1. Abstract
2. Introduction
3. Research Problem and Statement of Objectives
   1. Problem Statement
   2. Problem Definition
   3. Research Objectives
   4. Delimitation
4. Literature Review
5. Lexical Study of Emojis
   1. Research Methodology
   2. Dataset Selection
      1. Basic Emotional Theory
      2. Dimensional Theory
   3. Definition of Sentiment Parameters
      1. Basic Emotional Theory
      2. Dimensional Theory
   4. Feature Extraction of Emoji
      1. Use of Emotion Vocabulary Embeddings
      2. Word Vectors of Basic Emotions
      3. Binary Word Association Lexicon
      4. Word-Emotion Intensity Lexicon
      5. Sentiment-Aware Vector Space Modification
   5. Emoji Sentiment Prediction
      1. Data preparation
      2. Model Development and Evaluation
   6. Results and Discussion
6. Pragmatics of Sarcasm
   1. Evaluation of Annotation Strategies
      1. Primary Research Methodology
      2. Survey Outcome
   2. Statistical Analysis of Pragmatics in Sarcastic Content
   3. Quality Assessment of Present Sarcasm-Annotated Datasets
   4. Results and Discussion
7. Sarcasm Detection Integrating Emoji Pragmatics
   1. Proposed Novel Methodology
      1. Sentiment-Aware Attention Mechanism
   2. Data Preparation
      1. Expansion of current dataset
      2. Data Cleaning
   3. Model Selection and Evaluation
   4. Results and Discussion

\*\*Contain discussion of overall results, compare with and w/o the attention mechanism and then look at the effects of emojis on the result

1. Evaluation
   1. Solution Strengths
   2. Limitations
   3. Applications
2. Conclusions
3. Appendix
4. References

|  |  |  |
| --- | --- | --- |
| Chapter | Status | Notes |
| 1 |  |  |
| 2 |  |  |
| 3 | Initial draft provisionally complete |  |
| 4 | Initial draft provisionally complete | Will be edited to match body of work-Likely after artifact is completed.  Per feedback-rework conclusion  Maybe also intro? |
| 5 | Code Complete  Initial draft provisionally complete |  |
| 6 | Code Complete  Initial draft provisionally complete |  |
| 7 | Idea for architectures identified.  Data collection and cleaning complete over weekend  Basic baseline models Mon/Tues tuned  Work on sentiment attention mechanism between Tues/Wed  Idea for addressing other observations wed/thurs  Implement Thurs/Fri  More optimisation Sat/Sun  Compare to alternative pre-trained models Mon  Write up- happening as the work is being completed |  |
| 8 |  |  |
| 9 |  |  |
| 10 | Write-up up to date with chapter write ups  Need to do section about licenses etc. |  |

#Tokenize the text

ss["text"] = ss["text"].apply(lambda x: word\_tokenize(x))

ss.head()

#Check there are no rows with no tokens- remove if there are

#Count the tokens per row and find rows with zero tokens

zero\_tokens = ss['text'].apply(len) == 0

#Filter to find any rows with no tokens

rows\_with\_zero\_tokens = ss[zero\_tokens]

# Check if there are any rows with zero tokens

if len(rows\_with\_zero\_tokens) > 0:

print("Rows with zero tokens:")

print(rows\_with\_zero\_tokens)

else:

print("No rows with zero tokens found.")

#Drop rows with zero tokens

ss = ss[~zero\_tokens]

ss = ss.reset\_index(drop=True)

#Start at a sequence length and adjust to find the minimum value

target\_sequence\_length = 60

while True:

#Count the tokens per row and find rows with more than the target sequence length

more\_than\_target\_tokens = ss['text'].apply(lambda x: len(x)) > target\_sequence\_length

#Filter to find rows with more than the target sequence length

rows\_with\_more\_than\_target\_tokens = ss[more\_than\_target\_tokens]

#Get the quantity of rows with more than the target sequence length

num\_rows\_with\_more\_than\_target\_tokens = len(rows\_with\_more\_than\_target\_tokens)

#Check if there are any rows with more than the target sequence length

if num\_rows\_with\_more\_than\_target\_tokens > 0:

print(f"Rows with more than {target\_sequence\_length} tokens: {num\_rows\_with\_more\_than\_target\_tokens}")

#Adjust the target sequence length to the maximum length of these rows

target\_sequence\_length = max(rows\_with\_more\_than\_target\_tokens['text'].apply(len))

print(f"Adjusting target sequence length to: {target\_sequence\_length}")

else:

print(f"No rows found with more than {target\_sequence\_length} tokens.")

break

#Now target\_sequence\_length contains the minimum value to have no rows with more tokens than that value

print(f"Minimum sequence length to accommodate all rows: {target\_sequence\_length}")

ss['text'].iloc[0]

#Tokenise text to numeric form for NN

#Create a tokenizer

tokenizer = Tokenizer(num\_words=vocab\_length, oov\_token="<OOV>")

tokenizer.fit\_on\_texts(ss['text'])

#Convert text to sequences of integers

sequences = tokenizer.texts\_to\_sequences(ss['text'])

#Check one example

sequences[0]

#Set the maximum sequence length

max\_sequence\_length = 60

#Iterate through the rows and pad the text data to the max length

for index, sequence in enumerate(sequences):

if len(sequence) < max\_sequence\_length:

#Pad the sequence with zeros at the end

padded\_sequence = sequence + [0] \* (max\_sequence\_length - len(sequence))

else:

#Truncate the sequence if it's longer than the maximum length

padded\_sequence = sequence[:max\_sequence\_length]

#Update the 'text' column with the padded sequence

ss.at[index, 'text'] = padded\_sequence

# Define feature and label columns

X = ss['text']

y = ss['label']

# Split data into test and train

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Convert training data and labels to numpy arrays

X\_train = np.array(X\_train)

y\_train = np.array(y\_train)

# Check the data shape (optional)

print("X\_train shape:", X\_train.shape)

print("y\_train shape:", y\_train.shape)

# Define the maximum sequence length you want to use

maxlen = 60 # Adjust this value as needed

# Initialize an empty list to store the padded sequences

X\_train\_padded = []

# Iterate through the tokenized sequences and pad each one

for sequence in X\_train:

# Check the length of the sequence

if len(sequence) < maxlen:

# If it's shorter than maxlen, pad it with zeros at the end (not the beginning)

padded\_sequence = sequence + [0] \* (maxlen - len(sequence))

else:

# If it's longer than maxlen, truncate it to maxlen

padded\_sequence = sequence[:maxlen]

# Add the padded sequence to the list

X\_train\_padded.append(padded\_sequence)

# Convert the list of lists to a NumPy array

X\_train\_padded = np.array(X\_train\_padded)

# Now X\_train\_padded is a NumPy array of shape (number\_of\_samples, maxlen)

print("X\_train\_padded shape:", X\_train\_padded.shape)

X\_train\_padded[10]

#Generate vocab from the word tokens

vocab = list(tokenizer.word\_index.keys())

vocab\_length = len(vocab)

#Calculate vocab length

#Create a tokenizer

tokenizer = Tokenizer()

#Fit the tokenizer on your text data

tokenizer.fit\_on\_texts(ss["text"])

#Calculate vocab\_length- add 1 for OOV words

vocab\_length = len(tokenizer.word\_index) + 1

vocab\_length

#Generate vocab from the word tokens

vocab = list(tokenizer.word\_index.keys())

vocab\_length = len(vocab)

#Check this is correct

vocab

#Create numpy array embedding matrix - start empty with a dimensionality of 300 corresponding to the vectors

embedding\_matrix = zeros((vocab\_length, 300))

#For each word in the vocab, check if it is present in the embeddings\_dictionary

for index, word in enumerate(vocab):

if index < vocab\_length:

embedding\_vector = all\_vectors.get(word)

#If it is there, the vector is added to the correct row in the embedding matrix - otherwise, it will just remain zeros

if embedding\_vector is not None:

embedding\_matrix[index] = embedding\_vector

print('Matrix:')

print(embedding\_matrix[1])

print('Vector:')

print(embedding\_vector[1])

#Check shape

embedding\_matrix.shape

#Apply the function to the df with the most words first

vocabulary = build\_vocabulary(df\_a\_emoji)

#Initialize an empty dictionary to store word vectors

word\_vectors = {}

#Iterate through words in your vocabulary

for word in vocabulary:

#Check if the word exists in the Word2Vec model

if word in word2vec\_model:

#Get the word vector

word\_vector = word2vec\_model[word]

#Store the word vector in the dictionary

word\_vectors[word] = word\_vector

#Check for duplicates of rows in df

print(df\_emoji.duplicated(subset=['text'], keep=False).sum())

#Check for null values in dfs

df\_emoji.isnull().sum()

#Set the maximum sequence length

max\_sequence\_length = 60

#Iterate through the rows and pad the text data to the max length

for index, sequence in enumerate(sequences):

if len(sequence) < max\_sequence\_length:

#Pad the sequence with zeros at the end

padded\_sequence = sequence + [0] \* (max\_sequence\_length - len(sequence))

else:

#Truncate the sequence if it's longer than the maximum length

padded\_sequence = sequence[:max\_sequence\_length]

#Update the 'text' column with the padded sequence

ss.at[index, 'text'] = padded\_sequence

# Define feature and label columns

X = ss['text']

y = ss['label']

# Split data into test and train

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Convert training data and labels to numpy arrays

X\_train = np.array(X\_train)

y\_train = np.array(y\_train)

# Check the data shape (optional)

print("X\_train shape:", X\_train.shape)

print("y\_train shape:", y\_train.shape)

X\_train[0]

# Define the maximum sequence length you want to use

maxlen = 60 # Adjust this value as needed

# Initialize an empty list to store the padded sequences

X\_train\_padded = []

# Iterate through the tokenized sequences and pad each one

for sequence in X\_train:

# Check the length of the sequence

if len(sequence) < maxlen:

# If it's shorter than maxlen, pad it with zeros at the end (not the beginning)

padded\_sequence = sequence + [0] \* (maxlen - len(sequence))

else:

# If it's longer than maxlen, truncate it to maxlen

padded\_sequence = sequence[:maxlen]

# Add the padded sequence to the list

X\_train\_padded.append(padded\_sequence)

# Convert the list of lists to a NumPy array

X\_train\_padded = np.array(X\_train\_padded)

# Now X\_train\_padded is a NumPy array of shape (number\_of\_samples, maxlen)

print("X\_train\_padded shape:", X\_train\_padded.shape)

X\_train\_padded[10]

#Make sure this is converting back to words in a way that makes sense

#Sequence of integers

sequence = X\_test\_padded[0]

#Convert sequence back to words

decoded\_words = [tokenizer.index\_word.get(idx, "<OOV>") for idx in sequence]

#Join the words to form the original text

original\_text = " ".join(decoded\_words)

#Check if this makes sense

print(original\_text)