer()
tfidf_matrix = tfidf.fit_transform(preprocessed_sentences)
tfidf_scores = tfidf_matrix.sum(axis=1)

top_sentence_indices = np.argsort(tfidf_scores, axis=0)[-4:]
```

In [13]:

```
def generate_summary(preprocessed_sentences, clean_sentences, n=3):

    #Create a matrix with TF-IDF scores and calculate the total scores of each sentences
    tfidf = TfidfVectorizer()
    tfidf_matrix = tfidf.fit_transform(preprocessed_sentences)
    tfidf_scores = tfidf_matrix.sum(axis=1)

    #Find the most important n sentences and sort the indicies to the original order
    top_sentence_indices = np.argsort(tfidf_scores, axis=0)[-n:]
    new_top_sentence_indices = np.sort(top_sentence_indices, axis=0)
    top_sentences = []


    #Generate the summary based on the clean text and top sentence indicies
    summary = ''
    for i in range(len(new_top_sentence_indices)):
        index = new_top_sentence_indices.tolist()[i][0]
        val = clean_sentences[index]
        #Indent the first line if it is the beginning a new paragraph (\n\n)
        val = re.sub(r'\n\n', '\n\t', val)
        #Indent the first line if it is the beginning of the summary
        if i==0:
            val = '\t' + val

        summary += ''.join(val) +  '. '

    return summary
```

In [14]:

```
summary = generate_summary(preprocessed_sentences, clean_sentences, 4)
print(summary)
```

```
	Traditional Japanese haiku consist of three phrases composed of 17 phonetic units (called on in Japanese, which
are similar to syllables) in a 5, 7, 5 pattern; that include a kireji, or "cutting word"; and a kigo, or seasonal refer
ence. Haiku in English and haiku in other languages have different styles and traditions while still incorporating aspe
cts of the traditional haiku form. Additionally, a minority movement within modern Japanese haiku (現代俳句, gendai-haik
u), supported by Ogiwara Seisensui and his disciples, has varied from the tradition of 17 on as well as taking nature a
s their subject.
	In Japanese, haiku are traditionally printed as a single line, while haiku in English often appear as three lin
es, although variations exist.
```

Visualize Named Entities Using spaCy

In [15]:

```
import spacy
from spacy import displacy
nlp = spacy.load('en_core_web_sm')

colors = {'ORG': 'linear-gradient(to bottom right, #9E7BFF, #BDEDFF)',
          'DATE': 'radial-gradient(#FFFFCC, #50C878)'}
options = {'ents': ['ORG', 'DATE'], 'colors':colors}

text = nlp(summary)
displacy.render(text, style='ent', jupyter=True, options=options)
```

Traditional Japanese haiku consist of three phrases composed of 17 phonetic units (called on in Japanese, which are similar to syllables) in a 5, 7 DATE , 5 pattern; that include a kireji, or "cutting word"; and a kigo, or seasonal reference. Haiku in English and haiku in other languages have different styles and traditions while still incorporating aspects of the traditional haiku form. Additionally, a minority movement within modern Japanese haiku (現代俳句, gendai-haiku ORG), supported by Ogiwara Seisensui and his disciples, has varied from the tradition of 17 on as well as taking nature as their subject.

In Japanese, haiku are traditionally printed as a single line, while haiku in English often appear as three lines, although variations exist.

III: The beauty of original Japanese Haiku —*Translating Japanese Haiku using TextBlob*

I select some haiku written by 松尾芭蕉(Matsuo Bashō), the most famous poet in Japan, to share the beauty of Japanese haiku.

Because these haiku are printed as a single line, so I divided them into three lines to adapt English format.

Import libraries

In [16]:

```
from textblob import TextBlob
```

In [17]:

```
def haiku_translation(text):
    haiku = TextBlob(text)
    print('Original Japanese Haiku:')
    #print('—' * 25)
    print(text)
    print('\nEnglish translation:')
    #print('—' * 25)
    print(haiku.translate(from_lang='ja', to='en'))
```

In [18]:

```
haiku_list = ['夕晴れや\n桜に涼む\n波の華', '古池や\n蛙飛びこむ\n水の音', '閑けさや\n岩にしみいる\n蝉の声']
```

In [19]:

```python
for haiku in haiku_list:
    haiku_translation(haiku)
    print('… ' * 15, '\n')
```

```
Original Japanese Haiku:
夕晴れや
桜に涼む
波の華

English translation:
Sunny evening
Cool down on cherry blossoms
Wavy flower
… … … … … … … … … … … … … … … 


Original Japanese Haiku:
古池や
蛙飛びこむ
水の音

English translation:
old pond
Jump in the frog
Sound of water
… … … … … … … … … … … … … … … 


Original Japanese Haiku:
閑けさや
岩にしみいる
蝉の声

English translation:
Calmness
Push in the rock
Cicada's voice
… … … … … … … … … … … … … … … 
```

IV: Haiku Dataset Analysis

The dataset I use can be found in Kaggle: https://www.kaggle.com/datasets/hjhalani30/haiku-dataset

Load the dataset and Handle Missing Values

In [20]:

```python
data = pd.read_csv('data/all_haiku.csv')
data.head()
```

Out[20]:

	Unnamed: 0	0	1	2	source	hash
0	0	fishing boats	colors of	the rainbow	tempslibres	FISHINGBOATSCOLORSOFTHERAINBOW
1	1	ash wednesday--	trying to remember	my dream	tempslibres	ASHWEDNESDAYTRYINGTOREMEMBERMYDREAM
2	2	snowy morn--	pouring another cup	of black coffee	tempslibres	SNOWYMORNPOURINGANOTHERCUPOFBLACKCOFFEE
3	3	shortest day	flames dance	in the oven	tempslibres	SHORTESTDAYFLAMESDANCEINTHEOVEN
4	4	haze	half the horse hidden	behind the house	tempslibres	HAZEHALFTHEHORSEHIDDENBEHINDTHEHOUSE

In [21]:

```python
data.tail()
```

Out[21]:

	Unnamed: 0	0	1	2	source	hash
144118	118007	I'm not asking did	you say it nor clarify	what you said neither	twaiku	IMNOTASKINGDIDYOUSAYITNORCLARIFYWHATYOUSAIDNEI...
144119	118008	You are truly a	moron or a liar I'm	inclined to think both	twaiku	YOUARETRULYAMORONORALIARIMINCLINEDTOTHINKBOTH
144120	118009	Ain't no selfie on	this earth that's gonna make me	like Theresa May	twaiku	AINTNOSELFIEONTHISEARTHTHATSGONNAMAKEMELIKETHE...
144121	118010	is doing a great	job turning Independents	into Democrats	twaiku	ISDOINGAGREATJOBTURNINGINDEPENDENTSINTODEMOCRATS
144122	118011	Wanted to send a	quick follow up on if the	blood is loud Talk soon	twaiku	WANTEDTOSENDAQUICKFOLLOWUPONIFTHEBLOODISLOUDTA...

Column '0', '1', '2' represent three lines of Haiku. Although haiku in this dataset have three lines, not all of them have the correct 5-7-5 syllable pattern. Based on the dataset, I assume the haiku generated later will not follow the 5-7-5 rules. Instead, it will generate haiku-like poem with typical 3 lines.

In [22]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144123 entries, 0 to 144122
Data columns (total 6 columns):
 #   Column      Non-Null Count   Dtype 
---  ------      --------------   ----- 
 0   Unnamed: 0  144123 non-null  int64 
 1   0           144123 non-null  object
 2   1           144123 non-null  object
 3   2           144122 non-null  object
 4   source      144123 non-null  object
 5   hash        144122 non-null  object
dtypes: int64(1), object(5)
memory usage: 6.6+ MB
```

Check the missing value

In [23]:

```
data.isnull().sum()
```

Out[23]:

```
Unnamed: 0    0
0             0
1             0
2             1
source        0
hash          1
dtype: int64
```

There is only one missing value in the third line of one haiku.

In [24]:

```
#Check the haiku with null value
data[data['2'].isnull()]
```

Out[24]:

	Unnamed: 0	0	1	2	source	hash
18799	28	the busker	buttons his collar	NaN	sballas	NaN

In [25]:

```
#Drop the row
data = data.dropna()
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 144122 entries, 0 to 144122
Data columns (total 6 columns):
 #   Column      Non-Null Count   Dtype 
---  ------      --------------   ----- 
 0   Unnamed: 0  144122 non-null  int64 
 1   0           144122 non-null  object
 2   1           144122 non-null  object
 3   2           144122 non-null  object
 4   source      144122 non-null  object
 5   hash        144122 non-null  object
dtypes: int64(1), object(5)
memory usage: 7.7+ MB
```

In [26]:

```
data.isnull().sum()
```

Out[26]:

```
Unnamed: 0    0
0             0
1             0
2             0
source        0
hash          0
dtype: int64
```

Now there is no missing values in the dataset.

Preprocess Data

I only need to use three lines of haiku and the source. I will keep these lines and combine each line into a full haiku as a new column.

In [27]:

```
df_haiku = data[['source','0','1','2']]
df_haiku.head()
```

Out[27]:

	source	0	1	2
0	tempslibres	fishing boats	colors of	the rainbow
1	tempslibres	ash wednesday--	trying to remember	my dream
2	tempslibres	snowy morn--	pouring another cup	of black coffee
3	tempslibres	shortest day	flames dance	in the oven
4	tempslibres	haze	half the horse hidden	behind the house

Add a new column with all three lines of each haiku

In [28]:

```python
df_haiku = df_haiku.assign(full=lambda x: (x['0']+' '+ x['1'] + ' ' + x['2']))
df_haiku.head()
```

Out[28]:

	source	0	1	2	full
0	tempslibres	fishing boats	colors of	the rainbow	fishing boats colors of the rainbow
1	tempslibres	ash wednesday--	trying to remember	my dream	ash wednesday-- trying to remember my dream
2	tempslibres	snowy morn--	pouring another cup	of black coffee	snowy morn-- pouring another cup of black coffee
3	tempslibres	shortest day	flames dance	in the oven	shortest day flames dance in the oven
4	tempslibres	haze	half the horse hidden	behind the house	haze half the horse hidden behind the house

Since there are not too many words in one haiku and it is necessary to have stop words in poems, I will not remove stop words in the dataset. However, I still need to remove alphanumeric characters and convert the text into lowercases just in case there are capital letters in the dataset.

Add a new column with cleaned text

In [29]:

```python
#Function to clean the full haiku sentence
def clean_haiku(text):
    words = [word.lower() for word in word_tokenize(text) if word.isalnum()]
    return ' '.join(words)
```

In [30]:

```
df_haiku['clean_text'] = df_haiku['full'].apply(lambda x: clean_haiku(x))
df_haiku.head()
```

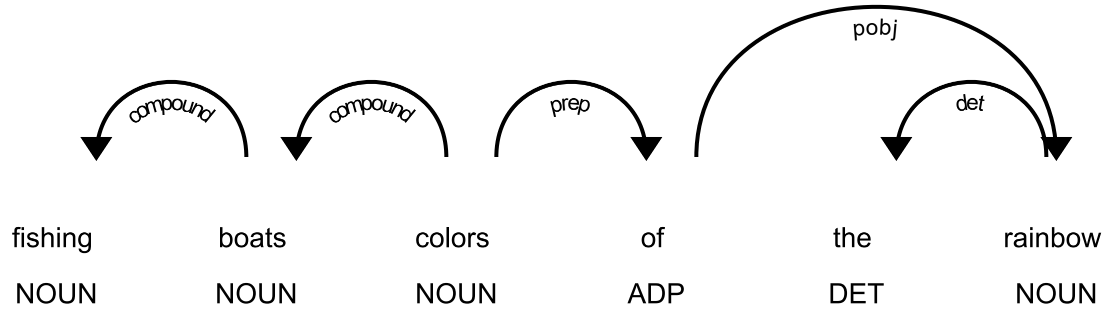
Out[30]:

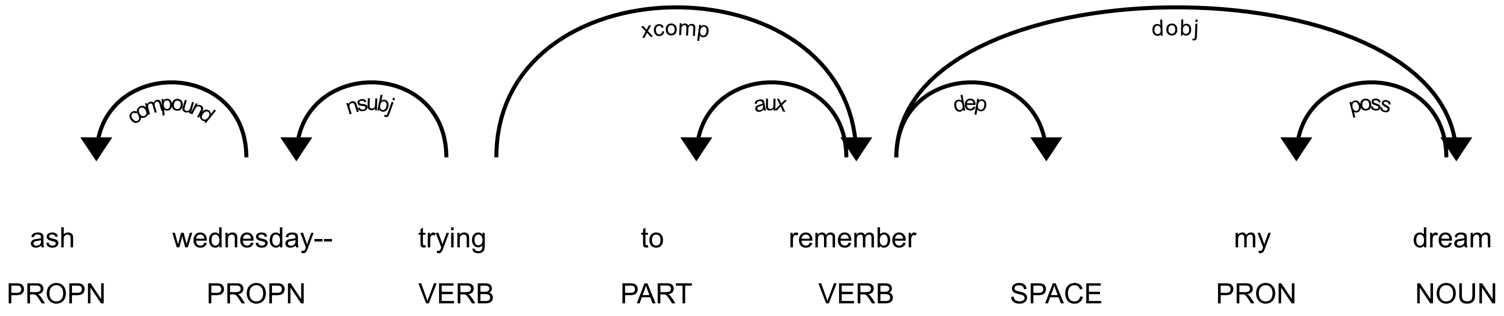
	source	0	1	2	full	clean_text
0	tempslibres	fishing boats	colors of	the rainbow	fishing boats colors of the rainbow	fishing boats colors of the rainbow
1	tempslibres	ash wednesday--	trying to remember	my dream	ash wednesday-- trying to remember my dream	ash wednesday trying to remember my dream
2	tempslibres	snowy morn--	pouring another cup	of black coffee	snowy morn-- pouring another cup of black coffee	snowy morn pouring another cup of black coffee
3	tempslibres	shortest day	flames dance	in the oven	shortest day flames dance in the oven	shortest day flames dance in the oven
4	tempslibres	haze	half the horse hidden	behind the house	haze half the horse hidden behind the house	haze half the horse hidden behind the house

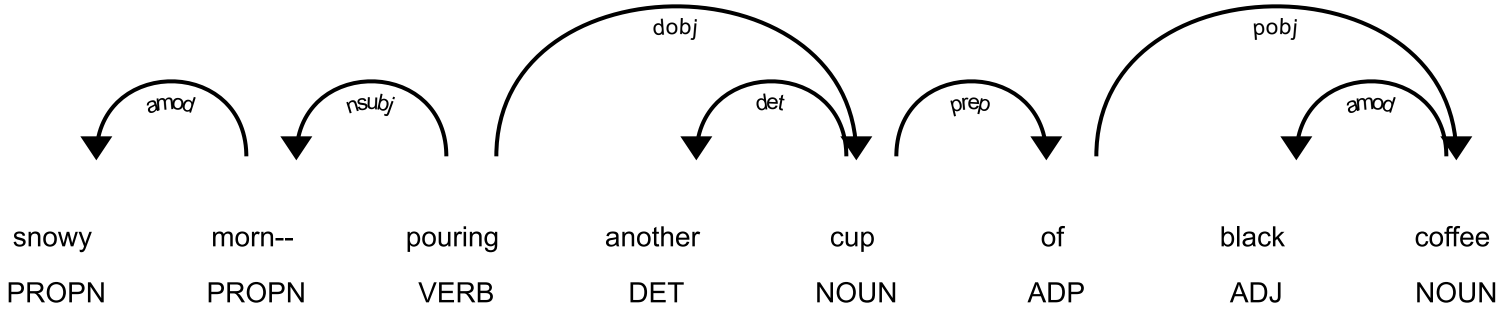
Visualize Haiku Dependency Parse Using spaCy

In [31]:

```
#Generate the frirst three rows of haiku
for n in range(3):
    doc = nlp(df_haiku['full'][n])

    #Display the haiku dpendency parse using spaCy
    displacy.render(doc, style="dep", jupyter=True, options={'distance':100})
```







Word2Vec Embeddings and Visualizing Vocabulary Using UMAP

Import Libraries

In [32]:
```
from gensim.models.word2vec import Word2Vec
import umap
```

Create a list with all the cleaned haiku and tokenize words in each row

In [33]:

```
#Create a list with all the cleaned haiku and tokenize words in each row
clean_text = list(df_haiku['clean_text'])
clean_text = [i.split() for i in clean_text]

#Show an example of the clean_text
clean_text[5]
```

Out[33]:

```
['low', 'sun', 'the', 'lady', 'in', 'red', 'on', 'high', 'heels']
```

Build and Train the Word2vec Model

In [34]:

```
model = Word2Vec(window=2, sg=1, #sg=1 is for skip-gram training algorithm
                 hs=0, negative=5, # for negative sampling
                 alpha=0.03, min_alpha=0.0007,
                 seed = 1001)

model.build_vocab(clean_text, progress_per=200)

model.train(clean_text, total_examples = len(clean_text),
            epochs=10, report_delay=1)
```

Out[34]:

```
(13955683, 18832260)
```

In [35]:

```
print(model)
```

```
Word2Vec<vocab=15223, vector_size=100, alpha=0.03>
```

This model has 15223 words and the size of the word vecors are set to 100.

In [36]:

```
#Put the vocabulary into a list
words = list(model.wv.key_to_index.keys())
len(words)
```

Out[36]:

```
15223
```

In [37]:

```
#Extract all vectors and put in an array
X = model.wv[model.wv.key_to_index]
X.shape
```

Out[37]:

```
(15223, 100)
```

Show some similar words

In [38]:

```python
print('The 5 most similar words to the word "house":')
model.wv.most_similar('house', topn=5)
```

```
The 5 most similar words to the word "house":
```

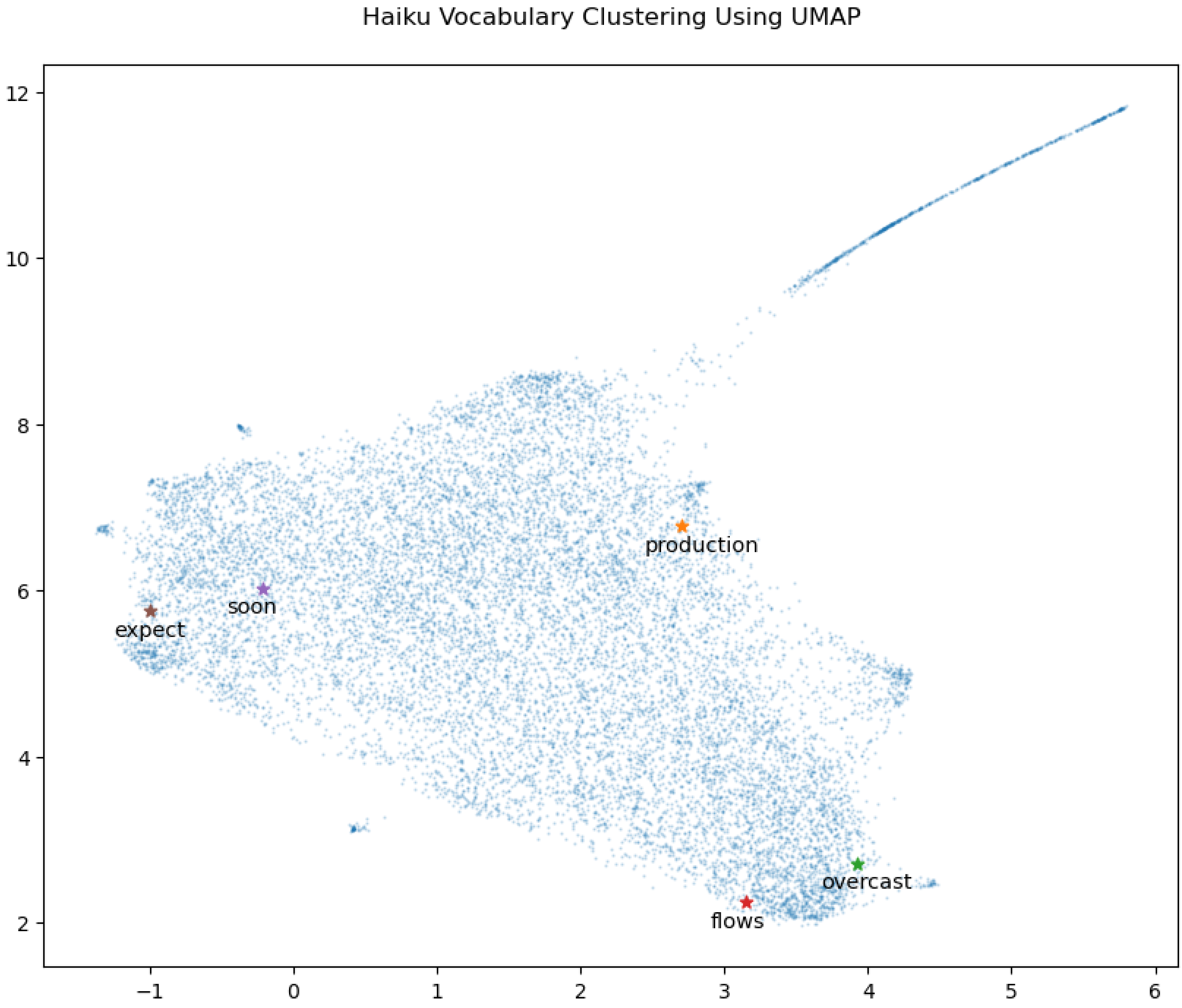
Out[38]:

```
[('apartment', 0.5960550308227539),
 ('closet', 0.5948619246482849),
 ('room', 0.5788634419441223),
 ('bookstore', 0.5610883235931396),
 ('garage', 0.5471485257148743)]
```

Haiku vocabulary visualization using UMAP

In [39]:

```python
import umap

cluster_embedding = umap.UMAP(n_neighbors=30, min_dist=0.0,
                              n_components=2, random_state=1001).fit_transform(X)

plt.figure(figsize=(10,8))
plt.scatter(cluster_embedding[:, 0], cluster_embedding[:, 1], s=0.5, alpha=0.2, cmap='Spectral')

#Generate 5 random numbers from the word list
import random
randomlist = random.sample(range(0, len(words)), 5)

#Add star marks for the chosen random words with corresponding annotation
for i in randomlist:
    plt.scatter(cluster_embedding[i,0], cluster_embedding[i,1], marker='*', s=40)
    plt.annotate(words[i], (cluster_embedding[i,0]-.25, cluster_embedding[i,1]-.3), fontsize=10.5)

plt.title('Haiku Vocabulary Clustering Using UMAP', pad=20)
plt.show()
```


Haiku Vocabulary Clustering Using UMAP
12
10
8
6
4
2
−1
0
1
2
3
4
5
6
production
soon
expect
overcast
flows

Clustering Haiku Using NMF and Visualizing Topics Using WordCloud

Create a TF-IDF matrix

In [40]:

```
vectorizer = TfidfVectorizer(max_df=0.95, min_df=2, stop_words='english')
tfidf_matrix = vectorizer.fit_transform(df_haiku['clean_text'])
```

Create a NMF Model with 15 topics

In [41]:

```
from sklearn.decomposition import NMF

# Apply NMF with 15 components
num_topics = 15

# Create an instance of the NMF (Nonnegative Matrix Factorization) class from scikit-learn.
nmf = NMF(n_components=num_topics, init='random', random_state=101)

# Apply the NMF to the TF-IDF matrix
nmf_matrix = nmf.fit_transform(tfidf_matrix)
```

Display the topics with top words

In [42]:

```
# Display the topics and the top words in each topic
feature_names = vectorizer.get_feature_names_out()

# This loop iterates over each topic extracted by NMF.
for topic_idx, topic in enumerate(nmf.components_):
    top_words_indices = topic.argsort()[-10:][::-1]
    top_words = [feature_names[i] for i in top_words_indices]
    # The join method is used to concatenate the words into a single string, separated by commas.
    print(f"Topic #{topic_idx + 1}:  {', '.join(top_words)}")
    print('-' * 10)
```

```
Topic #1:  like, feel, look, looks, shit, does, feeling, feels, writing, lol
----------
Topic #2:  love, happy, fall, person, heart, hate, birthday, baby, friends, thank
----------
Topic #3:  people, hate, think, say, shit, need, things, stop, care, understand
----------
Topic #4:  gon, na, im, think, today, try, week, make, year, tonight
----------
Topic #5:  just, mean, does, say, need, wanted, trying, fuck, realized, let
----------
Topic #6:  want, does, make, makes, eat, talk, baby, things, sleep, home
----------
Topic #7:  time, long, remember, work, think, year, waste, night, having, come
----------
Topic #8:  really, need, shit, good, hate, think, wish, bad, man, work
----------
Topic #9:  did, think, today, say, wish, said, things, come, lol, night
----------
Topic #10:  got, ta, shit, ai, say, fuck, work, man, cause, way
----------
Topic #11:  day, good, today, happy, hope, morning, birthday, great, night, new
----------
Topic #12:  going, im, today, sleep, work, way, tomorrow, home, tonight, shit
----------
Topic #13:  know, does, let, say, person, better, tell, good, right, anymore
----------
Topic #14:  wan, na, talk, make, home, work, sleep, kinda, come, friends
----------
Topic #15:  life, ca, make, believe, right, best, wait, need, better, things
----------
```

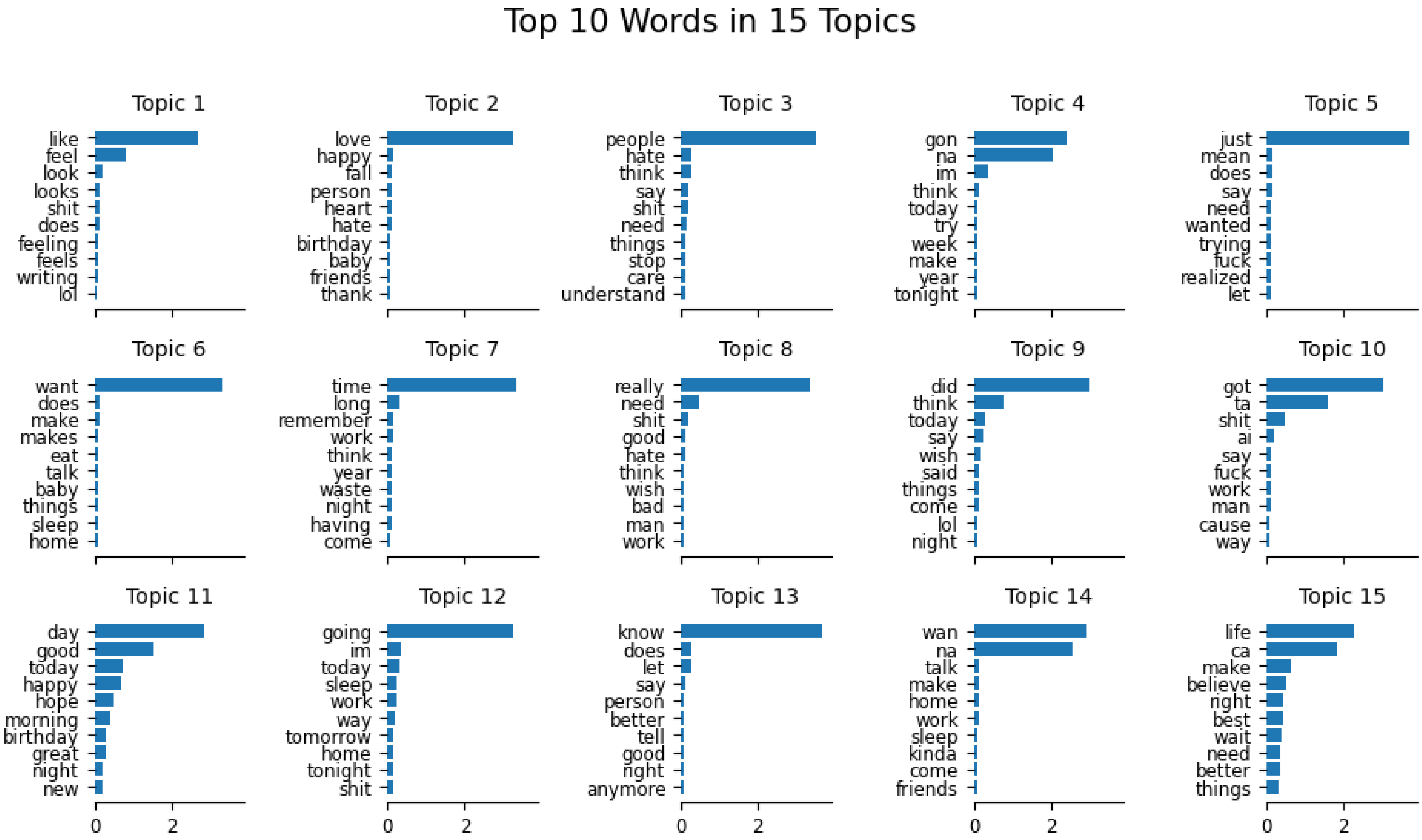
Visualize the top words in bar plot

In [43]:

```
def plot_top_words(model, feature_names, n_top_words, title):
    fig, axes = plt.subplots(3, 5, figsize=(9.5,5.5), sharex=True)
    axes = axes.flatten()
    for topic_idx, topic in enumerate(nmf.components_):
        top_words_indices = topic.argsort()[-n_top_words:]
        top_words = feature_names[top_words_indices]
        weights = topic[top_words_indices]

        ax = axes[topic_idx]
        ax.barh(top_words, weights, height=0.8)
        ax.set_title(f"Topic {topic_idx +1}", fontdict={"fontsize": 10})
        ax.tick_params(axis="both", which="major", labelsize=8.5)
        for i in "top right left".split():
            ax.spines[i].set_visible(False)
        fig.suptitle(title, fontsize=15, y=1.01)
    plt.tight_layout()
```

In [44]:

```
plot_top_words(nmf, feature_names, n_top_words=10, title='Top 10 Words in 15 Topics')
```



Use WordCloud to Show Some of the Topics

Add a new column to the dataframe that labels each haiku with one of the topics

In [45]:

```
topic_results = nmf.transform(tfidf_matrix)
df_haiku['Topic'] = topic_results.argmax(axis=1)+1
```

In [46]:

```
df_haiku.head()
```

Out[46]:

	source	0	1	2	full	clean_text	Topic
0	tempslibres	fishing boats	colors of	the rainbow	fishing boats colors of the rainbow	fishing boats colors of the rainbow	11
1	tempslibres	ash wednesday--	trying to remember	my dream	ash wednesday-- trying to remember my dream	ash wednesday trying to remember my dream	15
2	tempslibres	snowy morn--	pouring another cup	of black coffee	snowy morn-- pouring another cup of black coffee	snowy morn pouring another cup of black coffee	11
3	tempslibres	shortest day	flames dance	in the oven	shortest day flames dance in the oven	shortest day flames dance in the oven	11
4	tempslibres	haze	half the horse hidden	behind the house	haze half the horse hidden behind the house	haze half the horse hidden behind the house	7

Show some of the topics using WordCloud

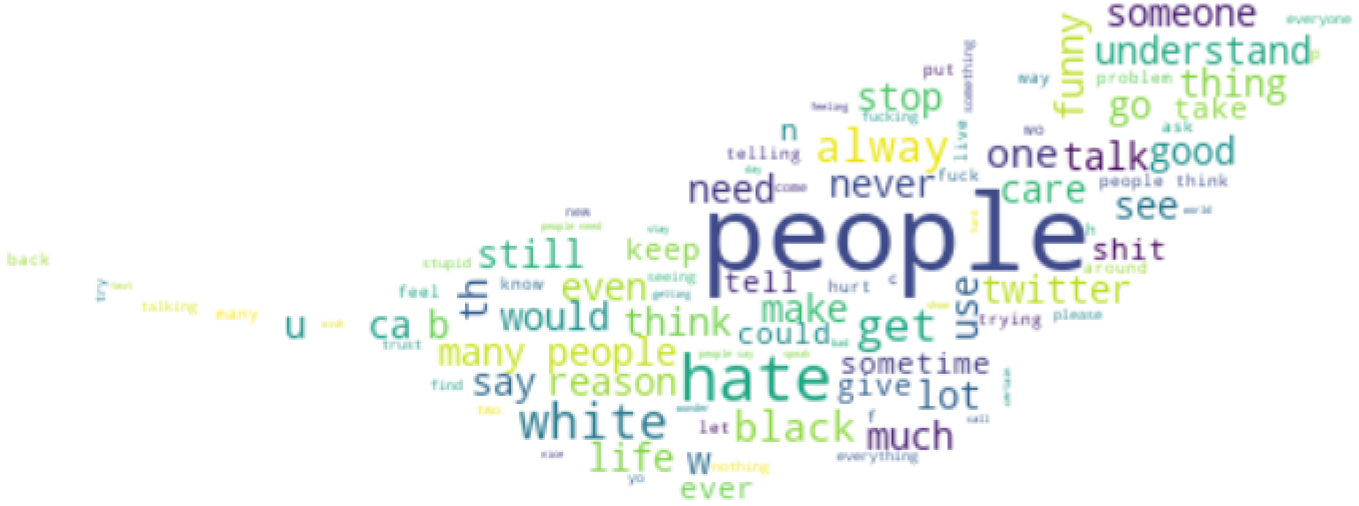
In [47]:

```
#Import libraries
import random
from wordcloud import WordCloud
from PIL import Image

#Create a random list to generate random 3 topics
randomlist = sorted(random.sample(range(0, num_topics), 3))

fig, axs = plt.subplots(3, figsize=(10,10))
mask = np.array(Image.open('1.png'))
```

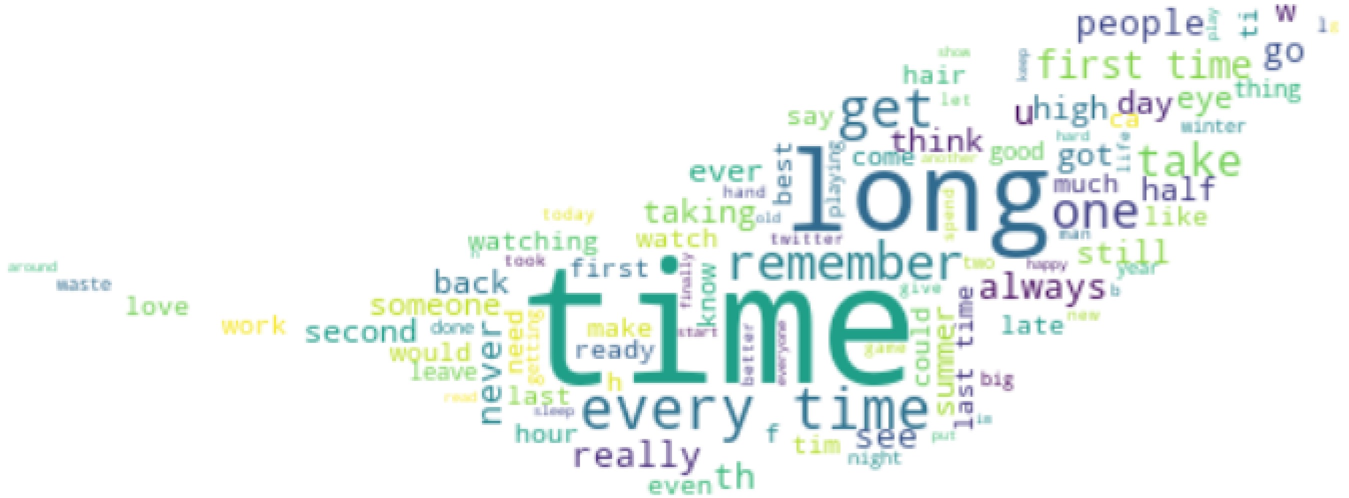
```python
k = 0
for i in range(len(axs)):
    wc = WordCloud(colormap='viridis_r', background_color='white', max_words=100,
                   stopwords=stopwords.words('english'), mask=mask)
    im = wc.generate(df_haiku[df_haiku['Topic']==randomlist[k]+1]['clean_text'].to_string())
    axs[i].imshow(im, interpolation='bilinear')
    axs[i].axis("off")
    axs[i].set_title('Topic ' + str(randomlist[k]+1), loc='left', fontsize=12)
    k += 1

plt.suptitle('Haiku Clustering WordCloud', size=15)

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

Haiku Clustering WordCloud

Topic 3



Topic 7



Topic 10

fucking
new
y
ta
go
go
ass
day
back
b
bitch
even
keep
got
ta
fuck
girl
money
getting
th
car
f
hit
baby
say
w
everybody
n
u
know
would
need
everyone
man
give
every
cut
shit
could
today
got
damn
thing
ai
one
sometime
still
much
never
finally
get
bitche
take
come
put
always
work
ca
see
something
ya
g

V: Is it possible to write Haiku like a poet? —*Generate Haiku using LSTM*

Import Libraries

```
In [48]: import tensorflow as tf
         from keras import Input, Model
         from keras.activations import softmax
         from tensorflow.keras.models import Sequential
         from keras.layers import Embedding, LSTM, Dense
         from tensorflow.keras import preprocessing , utils
         from tensorflow.keras.preprocessing.text import Tokenizer
         from tensorflow.keras.preprocessing.sequence import pad_sequences
```

Data Preprocessing

Due to the huge size of the dataset, it is difficult to train the model using my personal computer. Thus, I decided to check the data again to see if I can delete some haiku to make the data smaller.

Remove some source

In [49]:

```
df_haiku['source'].value_counts()
```

Out[49]:

```
source
twaiku         111727
img2poems       13538
sballas          8130
gutenberg        5494
tempslibres      5233
Name: count, dtype: int64
```

Column 'twaiku' has the most haiku. Unfortunately, the data is too huge for my model. I will delete this source and keep other sources for training later.

In [50]:

```
df_train = df_haiku.drop(df_haiku[df_haiku['source']=='twaiku'].index)
df_train['source'].value_counts()
```

Out[50]:

```
source
img2poems      13538
sballas         8130
gutenberg       5494
tempslibres     5233
Name: count, dtype: int64
```

The data size is smaller. Now I want to see the distribution of the haiku length to determine which length is ideal.

Calculate number of words in each haiku

In [51]:

```
df_train['length'] = df_train['clean_text'].apply(lambda x: len(x.split()))
```

In [52]:

```
df_train.head()
```

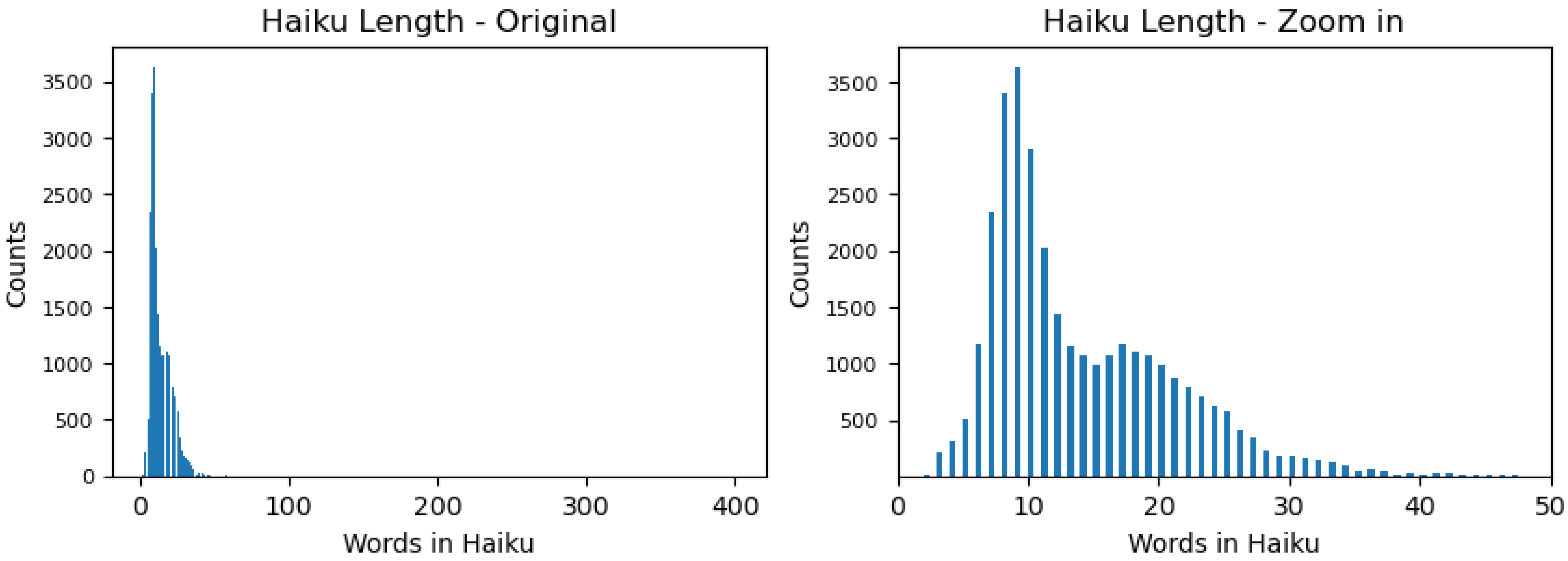
Out[52]:

	source	0	1	2	full	clean_text	Topic	length
0	tempslibres	fishing boats	colors of	the rainbow	fishing boats colors of the rainbow	fishing boats colors of the rainbow	11	6
1	tempslibres	ash wednesday--	trying to remember	my dream	ash wednesday-- trying to remember my dream	ash wednesday trying to remember my dream	15	7
2	tempslibres	snowy morn--	pouring another cup	of black coffee	snowy morn-- pouring another cup of black coffee	snowy morn pouring another cup of black coffee	11	8
3	tempslibres	shortest day	flames dance	in the oven	shortest day flames dance in the oven	shortest day flames dance in the oven	11	7
4	tempslibres	haze	half the horse hidden	behind the house	haze half the horse hidden behind the house	haze half the horse hidden behind the house	7	8

Show distribution of haiku length

In [53]:

```
fig, axes = plt.subplots(ncols=2, figsize=(10,3))

axes[0].hist(df_train['length'], bins=500)
axes[0].set_title('Haiku Length - Original')
axes[0].set_xlabel('Words in Haiku')
axes[0].set_ylabel('Counts')

axes[1].hist(df_train['length'], bins=800, bottom=1)
axes[1].set_xlim([0, 50])
axes[1].set_xlabel('Words in Haiku')
axes[1].set_ylabel('Counts')
axes[1].set_title('Haiku Length - Zoom in')

axes[0].yaxis.set_tick_params(labelsize=8)
axes[1].yaxis.set_tick_params(labelsize=8)

plt.show()
```

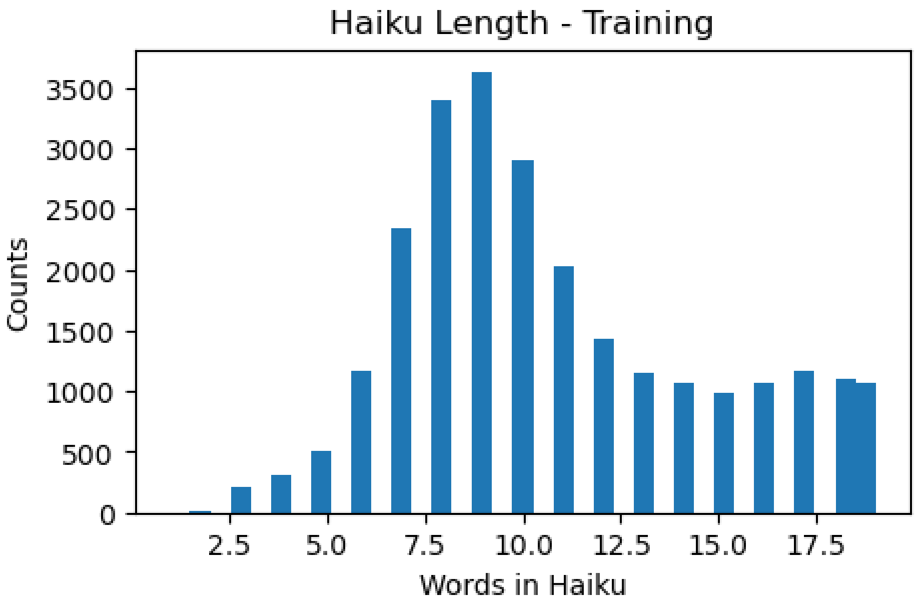


Haiku rules show us that there should be 17 (5-7-5 in each line) syllables in one haiku. If we have 17 words instead, it should be enough for the rule. However, based on the histogram above, I want to remove the length over 20 to keep the most of the data.

Drop haiku that is too long for training

In [54]:
```
#Drop haiku that have more than 25 words
df_train = df_train.drop(df_train[df_train['length'] >= 20].index)
```

In [55]:
```
plt.figure(figsize=(5,3))

plt.hist(df_train['length'], bins=35)
plt.title('Haiku Length - Training')
plt.xlabel('Words in Haiku')
plt.ylabel('Counts')

plt.show()
```



Now the data is better for training purpose.

Create a LSTM Model

```
In [56]: #Convert all haiku into a string
         text = '\n'.join(df_train['clean_text'].tolist())
```

```
In [57]: #Tokenize the text
         tokenizer = Tokenizer()
         tokenizer.fit_on_texts([text])
```

```
In [58]: #Calculate the vocabulary size
         VOCAB_SIZE = len(tokenizer.word_index) + 1
         print(VOCAB_SIZE)
```

```
25099
```

Create input sequences and corresponding labels

In [59]:

```
# Create input sequences and corresponding labels
input_sequences = []
for line in text.split('\n'):
    token_list = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i+1]
        input_sequences.append(n_gram_sequence)
```

In [60]:

```
input_sequences[:5]
```

Out[60]:

```
[[999, 1641],
 [999, 1641, 618],
 [999, 1641, 618, 3],
 [999, 1641, 618, 3, 1],
 [999, 1641, 618, 3, 1, 735]]
```

Calculate max sequence length for padding

In [61]:

```
max_sequence_length = max([len(x) for x in input_sequences])
print(max_sequence_length)
```

```
19
```

Pad the sequences and separate the sequences into X and y

In [62]:

```
input_sequences = pad_sequences(input_sequences, maxlen=max_sequence_length, padding='pre')
X, y = input_sequences[:,:-1],input_sequences[:,-1]
```

In [63]:

```
y = tf.keras.utils.to_categorical(y, num_classes=VOCAB_SIZE)
```

Build the LSTM model

In [64]:

```
# Build the LSTM model
#Initiate a Sequential model
model = Sequential()

#Add embedding layer with VOCAB_SIZE, 240 word embedding dimention, and the max length of the sequence(excluded the las
model.add(Embedding(VOCAB_SIZE, 240, input_length=max_sequence_length-1))
```

```
#Add LSTM layers
model.add(LSTM(150, return_sequences = True))
model.add(LSTM(100))

#Add Dense layer using relu
model.add(Dense(150, activation = 'relu'))

#Add the last layer with softmax and the number of units is the VOCAB_SIZE
model.add(Dense(VOCAB_SIZE, activation='softmax'))

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

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 18, 240)           6023760

 lstm (LSTM)                 (None, 18, 150)           234600

 lstm_1 (LSTM)               (None, 100)               100400

 dense (Dense)               (None, 150)               15150

 dense_1 (Dense)             (None, 25099)             3789949

=================================================================
Total params: 10,163,859
Trainable params: 10,163,859
Non-trainable params: 0
_________________________________________________________________
```

In [65]:

```
#Create a data generator to feed data with batches. It can avoid out-of-memory issue when training
from tensorflow.keras.utils import Sequence
class DataGenerator(Sequence):
    def __init__(self, x_set, y_set, batch_size):
        self.x, self.y = x_set, y_set
        self.batch_size = batch_size

    def __len__(self):
        return int(np.ceil(len(self.x) / float(self.batch_size)))
    def __getitem__(self, idx):
        batch_x = self.x[idx * self.batch_size:(idx + 1) * self.batch_size]
        batch_y = self.y[idx * self.batch_size:(idx + 1) * self.batch_size]
        return batch_x, batch_y
```

In [66]:

```
data = DataGenerator(X,y, batch_size=64)
```

In [67]:

```
model.fit(data, epochs=500, verbose=1)
model.save('lstm_model.keras')
```

```
Epoch 1/500
3946/3946 [==============================] - 53s 13ms/step - loss: 7.2900 - accuracy: 0.0827
Epoch 2/500
3946/3946 [==============================] - 50s 13ms/step - loss: 6.7316 - accuracy: 0.0999
Epoch 3/500
3946/3946 [==============================] - 51s 13ms/step - loss: 6.4274 - accuracy: 0.1106
Epoch 4/500
3946/3946 [==============================] - 50s 13ms/step - loss: 6.1857 - accuracy: 0.1188
Epoch 5/500
3946/3946 [==============================] - 50s 13ms/step - loss: 5.9741 - accuracy: 0.1262
Epoch 6/500
3946/3946 [==============================] - 50s 13ms/step - loss: 5.7833 - accuracy: 0.1342
Epoch 7/500
3946/3946 [==============================] - 50s 13ms/step - loss: 5.6017 - accuracy: 0.1409
Epoch 8/500
3946/3946 [==============================] - 50s 13ms/step - loss: 5.4289 - accuracy: 0.1487
Epoch 9/500
3946/3946 [==============================] - 50s 13ms/step - loss: 5.2666 - accuracy: 0.1557
Epoch 10/500
3946/3946 [==============================] - 50s 13ms/step - loss: 5.1088 - accuracy: 0.1629
Epoch 11/500
3946/3946 [==============================] - 50s 13ms/step - loss: 4.9549 - accuracy: 0.1712
Epoch 12/500
3946/3946 [==============================] - 50s 13ms/step - loss: 4.8049 - accuracy: 0.1795
Epoch 13/500
3946/3946 [==============================] - 50s 13ms/step - loss: 4.6604 - accuracy: 0.1882
Epoch 14/500
3946/3946 [==============================] - 51s 13ms/step - loss: 4.5171 - accuracy: 0.1993
Epoch 15/500
3946/3946 [==============================] - 51s 13ms/step - loss: 4.3824 - accuracy: 0.2118
Epoch 16/500
3946/3946 [==============================] - 50s 13ms/step - loss: 4.2470 - accuracy: 0.2251
Epoch 17/500
3946/3946 [==============================] - 50s 13ms/step - loss: 4.1138 - accuracy: 0.2401
Epoch 18/500
3946/3946 [==============================] - 50s 13ms/step - loss: 3.9889 - accuracy: 0.2549
Epoch 19/500
3946/3946 [==============================] - 50s 13ms/step - loss: 3.8715 - accuracy: 0.2691
Epoch 20/500
3946/3946 [==============================] - 50s 13ms/step - loss: 3.7588 - accuracy: 0.2830
Epoch 21/500
3946/3946 [==============================] - 50s 13ms/step - loss: 3.6536 - accuracy: 0.2967
Epoch 22/500
3946/3946 [==============================] - 50s 13ms/step - loss: 3.5512 - accuracy: 0.3114
Epoch 23/500
```

```
3946/3946 [==============================] - 49s 13ms/step - loss: 3.4604 - accuracy: 0.3232
Epoch 24/500
3946/3946 [==============================] - 50s 13ms/step - loss: 3.3681 - accuracy: 0.3372
Epoch 25/500
3946/3946 [==============================] - 50s 13ms/step - loss: 3.2856 - accuracy: 0.3477
Epoch 26/500
3946/3946 [==============================] - 50s 13ms/step - loss: 3.2065 - accuracy: 0.3605
Epoch 27/500
3946/3946 [==============================] - 50s 13ms/step - loss: 3.1341 - accuracy: 0.3717
Epoch 28/500
3946/3946 [==============================] - 50s 13ms/step - loss: 3.0624 - accuracy: 0.3825
Epoch 29/500
3946/3946 [==============================] - 49s 13ms/step - loss: 2.9916 - accuracy: 0.3937
Epoch 30/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.9287 - accuracy: 0.4029
Epoch 31/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.8662 - accuracy: 0.4131
Epoch 32/500
3946/3946 [==============================] - 49s 12ms/step - loss: 2.8103 - accuracy: 0.4211
Epoch 33/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.7513 - accuracy: 0.4312
Epoch 34/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.7013 - accuracy: 0.4395
Epoch 35/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.6534 - accuracy: 0.4485
Epoch 36/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.6000 - accuracy: 0.4565
Epoch 37/500
3946/3946 [==============================] - 49s 12ms/step - loss: 2.5517 - accuracy: 0.4650
Epoch 38/500
3946/3946 [==============================] - 49s 12ms/step - loss: 2.5131 - accuracy: 0.4707
Epoch 39/500
3946/3946 [==============================] - 49s 12ms/step - loss: 2.4678 - accuracy: 0.4779
Epoch 40/500
3946/3946 [==============================] - 49s 13ms/step - loss: 2.4306 - accuracy: 0.4844
Epoch 41/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.3900 - accuracy: 0.4914
Epoch 42/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.3531 - accuracy: 0.4968
Epoch 43/500
3946/3946 [==============================] - 49s 13ms/step - loss: 2.3155 - accuracy: 0.5046
Epoch 44/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.2822 - accuracy: 0.5103
Epoch 45/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.2509 - accuracy: 0.5144
```

```
Epoch 46/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.2154 - accuracy: 0.5216
Epoch 47/500
3946/3946 [==============================] - 49s 13ms/step - loss: 2.1864 - accuracy: 0.5264
Epoch 48/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.1574 - accuracy: 0.5316
Epoch 49/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.1307 - accuracy: 0.5368
Epoch 50/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.1033 - accuracy: 0.5406
Epoch 51/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.0784 - accuracy: 0.5459
Epoch 52/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.0507 - accuracy: 0.5512
Epoch 53/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.0274 - accuracy: 0.5551
Epoch 54/500
3946/3946 [==============================] - 50s 13ms/step - loss: 2.0049 - accuracy: 0.5589
Epoch 55/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.9798 - accuracy: 0.5634
Epoch 56/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.9640 - accuracy: 0.5670
Epoch 57/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.9378 - accuracy: 0.5719
Epoch 58/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.9166 - accuracy: 0.5756
Epoch 59/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.9023 - accuracy: 0.5782
Epoch 60/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.8779 - accuracy: 0.5827
Epoch 61/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.8612 - accuracy: 0.5858
Epoch 62/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.8482 - accuracy: 0.5876
Epoch 63/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.8265 - accuracy: 0.5918
Epoch 64/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.8114 - accuracy: 0.5946
Epoch 65/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.7940 - accuracy: 0.5977
Epoch 66/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.7794 - accuracy: 0.6005
Epoch 67/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.7727 - accuracy: 0.6019
Epoch 68/500
```

```
3946/3946 [==============================] - 50s 13ms/step - loss: 1.7500 - accuracy: 0.6060
Epoch 69/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.7391 - accuracy: 0.6085
Epoch 70/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.7303 - accuracy: 0.6097
Epoch 71/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.7103 - accuracy: 0.6144
Epoch 72/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.7037 - accuracy: 0.6146
Epoch 73/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.6897 - accuracy: 0.6178
Epoch 74/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.6806 - accuracy: 0.6200
Epoch 75/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.6636 - accuracy: 0.6222
Epoch 76/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.6518 - accuracy: 0.6246
Epoch 77/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.6473 - accuracy: 0.6252
Epoch 78/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.6309 - accuracy: 0.6288
Epoch 79/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.6241 - accuracy: 0.6298
Epoch 80/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.6140 - accuracy: 0.6317
Epoch 81/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.6044 - accuracy: 0.6336
Epoch 82/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.5982 - accuracy: 0.6357
Epoch 83/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.5847 - accuracy: 0.6382
Epoch 84/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.5837 - accuracy: 0.6387
Epoch 85/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.5621 - accuracy: 0.6418
Epoch 86/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.5626 - accuracy: 0.6411
Epoch 87/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.5552 - accuracy: 0.6446
Epoch 88/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.5439 - accuracy: 0.6450
Epoch 89/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.5337 - accuracy: 0.6468
Epoch 90/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.5284 - accuracy: 0.6495
```

```
Epoch 91/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.5196 - accuracy: 0.6509
Epoch 92/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.5117 - accuracy: 0.6516
Epoch 93/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.5091 - accuracy: 0.6528
Epoch 94/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4960 - accuracy: 0.6550
Epoch 95/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4941 - accuracy: 0.6556
Epoch 96/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4888 - accuracy: 0.6562
Epoch 97/500
3946/3946 [==============================] - 52s 13ms/step - loss: 1.4830 - accuracy: 0.6575
Epoch 98/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4751 - accuracy: 0.6597
Epoch 99/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4640 - accuracy: 0.6614
Epoch 100/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4651 - accuracy: 0.6605
Epoch 101/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.4545 - accuracy: 0.6638
Epoch 102/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4524 - accuracy: 0.6643
Epoch 103/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4431 - accuracy: 0.6654
Epoch 104/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4358 - accuracy: 0.6677
Epoch 105/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4323 - accuracy: 0.6676
Epoch 106/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4217 - accuracy: 0.6697
Epoch 107/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4233 - accuracy: 0.6697
Epoch 108/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4167 - accuracy: 0.6711
Epoch 109/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4097 - accuracy: 0.6721
Epoch 110/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4007 - accuracy: 0.6743
Epoch 111/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.4019 - accuracy: 0.6732
Epoch 112/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3943 - accuracy: 0.6746
Epoch 113/500
```

```
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3897 - accuracy: 0.6770
Epoch 114/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3851 - accuracy: 0.6763
Epoch 115/500
3946/3946 [==============================] - 49s 12ms/step - loss: 1.3777 - accuracy: 0.6781
Epoch 116/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3745 - accuracy: 0.6798
Epoch 117/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3687 - accuracy: 0.6809
Epoch 118/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3684 - accuracy: 0.6803
Epoch 119/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3629 - accuracy: 0.6815
Epoch 120/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3657 - accuracy: 0.6806
Epoch 121/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3542 - accuracy: 0.6824
Epoch 122/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3476 - accuracy: 0.6847
Epoch 123/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3437 - accuracy: 0.6863
Epoch 124/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3417 - accuracy: 0.6852
Epoch 125/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3427 - accuracy: 0.6858
Epoch 126/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3292 - accuracy: 0.6883
Epoch 127/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3300 - accuracy: 0.6880
Epoch 128/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3308 - accuracy: 0.6876
Epoch 129/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3213 - accuracy: 0.6895
Epoch 130/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3261 - accuracy: 0.6897
Epoch 131/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3175 - accuracy: 0.6908
Epoch 132/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3173 - accuracy: 0.6911
Epoch 133/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3111 - accuracy: 0.6921
Epoch 134/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3097 - accuracy: 0.6923
Epoch 135/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3028 - accuracy: 0.6931
```

```
Epoch 136/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3018 - accuracy: 0.6935
Epoch 137/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.3039 - accuracy: 0.6936
Epoch 138/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2959 - accuracy: 0.6951
Epoch 139/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2919 - accuracy: 0.6952
Epoch 140/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2907 - accuracy: 0.6971
Epoch 141/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2896 - accuracy: 0.6964
Epoch 142/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2854 - accuracy: 0.6966
Epoch 143/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2782 - accuracy: 0.6985
Epoch 144/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.2794 - accuracy: 0.6984
Epoch 145/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2829 - accuracy: 0.6976
Epoch 146/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2786 - accuracy: 0.6990
Epoch 147/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2723 - accuracy: 0.6999
Epoch 148/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2582 - accuracy: 0.7034
Epoch 149/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2652 - accuracy: 0.7019
Epoch 150/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2680 - accuracy: 0.7006
Epoch 151/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2657 - accuracy: 0.7016
Epoch 152/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2565 - accuracy: 0.7033
Epoch 153/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2568 - accuracy: 0.7034
Epoch 154/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2532 - accuracy: 0.7038
Epoch 155/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2518 - accuracy: 0.7042
Epoch 156/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2537 - accuracy: 0.7049
Epoch 157/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2549 - accuracy: 0.7036
Epoch 158/500
```

```
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2492 - accuracy: 0.7042
Epoch 159/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2421 - accuracy: 0.7064
Epoch 160/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2467 - accuracy: 0.7055
Epoch 161/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2365 - accuracy: 0.7068
Epoch 162/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2386 - accuracy: 0.7065
Epoch 163/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2336 - accuracy: 0.7077
Epoch 164/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2141 - accuracy: 0.7128
Epoch 165/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.2335 - accuracy: 0.7093
Epoch 166/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2369 - accuracy: 0.7075
Epoch 167/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2236 - accuracy: 0.7103
Epoch 168/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2269 - accuracy: 0.7100
Epoch 169/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2281 - accuracy: 0.7093
Epoch 170/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2267 - accuracy: 0.7091
Epoch 171/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2182 - accuracy: 0.7107
Epoch 172/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2234 - accuracy: 0.7096
Epoch 173/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2203 - accuracy: 0.7109
Epoch 174/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2121 - accuracy: 0.7123
Epoch 175/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2137 - accuracy: 0.7124
Epoch 176/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2133 - accuracy: 0.7119
Epoch 177/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2134 - accuracy: 0.7119
Epoch 178/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2086 - accuracy: 0.7136
Epoch 179/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.2103 - accuracy: 0.7131
Epoch 180/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2014 - accuracy: 0.7148
```

```
Epoch 181/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2055 - accuracy: 0.7136
Epoch 182/500
3946/3946 [==============================] - 49s 12ms/step - loss: 1.2009 - accuracy: 0.7148
Epoch 183/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1998 - accuracy: 0.7156
Epoch 184/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.2036 - accuracy: 0.7139
Epoch 185/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1971 - accuracy: 0.7149
Epoch 186/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.2003 - accuracy: 0.7145
Epoch 187/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1958 - accuracy: 0.7152
Epoch 188/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1883 - accuracy: 0.7172
Epoch 189/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1870 - accuracy: 0.7174
Epoch 190/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1852 - accuracy: 0.7178
Epoch 191/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1877 - accuracy: 0.7168
Epoch 192/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.1819 - accuracy: 0.7188
Epoch 193/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1893 - accuracy: 0.7172
Epoch 194/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1876 - accuracy: 0.7182
Epoch 195/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1882 - accuracy: 0.7176
Epoch 196/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1804 - accuracy: 0.7199
Epoch 197/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1790 - accuracy: 0.7184
Epoch 198/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1751 - accuracy: 0.7203
Epoch 199/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1784 - accuracy: 0.7196
Epoch 200/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1737 - accuracy: 0.7208
Epoch 201/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1819 - accuracy: 0.7187
Epoch 202/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1712 - accuracy: 0.7211
Epoch 203/500
```

```
3946/3946 [==============================] - 49s 13ms/step - loss: 1.1716 - accuracy: 0.7205
Epoch 204/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1640 - accuracy: 0.7229
Epoch 205/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1685 - accuracy: 0.7215
Epoch 206/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1741 - accuracy: 0.7197
Epoch 207/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1695 - accuracy: 0.7209
Epoch 208/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1571 - accuracy: 0.7239
Epoch 209/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1594 - accuracy: 0.7227
Epoch 210/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1634 - accuracy: 0.7224
Epoch 211/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1681 - accuracy: 0.7226
Epoch 212/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1587 - accuracy: 0.7233
Epoch 213/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1601 - accuracy: 0.7236
Epoch 214/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1612 - accuracy: 0.7233
Epoch 215/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1658 - accuracy: 0.7231
Epoch 216/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1484 - accuracy: 0.7262
Epoch 217/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1557 - accuracy: 0.7242
Epoch 218/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1531 - accuracy: 0.7252
Epoch 219/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1500 - accuracy: 0.7251
Epoch 220/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1515 - accuracy: 0.7253
Epoch 221/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1530 - accuracy: 0.7250
Epoch 222/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1470 - accuracy: 0.7253
Epoch 223/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1436 - accuracy: 0.7262
Epoch 224/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1472 - accuracy: 0.7254
Epoch 225/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1455 - accuracy: 0.7258
```

```
Epoch 226/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1418 - accuracy: 0.7267
Epoch 227/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1472 - accuracy: 0.7260
Epoch 228/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1464 - accuracy: 0.7254
Epoch 229/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1426 - accuracy: 0.7277
Epoch 230/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1433 - accuracy: 0.7273
Epoch 231/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1427 - accuracy: 0.7269
Epoch 232/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1341 - accuracy: 0.7287
Epoch 233/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1370 - accuracy: 0.7270
Epoch 234/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1404 - accuracy: 0.7274
Epoch 235/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.1348 - accuracy: 0.7279
Epoch 236/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1325 - accuracy: 0.7291
Epoch 237/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1347 - accuracy: 0.7286
Epoch 238/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1391 - accuracy: 0.7282
Epoch 239/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1298 - accuracy: 0.7298
Epoch 240/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1361 - accuracy: 0.7288
Epoch 241/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1350 - accuracy: 0.7290
Epoch 242/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1333 - accuracy: 0.7282
Epoch 243/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1294 - accuracy: 0.7290
Epoch 244/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1262 - accuracy: 0.7302
Epoch 245/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1289 - accuracy: 0.7294
Epoch 246/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1194 - accuracy: 0.7316
Epoch 247/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1308 - accuracy: 0.7304
Epoch 248/500
```

```
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1223 - accuracy: 0.7311
Epoch 249/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.1321 - accuracy: 0.7286
Epoch 250/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1206 - accuracy: 0.7321
Epoch 251/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1229 - accuracy: 0.7317
Epoch 252/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1227 - accuracy: 0.7307
Epoch 253/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1190 - accuracy: 0.7319
Epoch 254/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.1237 - accuracy: 0.7305
Epoch 255/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.1212 - accuracy: 0.7310
Epoch 256/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.1170 - accuracy: 0.7314
Epoch 257/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1225 - accuracy: 0.7317
Epoch 258/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1108 - accuracy: 0.7340
Epoch 259/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1158 - accuracy: 0.7326
Epoch 260/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1110 - accuracy: 0.7335
Epoch 261/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.1112 - accuracy: 0.7325
Epoch 262/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.1228 - accuracy: 0.7312
Epoch 263/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1166 - accuracy: 0.7319
Epoch 264/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1172 - accuracy: 0.7321
Epoch 265/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1154 - accuracy: 0.7327
Epoch 266/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1106 - accuracy: 0.7342
Epoch 267/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1132 - accuracy: 0.7331
Epoch 268/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1177 - accuracy: 0.7320
Epoch 269/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1180 - accuracy: 0.7323
Epoch 270/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1127 - accuracy: 0.7325
```

```
Epoch 271/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.1057 - accuracy: 0.7353
Epoch 272/500
3946/3946 [==============================] - 49s 12ms/step - loss: 1.1143 - accuracy: 0.7325
Epoch 273/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1117 - accuracy: 0.7341
Epoch 274/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1057 - accuracy: 0.7341
Epoch 275/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1178 - accuracy: 0.7314
Epoch 276/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.1116 - accuracy: 0.7338
Epoch 277/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.1044 - accuracy: 0.7351
Epoch 278/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1041 - accuracy: 0.7356
Epoch 279/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1067 - accuracy: 0.7339
Epoch 280/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.1052 - accuracy: 0.7346
Epoch 281/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1128 - accuracy: 0.7330
Epoch 282/500
3946/3946 [==============================] - 49s 12ms/step - loss: 1.1054 - accuracy: 0.7347
Epoch 283/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1104 - accuracy: 0.7336
Epoch 284/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1033 - accuracy: 0.7348
Epoch 285/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1063 - accuracy: 0.7338
Epoch 286/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1116 - accuracy: 0.7335
Epoch 287/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.1033 - accuracy: 0.7344
Epoch 288/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1064 - accuracy: 0.7338
Epoch 289/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1006 - accuracy: 0.7360
Epoch 290/500
3946/3946 [==============================] - 49s 12ms/step - loss: 1.0970 - accuracy: 0.7373
Epoch 291/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0969 - accuracy: 0.7362
Epoch 292/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1066 - accuracy: 0.7348
Epoch 293/500
```

```
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0961 - accuracy: 0.7362
Epoch 294/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1049 - accuracy: 0.7352
Epoch 295/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0856 - accuracy: 0.7391
Epoch 296/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0972 - accuracy: 0.7358
Epoch 297/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0945 - accuracy: 0.7368
Epoch 298/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1018 - accuracy: 0.7356
Epoch 299/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0945 - accuracy: 0.7366
Epoch 300/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.0972 - accuracy: 0.7357
Epoch 301/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.1080 - accuracy: 0.7339
Epoch 302/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0960 - accuracy: 0.7364
Epoch 303/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1023 - accuracy: 0.7355
Epoch 304/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1034 - accuracy: 0.7355
Epoch 305/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0927 - accuracy: 0.7365
Epoch 306/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0959 - accuracy: 0.7370
Epoch 307/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0984 - accuracy: 0.7351
Epoch 308/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.1005 - accuracy: 0.7356
Epoch 309/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0968 - accuracy: 0.7365
Epoch 310/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0898 - accuracy: 0.7381
Epoch 311/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0956 - accuracy: 0.7368
Epoch 312/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.1015 - accuracy: 0.7361
Epoch 313/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.0921 - accuracy: 0.7369
Epoch 314/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0940 - accuracy: 0.7378
Epoch 315/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0876 - accuracy: 0.7381
```

```
Epoch 316/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0928 - accuracy: 0.7377
Epoch 317/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0883 - accuracy: 0.7384
Epoch 318/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0935 - accuracy: 0.7377
Epoch 319/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0911 - accuracy: 0.7384
Epoch 320/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0876 - accuracy: 0.7380
Epoch 321/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0967 - accuracy: 0.7366
Epoch 322/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0821 - accuracy: 0.7392
Epoch 323/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0881 - accuracy: 0.7387
Epoch 324/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0857 - accuracy: 0.7386
Epoch 325/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0896 - accuracy: 0.7384
Epoch 326/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0900 - accuracy: 0.7380
Epoch 327/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0839 - accuracy: 0.7380
Epoch 328/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0845 - accuracy: 0.7389
Epoch 329/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0890 - accuracy: 0.7369
Epoch 330/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0841 - accuracy: 0.7395
Epoch 331/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0907 - accuracy: 0.7382
Epoch 332/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0885 - accuracy: 0.7388
Epoch 333/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0870 - accuracy: 0.7392
Epoch 334/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0830 - accuracy: 0.7401
Epoch 335/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0942 - accuracy: 0.7369
Epoch 336/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0867 - accuracy: 0.7394
Epoch 337/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0815 - accuracy: 0.7392
Epoch 338/500
```

```
3946/3946 [==============================] - 49s 13ms/step - loss: 1.0918 - accuracy: 0.7384
Epoch 339/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.0861 - accuracy: 0.7379
Epoch 340/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0833 - accuracy: 0.7395
Epoch 341/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0861 - accuracy: 0.7383
Epoch 342/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0808 - accuracy: 0.7403
Epoch 343/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0924 - accuracy: 0.7367
Epoch 344/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0736 - accuracy: 0.7415
Epoch 345/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.0800 - accuracy: 0.7401
Epoch 346/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0828 - accuracy: 0.7397
Epoch 347/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0821 - accuracy: 0.7394
Epoch 348/500
3946/3946 [==============================] - 49s 12ms/step - loss: 1.0848 - accuracy: 0.7393
Epoch 349/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.0810 - accuracy: 0.7395
Epoch 350/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0826 - accuracy: 0.7396
Epoch 351/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0831 - accuracy: 0.7390
Epoch 352/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0857 - accuracy: 0.7388
Epoch 353/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0889 - accuracy: 0.7385
Epoch 354/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0780 - accuracy: 0.7400
Epoch 355/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0841 - accuracy: 0.7389
Epoch 356/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0888 - accuracy: 0.7390
Epoch 357/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0835 - accuracy: 0.7384
Epoch 358/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0817 - accuracy: 0.7395
Epoch 359/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0765 - accuracy: 0.7406
Epoch 360/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0809 - accuracy: 0.7397
```

```
Epoch 361/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0811 - accuracy: 0.7403
Epoch 362/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0797 - accuracy: 0.7395
Epoch 363/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0730 - accuracy: 0.7417
Epoch 364/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0777 - accuracy: 0.7399
Epoch 365/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0864 - accuracy: 0.7387
Epoch 366/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0821 - accuracy: 0.7398
Epoch 367/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0797 - accuracy: 0.7408
Epoch 368/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0799 - accuracy: 0.7406
Epoch 369/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0803 - accuracy: 0.7408
Epoch 370/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0806 - accuracy: 0.7409
Epoch 371/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0719 - accuracy: 0.7413
Epoch 372/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0841 - accuracy: 0.7388
Epoch 373/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0798 - accuracy: 0.7404
Epoch 374/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0829 - accuracy: 0.7399
Epoch 375/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0691 - accuracy: 0.7429
Epoch 376/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0734 - accuracy: 0.7409
Epoch 377/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0800 - accuracy: 0.7404
Epoch 378/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0754 - accuracy: 0.7409
Epoch 379/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0880 - accuracy: 0.7385
Epoch 380/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0828 - accuracy: 0.7393
Epoch 381/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0784 - accuracy: 0.7410
Epoch 382/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0767 - accuracy: 0.7407
Epoch 383/500
```

```
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0828 - accuracy: 0.7398
Epoch 384/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0786 - accuracy: 0.7412
Epoch 385/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0738 - accuracy: 0.7408
Epoch 386/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0754 - accuracy: 0.7408
Epoch 387/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0866 - accuracy: 0.7392
Epoch 388/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0746 - accuracy: 0.7410
Epoch 389/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0748 - accuracy: 0.7423
Epoch 390/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0782 - accuracy: 0.7408
Epoch 391/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0756 - accuracy: 0.7414
Epoch 392/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0819 - accuracy: 0.7397
Epoch 393/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0703 - accuracy: 0.7423
Epoch 394/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0758 - accuracy: 0.7405
Epoch 395/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0737 - accuracy: 0.7420
Epoch 396/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0770 - accuracy: 0.7407
Epoch 397/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0826 - accuracy: 0.7394
Epoch 398/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0761 - accuracy: 0.7414
Epoch 399/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0777 - accuracy: 0.7407
Epoch 400/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0712 - accuracy: 0.7417
Epoch 401/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0810 - accuracy: 0.7395
Epoch 402/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0785 - accuracy: 0.7411
Epoch 403/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0787 - accuracy: 0.7411
Epoch 404/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0772 - accuracy: 0.7408
Epoch 405/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0770 - accuracy: 0.7412
```

```
Epoch 406/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0769 - accuracy: 0.7419
Epoch 407/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0736 - accuracy: 0.7415
Epoch 408/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0789 - accuracy: 0.7401
Epoch 409/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0727 - accuracy: 0.7411
Epoch 410/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0751 - accuracy: 0.7415
Epoch 411/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0647 - accuracy: 0.7437
Epoch 412/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0733 - accuracy: 0.7416
Epoch 413/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0778 - accuracy: 0.7401
Epoch 414/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0748 - accuracy: 0.7418
Epoch 415/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0714 - accuracy: 0.7417
Epoch 416/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0746 - accuracy: 0.7415
Epoch 417/500
3946/3946 [==============================] - 49s 12ms/step - loss: 1.0783 - accuracy: 0.7415
Epoch 418/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0828 - accuracy: 0.7398
Epoch 419/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0795 - accuracy: 0.7404
Epoch 420/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0758 - accuracy: 0.7421
Epoch 421/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0768 - accuracy: 0.7408
Epoch 422/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.0712 - accuracy: 0.7428
Epoch 423/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0750 - accuracy: 0.7407
Epoch 424/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0673 - accuracy: 0.7432
Epoch 425/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0796 - accuracy: 0.7408
Epoch 426/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0748 - accuracy: 0.7413
Epoch 427/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0802 - accuracy: 0.7407
Epoch 428/500
```

```
3946/3946 [==============================] - 49s 13ms/step - loss: 1.0710 - accuracy: 0.7421
Epoch 429/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0756 - accuracy: 0.7412
Epoch 430/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0757 - accuracy: 0.7410
Epoch 431/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0764 - accuracy: 0.7408
Epoch 432/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0728 - accuracy: 0.7415
Epoch 433/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0786 - accuracy: 0.7401
Epoch 434/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0783 - accuracy: 0.7409
Epoch 435/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0706 - accuracy: 0.7427
Epoch 436/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0811 - accuracy: 0.7411
Epoch 437/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0814 - accuracy: 0.7400
Epoch 438/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0733 - accuracy: 0.7410
Epoch 439/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0737 - accuracy: 0.7416
Epoch 440/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0728 - accuracy: 0.7413
Epoch 441/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0753 - accuracy: 0.7409
Epoch 442/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0895 - accuracy: 0.7386
Epoch 443/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0687 - accuracy: 0.7417
Epoch 444/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0761 - accuracy: 0.7407
Epoch 445/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0698 - accuracy: 0.7422
Epoch 446/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.0822 - accuracy: 0.7404
Epoch 447/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0823 - accuracy: 0.7399
Epoch 448/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0766 - accuracy: 0.7413
Epoch 449/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0774 - accuracy: 0.7406
Epoch 450/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0743 - accuracy: 0.7410
```

```
Epoch 451/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0798 - accuracy: 0.7407
Epoch 452/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.0731 - accuracy: 0.7410
Epoch 453/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0760 - accuracy: 0.7401
Epoch 454/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0758 - accuracy: 0.7412
Epoch 455/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0745 - accuracy: 0.7425
Epoch 456/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0649 - accuracy: 0.7433
Epoch 457/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0711 - accuracy: 0.7424
Epoch 458/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0754 - accuracy: 0.7411
Epoch 459/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0822 - accuracy: 0.7393
Epoch 460/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0781 - accuracy: 0.7407
Epoch 461/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0830 - accuracy: 0.7397
Epoch 462/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0736 - accuracy: 0.7415
Epoch 463/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0822 - accuracy: 0.7399
Epoch 464/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0787 - accuracy: 0.7401
Epoch 465/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0772 - accuracy: 0.7407
Epoch 466/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0767 - accuracy: 0.7405
Epoch 467/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0671 - accuracy: 0.7425
Epoch 468/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0777 - accuracy: 0.7406
Epoch 469/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0796 - accuracy: 0.7398
Epoch 470/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0840 - accuracy: 0.7391
Epoch 471/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0812 - accuracy: 0.7393
Epoch 472/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0802 - accuracy: 0.7404
Epoch 473/500
```

```
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0680 - accuracy: 0.7427
Epoch 474/500
3946/3946 [==============================] - 49s 12ms/step - loss: 1.0710 - accuracy: 0.7414
Epoch 475/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0765 - accuracy: 0.7408
Epoch 476/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0847 - accuracy: 0.7391
Epoch 477/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0907 - accuracy: 0.7385
Epoch 478/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0737 - accuracy: 0.7417
Epoch 479/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0750 - accuracy: 0.7417
Epoch 480/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0746 - accuracy: 0.7417
Epoch 481/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0750 - accuracy: 0.7414
Epoch 482/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0843 - accuracy: 0.7400
Epoch 483/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0737 - accuracy: 0.7409
Epoch 484/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0758 - accuracy: 0.7416
Epoch 485/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0708 - accuracy: 0.7420
Epoch 486/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0788 - accuracy: 0.7411
Epoch 487/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0824 - accuracy: 0.7404
Epoch 488/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0845 - accuracy: 0.7396
Epoch 489/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0762 - accuracy: 0.7411
Epoch 490/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0704 - accuracy: 0.7420
Epoch 491/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0678 - accuracy: 0.7426
Epoch 492/500
3946/3946 [==============================] - 49s 13ms/step - loss: 1.0812 - accuracy: 0.7391
Epoch 493/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0782 - accuracy: 0.7402
Epoch 494/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0734 - accuracy: 0.7414
Epoch 495/500
3946/3946 [==============================] - 51s 13ms/step - loss: 1.0785 - accuracy: 0.7400
```

```
Epoch 496/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0778 - accuracy: 0.7407
Epoch 497/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0776 - accuracy: 0.7409
Epoch 498/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0804 - accuracy: 0.7400
Epoch 499/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0788 - accuracy: 0.7395
Epoch 500/500
3946/3946 [==============================] - 50s 13ms/step - loss: 1.0711 - accuracy: 0.7416
```

Create a function to generate haiku using the LSTM model

Since the model generates text word by word instead of line by line, I need a format to make the generated text looks like a haiku. It is impossible to get 5-7-5 syllable pattern because I didn't train the model on character level, so the best option for me is to create a 5-7-5 word pattern instead.

In [68]:

```
def generate_haiku(seed_text):
    #Put seed_text into 3 lines with 5-7-5 word rule.
    text = seed_text.split()
    line_1, line_2, line_3 = ([] for i in range(3))
    if len(text)> 12:
        line_3 = text[13:]
        line_2 = text[6:13]
        line_1 = text[:5]
    elif len(text)>5:
        line_2 = text[6:]
        line_1 = text[:5]
    else:
        line_1 = text

    #Generate the rest of the haiku words from the seed_text
    #Keep the first line with 5 words, the second line with 7 words, and the third line with 5 words
    for i in range(1,4):
        if i==1:
            for _ in range(5-len(line_1)):
                token_list = tokenizer.texts_to_sequences([seed_text])[0] # tokenizing the seed text
                token_list = pad_sequences([token_list], maxlen=max_sequence_length-1, padding='pre') # padding
            #The model is used to predict the index of the next word in the sequence using the predict method.
                predicted = np.argmax(model.predict(token_list, verbose=0))

                predicted_word = ""
                for word, index in tokenizer.word_index.items():
                    if index == predicted:
                        predicted_word = word
                        break
                seed_text = seed_text + ' ' + predicted_word
                line_1.append(predicted_word)
            line_1_text = ' '.join(line_1)
        if i==2:
            for _ in range(7-len(line_2)):
                token_list = tokenizer.texts_to_sequences([seed_text])[0] # tokenizing the seed text
                token_list = pad_sequences([token_list], maxlen=max_sequence_length-1, padding='pre') # padding
            #The model is used to predict the index of the next word in the sequence using the predict method.
                predicted = np.argmax(model.predict(token_list, verbose=0))

                predicted_word = ""
                for word, index in tokenizer.word_index.items():
                    if index == predicted:
                        predicted_word = word
                        break
                seed_text = seed_text + ' ' + predicted_word
                line_2.append(predicted_word)
            line_2_text = ' '.join(line_2)

        if i==3:
            for _ in range(5-len(line_3)):
                token_list = tokenizer.texts_to_sequences([seed_text])[0] # tokenizing the seed text
                token_list = pad_sequences([token_list], maxlen=max_sequence_length-1, padding='pre') # padding
            #The model is used to predict the index of the next word in the sequence using the predict method.
                predicted = np.argmax(model.predict(token_list, verbose=0))

                predicted_word = ""
                for word, index in tokenizer.word_index.items():
                    if index == predicted:
                        predicted_word = word
                        break
                seed_text = seed_text + ' ' + predicted_word
                line_3.append(predicted_word)
            line_3_text = ' '.join(line_3)
    haiku = (line_1_text +'\n'+ line_2_text +'\n'+ line_3_text)
    print(haiku)
```

Create a ineraction interface for users

In [69]:

```
def interaction():
    print("Please enter some beginning words (recommanded less than 5 words)for your haiku. \nEnter 'Bye' if you want t
    text = ''
    while True:
        text = input()
        if text =='Bye':
            print('Goodbye')
            break
        else:
            print('-'*50)
            haiku = generate_haiku(text)
            print('-'*50)
```

Test the haiku generator

In [76]:

```
interaction()
```

```
Please enter some beginning words (recommanded less than 5 words)for your haiku.
Enter 'Bye' if you want to quit:

Christmas
--------------------------------------------------
Christmas eve candles flicker through
our wine glasses cover the tips of
dry solace tempest milk on
--------------------------------------------------
Summer
--------------------------------------------------
Summer end the beach umbrella
man flat winter leaves doth finds his
tank tower all our horses
--------------------------------------------------
autumn leaves
--------------------------------------------------
autumn leaves the creak in
my left knee someone must afford shadows
make chaos fight away with
--------------------------------------------------
Bye
Goodbye
```

VI: Conclusion

For this project, I shared the world of haiku by showing the Wikipedia page with text smmarization, the translation of original Japanese haiku, EDA of haiku dataset and clustering, and finally I create a LSTM model to generate some haiku.

- It is difficult to use the full data (with 144,122 rows and more than 400 sequence length) to pad sequences and do the one-hot encoding with vocabulary size of more than 40,000. I have to shrink the size of the dataset in order to train the model properly using my personal computer.
- The limitation of my LSTM model is that it can only generate fixed length of words. To get a better form of haiku, I set each line of words with 5, 7, 5 individually. However, because the nature of poetry is to separate lines based on the meaning of the context, it is hard to make the generated haiku separated naturally. Although the generated haiku does't make sense, I can still feel some 'poetic vibes' among the words.
- If I want to get the 5-7-5 syllables instead of 5-7-5 words, I may need to tokenize the text on character level. However, there might be some unknown words created randomly.
- I actually did another seq2seq model but does not perform as good as LSTM. I didn't include the seq2seq model in the project due to time constraints, but I will try to reconstruct the model once I have time to see if it generates better haikus. After all, it can generate different lengths of words for each line, which is better than the fixed number of words.

In [ ]: