Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Guadalajara

School of Engineering and Sciences

Comparing Techniques for Natural Language Processing in Sentiment Analysis

A research article presented by

Sofía Dalia Magallón Páramo (Student)

Submitted to the
School of Engineering and Sciences
in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Science and Technology

Zapopan, Jalisco, December, 2023

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Guadalajara

School of Engineering and Sciences

The committee members, hereby, certify that have read the research article presented by Sofía Dalia Magallón Páramo (Student) and that it is fully adequate in scope and quality as a partial requirement for the degree of Bachelor of Science in Computer Science and Technology.

Dr. Mahdi Zareei
Tecnológico de Monterrey
Principal Advisor

Alejandro de León Languré
Tecnológico de Monterrey
Committee Member

Dr. Luis Guillermo Hernández Rojas
Director of Undergraduate Studies
Computer Science and Technology
School of Engineering and Sciences

Zapopan, Jalisco, December, 2023

Declaration of Authorship

I, Sofía Dalia Magallón Páramo (Student), declare that this research article titled, ”Comparing Techniques for Natural Language Processing in Sentiment Analysis” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a bachelor degree at this University.
- Where any part of this article has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this dissertation is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Sofía Dalia Magallón Páramo (Student)
Zapopan, Jalisco, December, 2023

Comparing Techniques for Natural Language Processing in Sentiment Analysis

by

Sofía Dalia Magallón Páramo (Student)

Abstract

This study investigates the comparison between different Natural Language Processing (NLP) techniques in detecting emotions from textual data. Given the current state of NLP, and the growing popularity of the Artificial Intelligence (AI) research field, this research aims to shine some light on the field of Sentiment Analysis by comparing the performance of three different techniques: Rules, Neural Network, and Deep Learning, in the correct classification of six different emotions: Sadness, Anger, Love, Surprise, Fear, and Joy, using a dataset of over 16,000 labeled texts. The results show that while Rules are faster to compute and require less resources, models created using Neural Networks and Deep Learning demonstrated a performance improvement of 47% and 49% respectively in terms of accuracy against Rules, with a 59% accuracy. The study concludes that both Neural Networks and Deep Learning approaches are more powerful than Rules, making them more suitable for most applications of Emotion Classification, however, they require more computational resources.

List of Figures

2.1 Rule-based Approach 8
2.2 Rule-based Lexicons 8
2.3 Rule-based Prediction 9
2.4 Neural Network Approach 10
2.5 Rule-based Accuracy per Hyperparameter Set 12
2.6 Rule-based Accuracy per Counter Most Common 13
2.7 Neural Network Accuracy per Hyperparameter Set 14
2.8 Neural Network Accuracy Loss across Epochs 14
2.9 Deep Learning Accuracy per Hyperparameter Set 15
2.10 Deep Learning Accuracy Loss across Epochs 16

List of Tables

2.1 Rule-based Hyperparameters 9
2.2 Neural Network Hyperparameters 10
2.3 Deep Learning Hyperparameters 11
2.4 Rule-based Results 12
2.5 Neural Network Results 13
2.6 Deep Learning Results 15

Contents

Abstract **iii**

List of Figures **iv**

List of Tables **v**

1 Introduction **1**

1.1 Motivation 1

1.2 Literature Review 1

1.2.1 Symbolic Approaches 2

1.2.2 Statistical Approaches 2

1.2.3 Connectionist Approaches 2

1.2.4 Data preparation 2

1.2.5 N-Grams 3

1.2.6 Parameter Estimation 4

1.2.7 Neural Networks 5

1.2.8 Long Short-Term Memory 6

2 Comparing Techniques **7**

2.1 Experimentation 7

2.1.1 Rule-based Approach 7

2.1.2 Neural Network 9

2.1.3 Deep Learning 10

2.2 Results 11

2.2.1 Rule-based Approach 12

2.2.2 Neural Network 13

2.2.3 Deep Learning 15

2.3 Discussion 16

2.4 Conclusion 16

A Rules-based Code **18**

B Neural Network Code **22**

C Deep Learning Code **27**

Bibliography **33**

Chapter 1

Introduction

1.1 Motivation

As technology advances, the lifestyle of people has been shifting towards being surrounded by Artificial Intelligence, in a sense, the progress seems overwhelmingly fast and massive. The development of these technologies is, however, no easy task, as the complexity of modern systems also grow exponentially, both using a large variety of techniques and needing extremely expensive equipment with huge architectures.

Natural Language Processing (NLP) is an immense field within Artificial Intelligence, which is in itself a subset of Computer Science, however, something interesting happens when trying to define what NLP conveys to, that is, an intersection between Linguistics, Computer Science and Statistics. As it turns out, featuring the ability for computers to understand natural languages is not trivial.

This research is motivated by the extensive and diverse knowledge needed in this task, to identify which some of the most popular techniques used for NLP may be best suitable for different contexts, aiming to shine some light into the best usage of resources and techniques for a particular objective, rather than following the current trend to use Large Language Models as a one-size fits-all solution; to generate more information so better decisions are made in a constantly developed industry that is shaping the future of humanity concurrently.

1.2 Literature Review

As previously mentioned, Natural Language Processing (NLP) may be defined as a field at the intersection of Computer Science and Linguistics, aiming to enable computers to understand, interpret, and generate human language [6]. This is, however, not the only definition, as NLP is a large and complex field. As such, it's important to understand some definitions, such as levels of language, a set of descriptions on how Natural Languages carry not only meaning bun underlaying information and relations on different orders. These may be enumerated as follows:

1. Phonology.
2. Morphology.
3. Lexical.
4. Syntactic.
5. Semantic.
6. Discourse.
7. Pragmatic.

Ordered from higher to lower, however, not all levels are analyzed in NLP, and the lower levels have proven to be enough to perform complex analysis as part of a technique of NLP, for instance, Lexical level on the use of lexicons. There are many approaches that may be used to perform NLP, each with different implications.

1.2.1 Symbolic Approaches

They usually rely on manually created rules, making use of dictionaries and ontologies. The technique depends on a deep analysis of linguistic phenomena and is based on the explicit representation of facts about language through well-understood knowledge representation [8].

1.2.2 Statistical Approaches

These techniques employ mathematical techniques often relying on large text corpora to have enough data to create general linguistic models that accurately represent the linguistic phenomena. They are based on observation of examples rather than employing prior knowledge on language.

1.2.3 Connectionist Approaches

Similar to statistical approaches, they model linguistic phenomena via generalized models, with the additional implementation of theories of representation, allowing transformation, inference, and manipulation of logic, often used on artificial neural networks in combination with deep learning for both supervised and unsupervised learning.

1.2.4 Data preparation

Invariant to these approaches, though, there is a set of efforts that may be used to overall improve the results any of the models may generate. As in any other machine learning application, data is extremely important when trying to model the behavior of a phenome [4].

In particular for linguistics, there are some specific processes that may be followed to prepare the data, transforming raw text input into a structured and manageable format, not only building composition, but also reducing dimensionality resulting in both enhancement of quality and performance improvement on the model.

- Lowercasing: Turn all characters to lowercase
- Punctuation removal: Removing all punctuation marks.
- Handling of special characters: Dealing with symbols and emojis.
- Elimination of stop words: Removing words that may not contribute to the context, such as ”and”, “the”, etc.
- Tokenization: The sub-division of smaller elements from text.

Tokenization in particular is crucial for dealing both with artificial and natural languages, however, choosing what may be considered a token in natural languages is not as trivial as it may seem; while tokens are widely used in applications like compilers, the complexity of the problem grows rapidly when encountering natural languages and their rich variety of rules and instances, so many methods may be used, the most popular one being the separation of tokens by words from a character stream [3].

1.2.5 N-Grams

Another technique for the generation of language models is N-Grams, a method that organizes sequences of length n composed by words in a collection, often named after the length of the instance collection itself, for instance, the term given to a collection of sequences formed by two words is bi-gram, three words tri-gram, and so on [1].

While n-grams are one of the most popular methods for language modeling, they are not able to outperform other advanced techniques, such as neural networks, however, they are proven to be good on training time and low-resource settings. Some of the most popular techniques are:

- Kneser-Ney Smoothing: Introducing a discounting method to estimate probability based on context.
- Modified Kneser-Ney Smoothing: Discounting method more sensitive to existing mass.
- Generalized Kneser-Keny Smoothing: Richer parametrization of discount method to allow more flexibility [9].

These different techniques are useful in different scenarios, even though they seem to me an upgrade of the last one, however, the extra flexibility provided by the last one with the introduction of parametrization is certainly well received in an ongoing research field.

1.2.6 Parameter Estimation

Having outlined the concept and variants of N-Gram models and their applicability in certain scenarios, attention must now be turned to the critical aspect of Parameter Estimation in language modeling. This segment forms a pivotal bridge between the basic structural formation of language models and the more nuanced process of fine-tuning to achieve higher accuracy and efficiency.

Parameter Estimation is integral to the development and optimization of language models. It involves methodologies and techniques that determine the best parameters to accurately represent the linguistic patterns captured by language models. This process is foundational in optimization and in advanced language modeling techniques. Two general methods for this task, and coincidentally, the most popular, are the following:

- Maximum Likelihood Estimation (MLE)
- Least Squares Estimation (LSE)

However, MLE has proven to be more relevant in the scientific context due to its large applicability and strong fundamentals, contrary to LSE often getting discredited as a tool for hypothesis testing.

MLE is also considered pre-requisite for Bayesian inference when modeling missing data in random effects and in model selection criteria, providing sufficiency, consistency, efficiency, and invariance [7].

The method consists in identifying the population that most likely has generated a certain number of observations, also called data. Particularly in this context, data is represented as a vector that comes from a random sample from an unknown population. Data is represented in a probability distribution, with which model parameters are associated. As the parameters change in value, the distribution's shape fluctuates, leading to the objective of optimizing the values that maximize the probability of the observed data.

There are analytic approaches to do the optimization making use of partial derivates, often relying on logarithmic variations of the probability function to solve the maximization problem, however, this method is not viable in all cases and functions increase in complexity as dimensionality grows, leading to not always analytically solvable equations.

Often, the problem requires heuristics and stochastic numerical methods to gain field on the growing complexity. One of the most popular methods to achieve this is Gradient Descent, leveraging the technique to the exploration of multiple dimensions without solving analytically. This is, however, not perfect, as it's easy to encounter the local maxima problem, causing the algorithm to get stuck on a sub-optimal solution, but there are several alternatives to face that problem, such as using multiple starting points, inserting variation in the optimization algorithms, and making use of regularization.

Since the final objective of a model is to generalize and predict new data based on observations, the use of these techniques may be used as additional tools to help different approaches to achieve that.

1.2.7 Neural Networks

Artificial Neural Networks, as their name suggest, are a series of systems that follow a paradigm inspired by the way a real biological nervous system work. ANNs are a major topic in computer science and artificial intelligence, therefore, it would be impossible to cover all aspects of them, however, focusing on some relevant sections is beneficial to explain their usage and approach in the analysis of Natural Language Processing.

Neural Networks consist of multiple layers and nodes interconnected forming a network topology. Nodes are usually called neurons, and every single one of them has a simple processing task: Colleting inputs and generating a single output to all subscribers.

The network topology of a Neural Network must be a directed graph with layers only connected forward, forming no cycles, and with each node conveying a single signal labeled by strength, usually called weight.

Neurons are simple linear functions, usually containing an activation function that actively limits the range of values a given neuron may output, avoiding overscale and smoothing the outputs, increasing the performance of a model. Neurons by themselves only may model linear behavior, however, forming a network allows them to model very complex phenomena, even allowing techniques to simulate non-linearity, such as logarithmic sigmoid functions.

ANNs usually allow a series of parameters to be adjusted, and they are very flexible to different problems. These parameters are often chosen at random or by a human operator with prior knowledge on the parametrization of the problem of interest.Since most of the terminology used in ANNs is inspired by the nervous system, it is relevant to mention some of the most commonly used terms, such as a synapse, used to describe a connection between two neurons. When the weight of a synapse is positive the term used is excitatory synapse, while a negative weight on a synapse is described as an inhibitory synapse.

With this knowledge on ANNs it may be possible to start building a system, although, there are different procedures to train a model. Some of the most relevant are:

- Supervised learning: Inputs have a target output. Needs a Fed Forward Neural Network to account for error between predictions and observations.
- Unsupervised learning: Data is provided with no label. The neural network tries to learn from any apparent pattern, responding to clusters and identifying statistically significant features.
- Reinforcement learning: System gets feedback from a simulation, either in form of a reward, or as punishment [2].

Once defined what a Neural Network is, it is important to describe some of the techniques used on top of the neural network that will be employed during the experimentation process.

1.2.8 Long Short-Term Memory

Working with Neural Networks usually comes with some challenges, such as the exponential decay of backpropagation or the blow-up of error signals. Long Short-Term Memory (LSTM) helps overcome these issues by bridging minimal time lags exceeding a given number of discrete time steps, enforcing a constant error flow through special units called constant error carrousels.

To achieve this, the LSTM architecture implements memory cells and gate units to prevent error decay and allow the network to learn over long time intervals. Also, introducing multiplicative input and output gates on this implementation effectively protects the content of the cell memory from irrelevant inputs and ultimately prevents the perturbation and distortion of other units.

The core components of a gate unit are the following:

- Input gate: Determines how much of the new information will be restored in the cell state.
- Forget gate: Decides what portion of existing information in the cell state is discarded.
- Output gate: Controls the extent to which the vale in the cell influences the output [5].

Implementing LSTM in Neural Networks effectively addresses the natural limitations of the approach when encountering long training intervals, also allowing the network to handle increasingly complex tasks that involve long time lags.

Chapter 2

Comparing Techniques

2.1 Experimentation

The three approaches to compare each have a set of properties and parameters, however, the training data is the same for all of them.

The training data consists of 16,000 lines of text, each of which has an emotion that the text reflects. These two columns are separated by a semi-colon. Each line of text represents a different observation.

To conduct the testing, an exclusive set of 2000 lines was separated. An extra exclusive set 2000 lines was separated for model validation purposes. The total data sums up to 20,000 observations.

The experiments conducted are three different approaches to generate a language model:

- Rule-based Approach
- Neural Network
- Deep learning

The specifics for each experiment are described in the following section.

2.1.1 Rule-based Approach

This is a simple symbolic approach implemented in Python, making use of lexicons to predict the emotion of the given line. Figure 2.1 depicts the general algorithm.

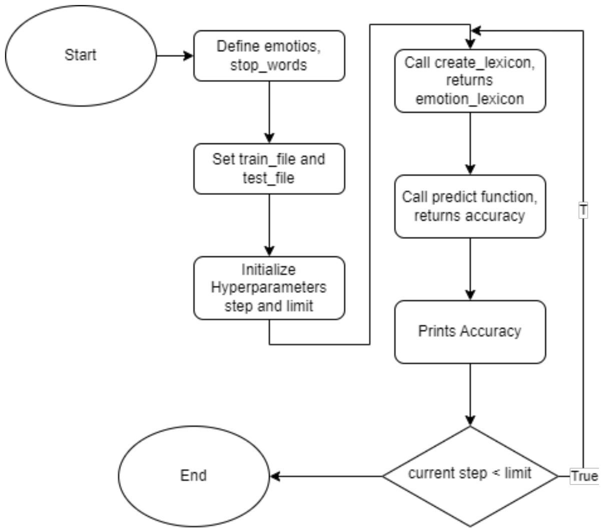
Start
Define emotios, stop_words
Set train_file and test_file
Initialize Hyperparameters step and limit
Call create_lexicon, returns emotion_lexicon
Call predict function, returns accuracy
Prints Accuracy
current step < limit
True
T
End

Figure 2.1: Rule-based Approach

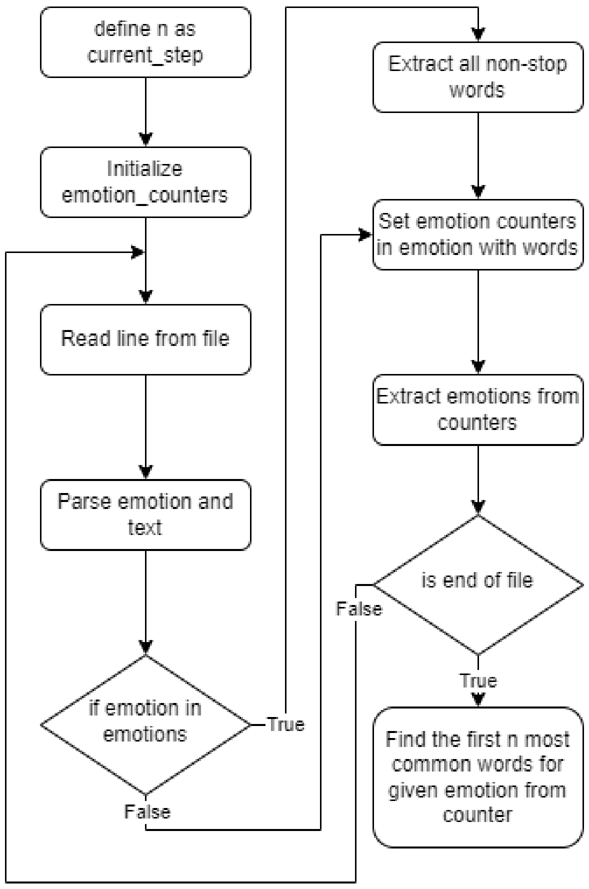
define n as current_step
Initialize emotion_counters
Read line from file
Parse emotion and text
if emotion in emotions
True
False
Extract all non-stop words
Set emotion counters in emotion with words
Extract emotions from counters
is end of file
False
True
Find the first n most common words for given emotion from counter

Figure 2.2: Rule-based Lexicons

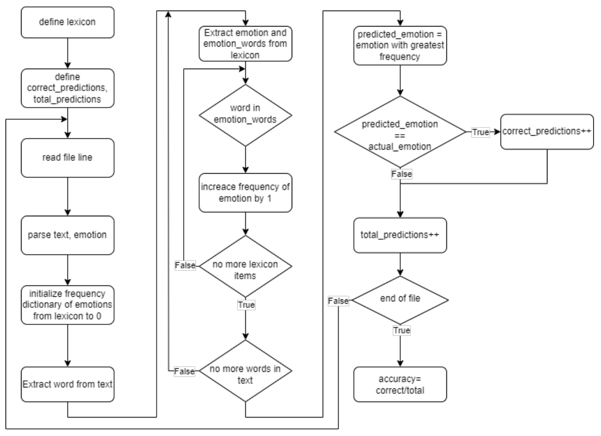
define lexicon
define correct_predictions, total_predictions
read file line
parse text, emotion
initialize frequency dictionary of emotions from lexicon to 0
Extract word from text
Extract emotion and emotion_words from lexicon
word in emotion_words
increace frequency of emotion by 1
no more lexicon items
False
True
no more words in text
False
predicted_emotion = emotion with greatest frequency
predicted_emotion == actual_emotion
True
correct_predictions++
False
total_predictions++
end of file
False
True
accuracy= correct/total

Figure 2.3: Rule-based Prediction

Figure 2.2 describes the creation of a lexicon, finally, figure 2.3 describes the prediction of emotions.

The two hyperparameters for this approach were Steps and Word Max Count. To choose them, a set of diverging quantities was generated, starting with 5 steps and 200 words, then diverging in both directions.

The full set of hyperparameters chosen is visible on the table 2.1.

Steps	Word Max Count
1	100
1	200
3	100
3	200
5	200
10	200
15	300
20	400
20	800
20	1000

Table 2.1: Hyperparameter for Rule-based Approach

2.1.2 Neural Network

The Neural Network approach was implemented in Python using PyTorch as framework for the Neural Network. Figure 2.4 describes the process of the algorithm found in the experiment.

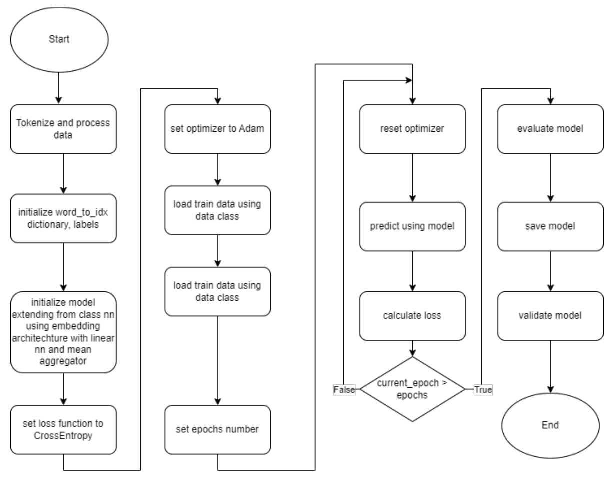
Start
Tokenize and process data
initialize word_to_idx dictionary, labels
initialize model extending from class nn using embedding architechture with linear nn and mean aggregator
set loss function to CrossEntropy
set optimizer to Adam
load train data using data class
load train data using data class
set epochs number
reset optimizer
predict using model
calculate loss
current_epoch > epochs
False
True
evaluate model
save model
validate model
End

Figure 2.4: Neural Network Approach

This neural network implements an embedding layer and makes use of hidden layers topology. It is not possible to describe or depict the actual network since the dimensionality of the experiment fluctuates as hyperparameter.

The full set of hyperparameters is Max Sequence Length, Epochs and Embedded Dimension. Again, the values were chosen by diverging starting values of 50, 50 and 100 respectively, however, trying to permutate possibilities as they diverge, as shown in the table 2.2.

Max Sequence Length	Epochs	Embedded Dimension
25	50	50
25	50	100
25	50	200
25	100	100
25	100	200
50	50	100
50	50	200
50	100	100
50	100	200
100	50	100
100	50	200

Table 2.2: Hyperparameter for Neural Network Approach

2.1.3 Deep Learning

The Deep Learning approach has a similar algorithm to the Neural Network approach, and follow the same architecture, however, the Deep Learning Neural Network implements LSTM as part of the architecture, also increasing the number of epochs to get into deep learning.

The chosen hyperparameters were Batch Size, Embedding Dimension, Hidden Dimension and Epochs (in thousands). The values were chosen to have different permutations of increasing values in epoch with different cases of the rest of parameters, as shown in the table 2.3.

Batch Size	Embedding Dimension	Hidden Dimension	Epochs (k)
32	100	128	5
64	100	128	5
32	200	128	5
32	100	256	5
32	100	128	10
32	100	128	20
32	100	128	30
32	100	128	40
32	100	128	50
32	100	128	100

Table 2.3: Hyperparameter for Deep Learning Approach

2.2 Results

The success metric chosen for these experiments was Accuracy, which is defined as the number of lines the model successfully predicted divided by the total number of lines tested.

Since the first experiment has a very different approach to the latter two, which are very similar, the figures shown to describe them will vary.

The following figures will show a table of results for each hyper parameter chosen as well as a bar graphic depicting each result. Experimentation runs were conducted once per hyperparameter, and the Hyperparameter columns will indicate the horizontal axis on the bar graphics.

2.2.1 Rule-based Approach

Hyperparameter	Steps	Word Max Count	Accuracy
1	1	100	43.65%
2	1	200	56.25%
3	3	100	43.35%
4	3	200	55.60%
5	5	200	55.70%
6	10	200	55.70%
7	15	300	59.80%
8	20	400	58.65%
9	20	800	57.40%
10	20	1000	55.50%

Table 2.4: Accuracy for Rule-based Set Up

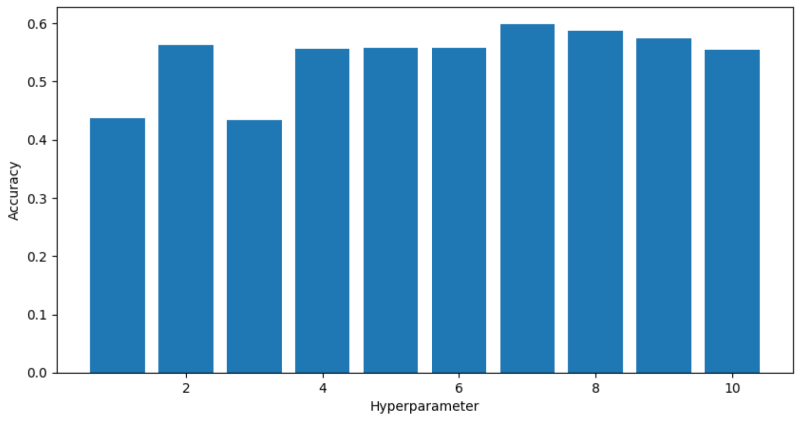

Figure 2.5: Rule-based Accuracy per Hyperparameter Set

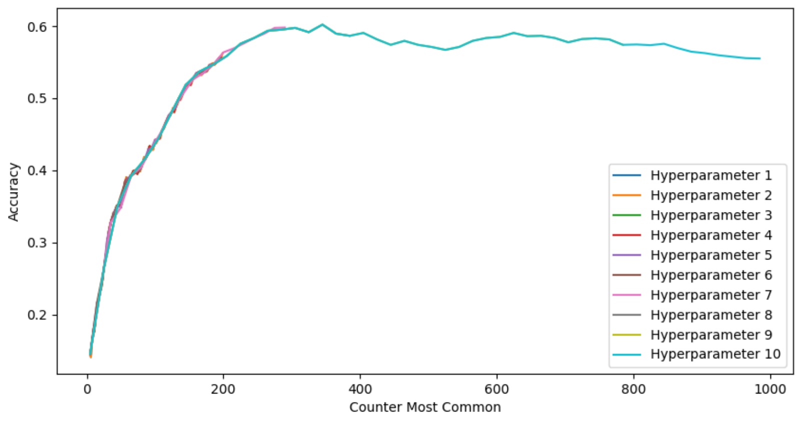

Figure 2.6: Rule-based Accuracy per Counter Most Common

2.2.2 Neural Network

Hyperparameter	Max Sequence Length	Epochs	Embedded Dimension	Accuracy
1	25	50	50	84.65%
2	25	50	100	83.70%
3	25	50	200	83.75%
4	25	100	100	81.35%
5	25	100	200	81.30%
6	50	50	100	88.00%
7	50	50	200	86.95%
8	50	100	100	84.55%
9	50	100	200	83.85%
10	100	50	100	88.40%
11	100	50	200	88.15%

Table 2.5: Accuracy for Neural Network Set Up

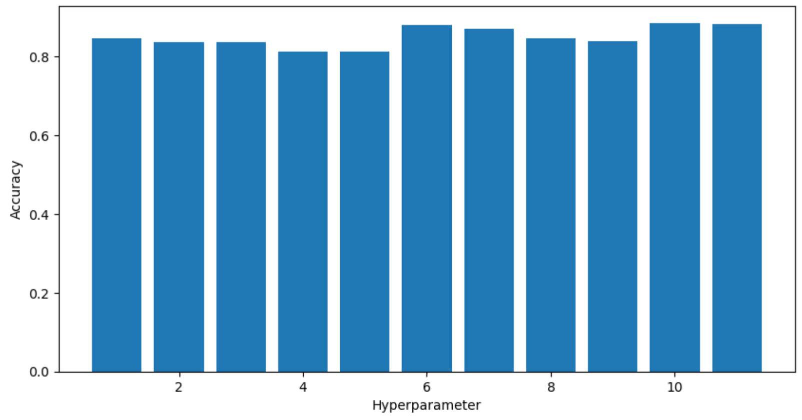

Figure 2.7: Neural Network Accuracy per Hyperparameter Set

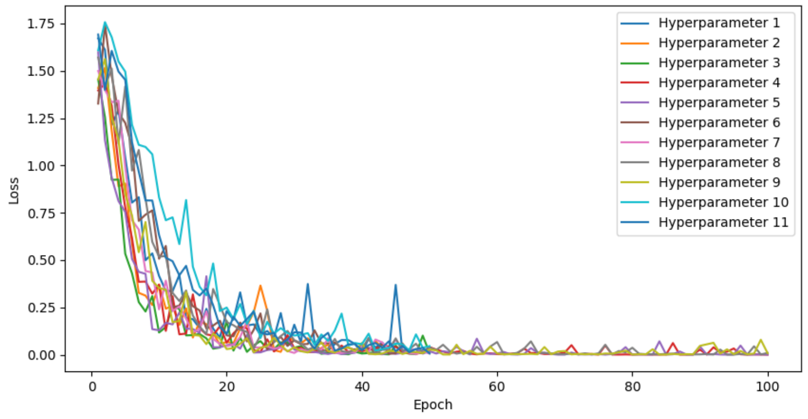

Figure 2.8: Neural Network Accuracy Loss across Epochs

2.2.3 Deep Learning

Hyper-parameter	Batch Size	Embedding Dimension	Hidden Dimen-sion	Epochs (k)	Accuracy
1	32	100	128	5	28.95%
2	64	100	128	5	34.60%
3	32	200	128	5	60.80%
4	32	100	256	5	37.95%
5	32	100	128	10	68.2%
6	32	100	128	20	88.9%
7	32	100	128	30	89.1%
8	32	100	128	40	88.55%
9	32	100	128	50	88.95%
10	32	100	128	100	88.70%

Table 2.6: Accuracy for Deep Learning Set Up

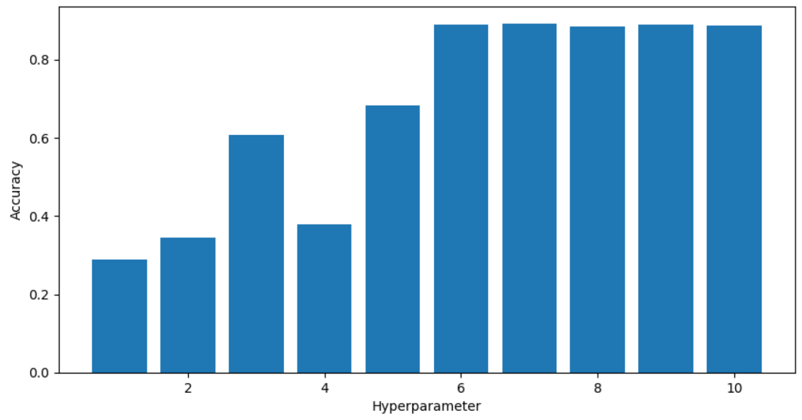

Figure 2.9: Deep Learning Accuracy per Hyperparameter Set

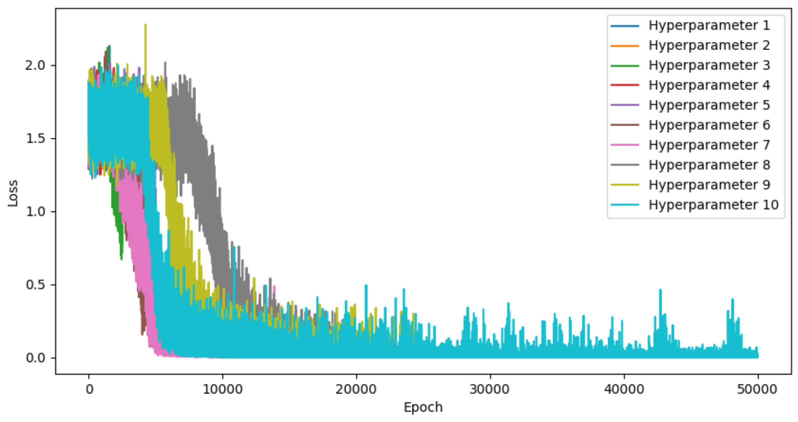

Figure 2.10: Deep Learning Accuracy Loss across Epochs

2.3 Discussion

The results clearly show a difference between the accuracy of the Rule-Based Approach versus the other two, being the former the worst performer of them all, however, it's important to recall the limitations of the rule-based approach as it generates a model that is not relevantly accurate, but most of the time gets the emotion from the observation right. Also, this approach is faster than the other two requiring not only less iterations, but also involving fewer complex operations.

It is also important to mention that increasing the counter most common value affected the model negatively, as shown in figure 2.6. This is due to grabbing all the words in an observation as part of the lexicon, instead of choosing the top (best) ones.

As for the Neural Network, the accuracy was similar in all results, doing best on the hyperparameter 6 as shown in table 2.5. Also, no highlight in improvement is visible on loss, seeing all hyperparameters converge to approximately the same loss the more epochs passed.

Finally, the Deep Learning Neural Network showed a poor performance in the first epochs, but the loss function quickly converged the more time lags passed (see figure 2.10). As mentioned in the literature review, this is due to the strength of deep learning models when learning over large amounts of data and large time lags. The performance grew the more epochs used, but it stabilized after 20 thousand epochs, as shown in table 2.6 and figure 2.9.

2.4 Conclusion

The Natural Language Processing approaches for Emotion Classification serve different purposes and may be used on different contexts depending on the kind of data and computational power available.

However, the efficiency of the generated models by each approach is measurable, giving a clear image on the performance of each one in a controlled problem.

After the experimentation and results, it's clear that the Neural Network and Deep Learning approaches are better suited for a highly complex and large dataset on an environment with high computational resources, by reaching accuracies of 88

While both Neural Network and Deep Learning approaches have similar efficiency, the Neural Network experiment reached peak performance in less computational time as measured by epochs and function calling. Further research would have to be conducted in order to find whether there's any major advantage or context in which a Deep Learning model may perform better than the Neural Network, despite running longer on computational time.

Appendix A

Rules-based Code

```
import re
from collections import Counter
import matplotlib.pyplot as plt

emotions = ['love', 'fear', 'sadness', 'surprise', 'joy', '
   anger']

stop_words = set(['i', 'me', 'my', 'myself', 'we', 'our', '
   ours', 'ourselves', 'you', "you're", "you've", "you'll", "
   you'd", 'your', 'yours', 'yourself', 'yourselves',
                 'he', 'him', 'his', 'himself', 'she', "she's"
                    , 'her', 'hers', 'herself', 'it', "it's",
                    'its', 'itself', 'they', 'them', 'their',
                    'theirs', 'themselves',
                 'what', 'which', 'who', 'whom', 'this', 'that
                    ', "that'll", 'these', 'those', 'am', 'is'
                    , 'are', 'was', 'were', 'be', 'been', '
                    being', 'have', 'has', 'had',
                 'having', 'do', 'does', 'did', 'doing', 'a',
                    'an', 'the', 'and', 'but', 'if', 'or', '
                    because', 'as', 'until', 'while', 'of', '
                    at', 'by', 'for', 'with', 'about',
                 'against', 'between', 'into', 'through', '
                    during', 'before', 'after', 'above', '
                    below', 'to', 'from', 'up', 'down', 'in',
                    'out', 'on', 'off', 'over', 'under',
                 'again', 'further', 'then', 'once', 'here', '
                    there', 'when', 'where', 'why', 'how', '
                    all', 'any', 'both', 'each', 'few', 'more'
                    , 'most', 'other', 'some',
                 'such', 'no', 'nor', 'not', 'only', 'own', '
                    same', 'so', 'than', 'too', 'very', 's', '
                    t', 'can', 'will', 'just', 'don', "don't",
                     'should', "should've",
                 'now', 'd', 'll', 'm', 'o', 're', 've', 'y',
                    'ain', 'aren', "aren't", 'couldn', "couldn
                    't", 'didn', "didn't", 'doesn', "doesn't",
                     'hadn', "hadn't",
                 'hasn', "hasn't", 'haven', "haven't", 'isn',
                    "isn't", 'ma', 'mightn', "mightn't", '
                    mustn', "mustn't", 'needn', "needn't", '
                    shan', "shan't", 'shouldn',
                 "shouldn't", 'wasn', "wasn't", 'weren', "
                    weren't", 'won', "won't", 'wouldn', "
                    wouldn't", 'feeling', 'feel', 'really', '
                    im', 'like',
                 'know', 'get', 'ive', "im'", 'stil', 'even',
                    'time', 'want', 'one', 'cant', 'think', '
                    go', 'much', 'never', 'day', 'back', 'see'
                    , 'still', 'make', 'thing',
                 'would', "would'", "could'", 'little',])


def create_lexicon(file_path, counter_most_common):

    emotion_counters = {emotion: Counter() for emotion in
       emotions}

    with open(file_path, 'r', encoding='utf-8') as file:
        for line in file:
            text, emotion = line.strip().split(';')
            if emotion in emotions:
                words = [word for word in re.findall(r'\w+',
                   text.lower()) if word not in stop_words]
                emotion_counters[emotion].update(words)

    emotion_lexicon = {emotion: [word for word, _ in counter.
       most_common(counter_most_common)] for emotion, counter
       in emotion_counters.items()}

    return emotion_lexicon

def predict(file_path, lexicon):
    correct_predictions = 0
    total_predictions = 0
    with open(file_path, 'r', encoding='utf-8') as file:
        for line in file:
            text, actual_emotion = line.strip().split(';')
            words = set(re.findall(r'\w+', text.lower()))

            emotion_scores = {emotion: 0 for emotion in
               lexicon}

            for word in words:
                for emotion, emotion_words in lexicon.items():
                    if word in emotion_words:
                        emotion_scores[emotion] += 1

            predicted_emotion = max(emotion_scores, key=
               emotion_scores.get)

            if predicted_emotion == actual_emotion:
                correct_predictions += 1
            total_predictions += 1

    accuracy = correct_predictions / total_predictions
    return accuracy



def main(step = 5, wc = 200):
    train_file_path = 'data/train.txt'
    test_file_path = 'data/test.txt'

    # Hyperparameters:
    increment_step = step
    max_word_count = wc
    accuracy_list = []

    for counter_most_common in range(5, max_word_count,
       increment_step):
        lexicon = create_lexicon(train_file_path,
           counter_most_common)
        accuracy = predict(test_file_path, lexicon)
        accuracy_list.append(accuracy)
        print(f"Counter Most Common: {counter_most_common}, 
           Prediction Accuracy: {accuracy*100:.2f}%")

    return accuracy_list
```

```
# Run 10 times with different hyperparameters and plot the
   accuracy over steps in the same plot with different colors
# When runs have different step sizes, scale the x-axis to the
    same size
# Then plot the last accuracy for each hyperparameter in a bar
    plot

hyperparameters = [(1,100), (1,200), (3,100), (3, 200), (5,
   200), (10, 200), (15, 300), (20, 400), (20, 800), (20,
   1000)]

accuracy_matrix = []
for step, wc in hyperparameters:
    accuracy_matrix.append(main(step, wc))

print(accuracy_matrix)

plt.figure(figsize=(10, 5))
for i in range(len(accuracy_matrix)):
    plt.plot(range(5, hyperparameters[i][1], hyperparameters[i
       ][0]), accuracy_matrix[i], label=f'Hyperparameter■{i+1}
       ')
plt.xlabel('Counter■Most■Common')
plt.ylabel('Accuracy')
plt.legend()
plt.savefig('accuracy_plot.png')

plt.figure(figsize=(10, 5))
plt.bar(range(1, len(accuracy_matrix)+1), [accuracy[-1] for
   accuracy in accuracy_matrix])
plt.xlabel('Hyperparameter')
plt.ylabel('Accuracy')
plt.savefig('accuracy_bar.png')

for i in range(len(accuracy_matrix)):
    print(f'Hyperparameter■{i+1}:■{accuracy_matrix[i
       ][-1]*100:.2f}%')
```

Appendix B

Neural Network Code

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from sklearn.preprocessing import LabelEncoder
from collections import Counter
import re
import matplotlib.pyplot as plt

# Tokenize and pad/truncate
def tokenize(text, max_length):
    tokens = re.findall(r'\w+', text.lower())
    if len(tokens) > max_length:
        return tokens[:max_length]
    return tokens + ['<PAD>'] * (max_length - len(tokens))

def load_data(file_path, max_length):
    texts, labels = [], []
    with open(file_path, 'r', encoding='utf-8') as file:
        for line in file:
            text, label = line.strip().split(';')
            texts.append(tokenize(text, max_length))
            labels.append(label)
    return texts, labels

class EmotionDataset(Dataset):
    def __init__(self, texts, labels, word_to_idx):
        self.texts = [[word_to_idx.get(word, word_to_idx['<UNK
          >']) for word in text] for text in texts]
        self.labels = labels
    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        return torch.tensor(self.texts[idx]), torch.tensor(
           self.labels[idx])

class EmotionClassifier(nn.Module):
    def __init__(self, vocab_size, EMBED_DIM, num_classes):
        super(EmotionClassifier, self).__init__()
        self.embedding = nn.Embedding(vocab_size, EMBED_DIM)
        self.fc = nn.Linear(EMBED_DIM, num_classes)

    def forward(self, x):
        x = self.embedding(x)
        x = torch.mean(x, dim=1)
        return self.fc(x)

# Function to evaluate model
def evaluate_model(model, data_loader):
    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        for texts, labels in data_loader:
            outputs = model(texts)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return correct / total

# Hyperparameters
hyperparameters = [
    {'MAX_SEQ_LENGTH': 25, 'EPOCHS': 50, 'EMBED_DIM': 50},
    {'MAX_SEQ_LENGTH': 25, 'EPOCHS': 50, 'EMBED_DIM': 100},
    {'MAX_SEQ_LENGTH': 25, 'EPOCHS': 50, 'EMBED_DIM': 200},
    {'MAX_SEQ_LENGTH': 25, 'EPOCHS': 100, 'EMBED_DIM': 100},
    {'MAX_SEQ_LENGTH': 25, 'EPOCHS': 100, 'EMBED_DIM': 200},
    {'MAX_SEQ_LENGTH': 50, 'EPOCHS': 50, 'EMBED_DIM': 100},
    {'MAX_SEQ_LENGTH': 50, 'EPOCHS': 50, 'EMBED_DIM': 200},
    {'MAX_SEQ_LENGTH': 50, 'EPOCHS': 100, 'EMBED_DIM': 100},
    {'MAX_SEQ_LENGTH': 50, 'EPOCHS': 100, 'EMBED_DIM': 200},
    {'MAX_SEQ_LENGTH': 100, 'EPOCHS': 50, 'EMBED_DIM': 100},
    {'MAX_SEQ_LENGTH': 100, 'EPOCHS': 50, 'EMBED_DIM': 200},
]
```

```
results = []
loss_per_epoch_matrix = []

def main(MAX_SEQ_LENGTH, EPOCHS, EMBED_DIM):
    loss_list = []
    # Load and process data
    train_texts, train_labels = load_data('data/train.txt',
        MAX_SEQ_LENGTH)
    test_texts, test_labels = load_data('data/test.txt',
        MAX_SEQ_LENGTH)

    # Encode labels
    label_encoder = LabelEncoder()
    train_labels = label_encoder.fit_transform(train_labels)
    test_labels = label_encoder.transform(test_labels)

    # Build vocabulary
    all_words = [word for text in train_texts for word in text
        ]
    word_counts = Counter(all_words)
    vocab = ['<PAD>', '<UNK>'] + [word for word, count in
        word_counts.items() if count > 1]
    word_to_idx = {word: idx for idx, word in enumerate(vocab)
        }

    # Model parameters
    vocab_size = len(vocab)

    num_classes = len(label_encoder.classes_)

    # Model, Loss, and Optimizer
    model = EmotionClassifier(vocab_size, EMBED_DIM,
        num_classes)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    # Data loaders
    train_dataset = EmotionDataset(train_texts, train_labels,
        word_to_idx)
    train_loader = DataLoader(train_dataset, batch_size=32,
        shuffle=True)

    test_dataset = EmotionDataset(test_texts, test_labels,
        word_to_idx)
```

```
test_loader = DataLoader(test_dataset, batch_size=32,
   shuffle=False)

# Training loop
for epoch in range(EPOCHS):
    for texts, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(texts)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
    print(f'Epoch {epoch+1}, Loss: {loss.item()}')
    loss_list.append(loss.item())

# Evaluate the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for texts, labels in test_loader:
        outputs = model(texts)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = correct / total
print(f'Accuracy: {accuracy*100:.2f}%')

# Save the model to disk
torch.save(model.state_dict(), 'emotion_classifier_model.
   pth')

# Load the model from disk
loaded_model = EmotionClassifier(vocab_size, EMBED_DIM,
   num_classes)
loaded_model.load_state_dict(torch.load('
   emotion_classifier_model.pth'))



# Load and process validation data
val_texts, val_labels = load_data('data/val.txt',
   MAX_SEQ_LENGTH)
val_labels = label_encoder.transform(val_labels)
```

```
    val_dataset = EmotionDataset(val_texts, val_labels,
        word_to_idx)
    val_loader = DataLoader(val_dataset, batch_size=32,
        shuffle=False)

    # Evaluate the loaded model on validation data
    accuracy = evaluate_model(loaded_model, val_loader)
    print(f'Validation■Accuracy:■{accuracy*100:.2f}%')

    results.append(accuracy)
    loss_per_epoch_matrix.append(loss_list)

for hyperparameter in hyperparameters:
    main(**hyperparameter)


plt.figure(figsize=(10, 5))
for i in range(len(loss_per_epoch_matrix)):
    plt.plot(range(1, hyperparameters[i]['EPOCHS']+1),
        loss_per_epoch_matrix[i], label=f'Hyperparameter■{i+1}'
        )
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.savefig('loss_plot_nn.png')

plt.figure(figsize=(10, 5))
plt.bar(range(1, len(results)+1), results)
plt.xlabel('Hyperparameter')
plt.ylabel('Accuracy')
plt.savefig('accuracy_bar_nn.png')


for i in range(len(results)):
    print(f'Hyperparameter■{i+1}:■{results[i]*100:.2f}%')
```

Appendix C

Deep Learning Code

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, Dataset
from collections import Counter
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from torch.nn.utils.rnn import pad_sequence
import matplotlib.pyplot as plt

# Parameters


# Load and preprocess data
def load_data(file_path):
    df = pd.read_csv(file_path, delimiter=';', header=None,
       names=['text', 'emotion'])
    return df['text'].values, df['emotion'].values

train_texts, train_labels = load_data('data/train.txt')
test_texts, test_labels = load_data('data/test.txt')

# Build vocabulary
word_counter = Counter()
for text in train_texts:
    word_counter.update(text.split())
vocab = ['<PAD>', '<UNK>'] + [word for word, freq in
   word_counter.items() if freq > 1]
word_to_idx = {word: idx for idx, word in enumerate(vocab)}
vocab_size = len(vocab)
```

```
# Tokenize and encode labels
def tokenize(text):
    return [word_to_idx.get(word, word_to_idx['<UNK>']) for
       word in text.split()]

def encode_labels(labels):
    label_encoder = LabelEncoder()
    return label_encoder.fit_transform(labels), label_encoder


# Dataset
class EmotionDataset(Dataset):
    def __init__(self, texts, labels, word_to_idx):
        self.texts = [torch.tensor(tokenize(text)) for text in
            texts]
        self.labels = torch.tensor(labels)

    def __len__(self):
        return len(self.texts)

    def __getitem__(self, idx):
        return self.texts[idx], self.labels[idx]

# Padding function
def collate_fn(batch):
    texts, labels = zip(*batch)
    texts = pad_sequence(texts, batch_first=True,
       padding_value=word_to_idx['<PAD>'])
    return texts, torch.tensor(labels)

# LSTM Model
class LSTMEmotionClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim,
       num_classes):
        super(LSTMEmotionClassifier, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim,
           padding_idx=word_to_idx['<PAD>'])
        self.lstm = nn.LSTM(embed_dim, hidden_dim, batch_first
           =True)
        self.fc = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):
        x = self.embedding(x)
        lstm_out, _ = self.lstm(x)
        x = lstm_out[:, -1, :]
        return self.fc(x)

    # Evaluation function
def evaluate_model(model, data_loader):
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for texts, labels in data_loader:
            outputs = model(texts)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    return correct / total

def main(BATCH_SIZE = 32, EMBEDDING_DIM = 100, HIDDEN_DIM =
   128, EPOCHS = 5):
    loss_list = []

    encoded_train_labels, label_encoder = encode_labels(
       train_labels)
    encoded_test_labels, _ = encode_labels(test_labels)

    # Data Loaders
    train_dataset = EmotionDataset(train_texts,
       encoded_train_labels, word_to_idx)
    train_loader = DataLoader(train_dataset, batch_size=
       BATCH_SIZE, shuffle=True, collate_fn=collate_fn)

    test_dataset = EmotionDataset(test_texts,
       encoded_test_labels, word_to_idx)
    test_loader = DataLoader(test_dataset, batch_size=
       BATCH_SIZE, shuffle=False, collate_fn=collate_fn)

    # Model initialization
    model = LSTMEmotionClassifier(vocab_size, EMBEDDING_DIM,
       HIDDEN_DIM, len(label_encoder.classes_))
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters())

    # Training loop
    for epoch in range(EPOCHS):
        model.train()
        for texts, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(texts)
            labels = labels.long()  # Convert labels to
               LongTensor
            loss = criterion(outputs, labels)
            loss.backward()
            loss_list.append(loss.item())
            optimizer.step()
        print(f'Epoch■{epoch+1},■Loss:■{loss.item()}')



    # Test the model
    test_accuracy = evaluate_model(model, test_loader)
    print(f'Test■Accuracy:■{test_accuracy*100:.2f}%')

    # Save the model to disk
    torch.save(model.state_dict(), '
       lstm_emotion_classifier_model.pth')

    # Load the model from disk
    loaded_model = LSTMEmotionClassifier(vocab_size,
       EMBEDDING_DIM, HIDDEN_DIM, len(label_encoder.classes_))
    loaded_model.load_state_dict(torch.load('
       lstm_emotion_classifier_model.pth'))

    # Validate the loaded model
    val_texts, val_labels = load_data('data/val.txt')
    encoded_val_labels, _ = encode_labels(val_labels)
    val_dataset = EmotionDataset(val_texts, encoded_val_labels
       , word_to_idx)
    val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE
       , shuffle=False, collate_fn=collate_fn)

    validation_accuracy = evaluate_model(loaded_model,
       val_loader)
    print(f'Validation■Accuracy:■{validation_accuracy*100:.2f
       }%')
    return test_accuracy, loss_list

hyperparameters = [
    {
        'BATCH_SIZE': 32,
        'EMBEDDING_DIM': 100,
        'HIDDEN_DIM': 128,
        'EPOCHS': 5
    },
    {
        'BATCH_SIZE': 64,
        'EMBEDDING_DIM': 100,
        'HIDDEN_DIM': 128,
        'EPOCHS': 5
    },
    {
        'BATCH_SIZE': 32,
        'EMBEDDING_DIM': 200,
        'HIDDEN_DIM': 128,
        'EPOCHS': 5
    },
    {
        'BATCH_SIZE': 32,
        'EMBEDDING_DIM': 100,
        'HIDDEN_DIM': 256,
        'EPOCHS': 5
    },
    {
        'BATCH_SIZE': 32,
        'EMBEDDING_DIM': 100,
        'HIDDEN_DIM': 128,
        'EPOCHS': 10
    },
    {
        'BATCH_SIZE': 32,
        'EMBEDDING_DIM': 100,
        'HIDDEN_DIM': 128,
        'EPOCHS': 20
    },
    {
        'BATCH_SIZE': 32,
        'EMBEDDING_DIM': 100,
        'HIDDEN_DIM': 128,
        'EPOCHS': 30
    },
    {
        'BATCH_SIZE': 32,
        'EMBEDDING_DIM': 100,
        'HIDDEN_DIM': 128,
        'EPOCHS': 40
    },
    {
        'BATCH_SIZE': 32,
        'EMBEDDING_DIM': 100,
        'HIDDEN_DIM': 128,
        'EPOCHS': 50
    },
    {
        'BATCH_SIZE': 32,
        'EMBEDDING_DIM': 100,
        'HIDDEN_DIM': 128,
        'EPOCHS': 100
    }
]

results = []
loss_per_epoch_matrix = []

for hyperparameter in hyperparameters:
    test_accuracy, loss_list = main(**hyperparameter)
    loss_per_epoch_matrix.append(loss_list)
    results.append(test_accuracy)

plt.figure(figsize=(10, 5))
for i in range(len(loss_per_epoch_matrix)):
    plt.plot(range(len(loss_per_epoch_matrix[i])),
       loss_per_epoch_matrix[i], label=f'Hyperparameter■{i+1}'
       )
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.savefig('loss_plot_lstm.png')

plt.figure(figsize=(10, 5))
plt.bar(range(1, len(results)+1), results)
plt.xlabel('Hyperparameter')
plt.ylabel('Accuracy')
plt.savefig('accuracy_bar_lstm.png')

for i in range(len(results)):
    print(f'Hyperparameter■{i+1}:■{results[i]*100:.2f}%')
```

Bibliography

[1] BRILL, E., FLORIAN, R., HENDERSON, J. C., AND MANGU, L. Beyond n-grams: Can linguistic sophistication improve language modeling? Department of Computer Science, Johns Hopkins University, 1998.

[2] DONGARE, A. D., KHARDE, R. R., AND KACHARE, A. D. Introduction to artificial neural network. *International Journal of Engineering and Innovative Technology (IJEIT) 2*, 1 (2012).

[3] GREFENSTETTE, G., AND TAPANAINEN, P. What is a word? what is a sentence? problems of tokenization. Rank Xerox Research Centre, Grenoble Laboratory, 1994.

[4] HICKMAN, L., THAPA, S., TAY, L., CAO, M., AND SRINIVASAN, P. Text preprocessing for text mining in organizational research: Review and recommendations. *Organizational Research Methods 25*, 1 (2020), 114–146.

[5] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural Computation 9*, 8 (1997), 1735–1780.

[6] LIDDY, E. D. *Natural Language Processing*, 2 ed. Marcel Decker Inc, 2001.

[7] MYUNG, I. J. Tutorial on maximum likelihood estimation. *Journal of Mathematical Psychology 47*, 1 (2003), 90–100.

[8] NADKARNI, P. M., OHNO-MACHADO, L., AND CHAPMAN, W. W. Natural language processing: An introduction. *Journal of the American Medical Informatics Association 18*, 5 (2011), 544–551.

[9] SHAREGHI, E., GERZ, D., VULIĆ, I., AND KORHONEN, A. Show some love to your -grams: A bit of progress and stronger -gram language modeling baselines. In *Proceedings of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (2019), vol. 1, pp. 4173–4185.

Curriculum Vitae

Sofía Dalia Magallón Páramo was born in Morelia, Michoacán, México. She is currently completing her Bachelor of Science in Computer Science at the Instituto Tecnológico y de Estudios Superiores de Monterrey, where she is set to graduate in December 2023. She has maintained an impressive GPA of 98/100 and was honored with the Líderes del Mañana Distinction, a recognition for her leadership skills and impactful projects.

During her undergraduate studies, Sofía gained valuable professional experience through multiple internships. In the summer of 2023, she interned at Microsoft in Redmond, Washington, as a Software Engineer, where she developed an automation tool that significantly improved the product update process. She also completed internships at Lyft in Mexico City and Oracle in Zapopan, Jalisco, focusing on software engineering and technical troubleshooting, respectively. In the summer of 2021, Sofía was part of a Production Engineer Internship at Meta and Major League Hacking, where she participated in a 12-week educational program and completed projects using a range of technologies including Flask, PostgreSQL, MongoDB, and React.

Sofía has been actively involved in significant projects, such as NavigAbility, an award-winning accessibility navigation app, and Pedagog, an educational platform created during a hackathon. Her technical expertise spans a variety of programming languages and technologies, including Java, JavaScript, Python, HTML, CSS, Git, Linux, ReactJS, NextJS, Flask, NodeJS, MySQL, MongoDB, PostgreSQL, Docker, and Kubernetes.

In addition to her technical achievements, Sofía has demonstrated outstanding leadership qualities. She has been a GitHub Campus Expert since October 2021, organizing computer science events and mentoring at tech gatherings. As President of TECoding since July 2020, she has led initiatives to engage and grow the tech community at her university. Her role as a Peer Mentor and Community Leader from August 2020 to July 2022 further exemplifies her commitment to supporting and guiding fellow students in computer science.

This document was typed in using $\LaTeX\,2_{\varepsilon}$[a] by Sofía Dalia Magallón Páramo (Student).

[a]The style file `phdThesisFormat.sty` used to set up this thesis was prepared by the Center of Intelligent Systems of the Instituto Tecnológico y de Estudios Superiores de Monterrey, Monterrey Campus